The Vitess Documentation

Contents

Concepts . . . 20
Cell . . . 20
Execution Plans . . . 20
Keyspace ID . . . 21
Keyspace . . . 21
MoveTables . . . 21
    Identifying Candidate Tables . . . 21
Query Rewriting . . . 22
Replication Graph . . . 26
Shard . . . 26
    Shard Naming . . . 26
    Resharding . . . 27
Tablet . . . 27
    Tablet Types . . . 27
Topology Service . . . 28
    Global Topology . . . 28
    Local Topology . . . 28
VSchema . . . 28
VStream . . . 28
vtctl . . . 29
Contribute . . . 30
    Learning Go . . . 30
    Learning Vitess . . . 30
Build on CentOS . . . 31
    Install Dependencies . . . 31
    Build Vitess . . . 32
    Testing your Binaries . . . 32
    Common Build Issues . . . 32
Build on macOS . . . 33
    Install Dependencies . . . 33

Build Vitess . . . 33
Testing your Binaries . . . 33
Common Build Issues . . . 34
Build on Ubuntu/Debian . . . 34
Install Dependencies . . . 34
Build Vitess . . . 35
Testing your Binaries . . . 36
Common Build Issues . . . 36
Coding Standards . . . 37
Backwards Compatibility . . . 37
What does a good PR look like? . . . 37
Assigning a Pull Request . . . 38
Approving a Pull Request . . . 38
Merging a Pull Request . . . 38
GitHub Workflow . . . 39
Remotes . . . 39
Topic Branches . . . 40
Committing your work . . . 40
Sending Pull Requests . . . 40
Addressing Changes . . . 41
Submitting Issues . . . 41
Design Docs . . . 41
Benchmark . . . 41
arewefastyet . . . 41
Background . . . 41
Benchmarking Tool . . . 41
Website . . . 42
Build And Process . . . 42
PR Naming Conventions . . . 42
Background . . . 42
Feature . . . 42
Solution . . . 42
Other Suggestions . . . 43
Call for feedback . . . 44
Query Serving . . . 44
Consistent lookup vindexes . . . 44
Problem . . . 44
Existing solution . . . 44

Approach . . . 44
Implementation . . . 45
Extending the Vindex API . . . 46
New vindexes . . . 46
Corner cases to keep in mind . . . 46
What won't work . . . 46
Locking functions . . . 46

Locking Functions **46**

Functions covered . . . 47
Restrictions . . . 47
Examples of valid queries . . . 47
Examples of queries not supported in the first implementation . . . 47
Owned Primary Vindexes . . . 47
Problem statement . . . 47
Solution . . . 48
Resharding . . . 48
Replica Transactions . . . 48
Reserved Connections . . . 49
Feature Description . . . 49
Set Statements . . . 49
Use Case . . . 50
System variable types . . . 50
Design . . . 50
Release Plan . . . 51
Assumption . . . 51
Table Lifecycle . . . 51
Safe, lazy DROP TABLE . . . 51
Problem statement . . . 51
Illustrating custom DROP workaround steps . . . 52
Suggested implementation: table lifecycle aka table garbage collection . . . 52
Transitioning and skipping of states . . . 53
Asking for a safe DROP . . . 54
VReplication . . . 54
File:Position based VReplication . . . 54
Problem Statement . . . 54
Requirements . . . 54
Design . . . 55
Future improvements . . . 55

Vstream . . . 55
VStream Skew Minimization . . . 55
VStream Skew Detection . . . 55
VStream API and Resharding . . . 56
Stream migration on a resharding operation . . . 57
Schema Tracker . . . 58

Tracking schema changes in Vstreams **58**
Motivation . . . 58
Goals . . . 59
Model . . . 59
Actors . . . 59
Basic Flow for version tracking . . . 59
Flags . . . 60
Caveat . . . 61
VStream Copy . . . 61
VStream Copy . . . 61
Motivation . . . 61
Architecture/Design . . . 62
VTTablet . . . 63
Componentizing TabletServer . . . 63

Introduction **64**

Features **64**
Multi-schema . . . 64
VTShovel . . . 64
VTDirect . . . 65

Requirements **65**
Stats . . . 65
HTTP endpoints . . . 65
Command line flags . . . 66

Design **66**
Dimension Dropper . . . 66
Exporter . . . 66
Config loader . . . 67
TabletManager . . . 67
VReplication . . . 68

Fast vttablet state transitions . . . 68
VTTablet with RBR mode . . . 69
Summary . . . 69
Pass-through DMLs . . . 69
Autocommits . . . 69
Sequences . . . 69
Messages . . . 69
Plan IDs . . . 69
Schema . . . 70
Real-time parameter change . . . 70
Problem Statement . . . 70
Proposal . . . 70
Other options considered . . . 70
VTTablet YAML configuration . . . 70
Background . . . 71
Proposed Approach . . . 71
Detailed specification . . . 71
Implementation . . . 73
The tabletmanager model . . . 74
Background . . . 74
Details . . . 74
Advantages . . . 75
Transition . . . 75
FAQ . . . 75
Where can I ask questions about Vitess? . . . 75
What are the key slack channels to join? . . . 75
How can I contribute a Pull Request to Vitess? . . . 76
What are good videos to watch to get started learning about Vitess? . . . 76
Where can I read additional FAQs? . . . 76
Get Started . . . 76
Helm Chart (deprecated) . . . 76
Prerequisites . . . 76
Start a single keyspace cluster . . . 77
Setup Port-forward . . . 78
Next Steps . . . 79
Local Install via Homebrew . . . 79
Install Vitess with Homebrew . . . 79
Start a Single Keyspace Cluster . . . 80

Setup Aliases . . . 82
Connect to your cluster . . . 82
Summary . . . 82
Next Steps . . . 83
Local Install via Docker . . . 83
Check out the vitessio/vitess repository . . . 83
Build the docker image . . . 83
Run the docker image . . . 84
Summary . . . 84
Next Steps . . . 85
Local Install . . . 85
Install MySQL and etcd . . . 85
Disable AppArmor or SELinux . . . 85
Install Vitess . . . 86
Start a Single Keyspace Cluster . . . 86
Setup Aliases . . . 87
Connect to your cluster . . . 87
Summary . . . 88
Next Steps . . . 88
Vitess Operator for Kubernetes . . . 88
Prerequisites . . . 89
Install the Operator . . . 89
Bring up an initial cluster . . . 89
Setup Port-forward . . . 90
Create Schema . . . 90
Next Steps . . . 91
Vttestserver Docker Image . . . 91
Get the docker image . . . 91
Run the docker image . . . 92
Example . . . 93
Overview . . . 93
Architecture . . . 93
Cloud Native . . . 93
Vitess on Kubernetes . . . 93
History . . . 94
Vitess becomes a CNCF project . . . 95
Scalability Philosophy . . . 95
Small instances . . . 95

Durability through replication . . . 95
Consistency model . . . 95
Multi-cell . . . 96
Supported Databases . . . 97
MySQL versions 5.6 to 8.0 . . . 97
MariaDB versions 10.0 to 10.3 . . . 97
See also . . . 97
What Is Vitess . . . 97
Features . . . 98
Comparisons to other storage options . . . 98
Older Version Docs . . . 99
Reference . . . 99
Compatibility . . . 99
MySQL Compatibility . . . 99
Transaction Model . . . 100
SQL Syntax . . . 100
Network Protocol . . . 101
Temporary Tables . . . 101
Character Set and Collation . . . 102
SQL Mode . . . 102
Data Types . . . 102
Auto Increment . . . 102
Extensions to MySQL Syntax . . . 102
Errors . . . 104
Query Serving . . . 104
Features . . . 108
Messaging . . . 108
Creating a message table . . . 108
Enqueuing messages . . . 109
Receiving messages . . . 109
Acknowledging messages . . . 110
Exponential backoff . . . 110
Purging . . . 110
Advanced usage . . . 110
Undocumented features . . . 111
Known limitations . . . 111
Monitoring . . . 111

Current state of monitoring **111**

1. Vitess status pages . . . 111
2. Pull-based metrics system . . . 111
3. Push-based metrics system . . . 111

Monitoring with Kubernetes **112**

Replication . . . 112
Semi-Sync . . . 112
Database Schema Considerations . . . 113
Point In Time Recovery . . . 113
Point in Time Recovery . . . 113
Schema Management . . . 115
Reviewing your schema . . . 116
Changing your schema . . . 116
Schema Routing Rules . . . 118
ApplyRoutingRules . . . 118
Syntax . . . 118
Schema Tracking . . . 119
VTGate . . . 119
VTTablet . . . 119
Sharding . . . 119
Overview . . . 120
Sharding scheme . . . 120
Resharding . . . 121
Table lifecycle . . . 121
Problems with DROP TABLE . . . 121
Vitess table lifecycle . . . 122
Lifecycle subsets and configuration . . . 122
Stateless flow by table name hints . . . 123
Automated lifecycle . . . 123
User-facing DROP TABLE lifecycle . . . 123
Tablet throttler . . . 123
Why throttler: maintaining shard health via low replication lag . . . 123
Throttler overview . . . 124
Custom metrics & queries . . . 125
Configuration . . . 125
API & usage . . . 126
Resources . . . 127
Topology Service . . . 128

Requirements and usage . . . 128
Global data . . . 128
Local data . . . 129
Workflows involving the Topology Service . . . 130
Exploring the data in a Topology Service . . . 130
Implementations . . . 131
Running in only one cell . . . 133
Migration between implementations . . . 134
Transport Security Model . . . 136
Overview . . . 136
Caller ID . . . 136
gRPC Transport . . . 137
MySQL Transport to VTGate . . . 139
Two-Phase Commit . . . 139
Isolation . . . 139
Driver APIs . . . 140
Configuring VTTablet . . . 140
Configuring MySQL . . . 140
Monitoring . . . 140
Critical failures . . . 141
Alertable failures . . . 141
Repairs . . . 141
Vindexes . . . 141
A Vindex maps column values to keyspace IDs . . . 141
Advantages . . . 141
Sequences . . . 147
Motivation . . . 147
When *not* to Use Auto-Increment . . . 147
MySQL Auto-increment Feature . . . 148
Vitess Sequences . . . 148
VSchema . . . 149
Overview . . . 149
Architecture . . . 150
Database Access Model . . . 150
Sharded keyspaces require a VSchema . . . 151
Sharding Model . . . 151
Vindexes . . . 151
Sequences . . . 152

Reference tables . . . 152
Per-Keyspace VSchema . . . 152
Routing Rules . . . 156
Programs . . . 157
mysqlctl . . . 157
Commands . . . 157
Options . . . 158
vtctl . . . 167
Commands . . . 167
Options . . . 171
vtctl Cell Aliases Command Reference . . . 179
Commands . . . 179
See Also . . . 180
vtctl Cell Command Reference . . . 180
Commands . . . 181
See Also . . . 182
vtctl Generic Command Reference . . . 182
Commands . . . 183
See Also . . . 184
vtctl Keyspace Command Reference . . . 184
Commands . . . 184
See Also . . . 194
vtctl Query Command Reference . . . 194
Commands . . . 194
See Also . . . 197
vtctl Replication Graph Command Reference . . . 198
Commands . . . 198
See Also . . . 198
vtctl Resharding Throttler Command Reference . . . 198
Commands . . . 198
See Also . . . 201
vtctl Schema, Version, Permissions Command Reference . . . 201
Commands . . . 201
See Also . . . 208
vtctl Serving Graph Command Reference . . . 208
Commands . . . 208
See Also . . . 210
vtctl Shard Command Reference . . . 210

Commands . . . 210
See Also . . . 219
vtctl Tablet Command Reference . . . 219
Commands . . . 219
See Also . . . 228
vtctl Topo Command Reference . . . 228
Commands . . . 228
See Also . . . 229
vtctl Workflow Command Reference . . . 229
Commands . . . 229
See Also . . . 230
vtctld . . . 230
Example Usage . . . 231
Options . . . 231
vtexplain . . . 239
Example Usage . . . 239
Options . . . 239
Limitations . . . 240
vtgate . . . 240
Example Usage . . . 240
Options . . . 241
vttablet . . . 245
Example Usage . . . 245
Options . . . 246
VReplication . . . 256
— Cancel . . . 256
— Complete . . . 256
“— Create” . . . 257
Flags . . . 259
Life of a stream . . . 261
Materialize . . . 264
Metrics . . . 266
Target Metrics . . . 266
Source Metrics . . . 267
Example . . . 267
Migrate . . . 268
Mount . . . 269
MoveTables . . . 270

— Progress . . . 271
Reshard . . . 271
— ReverseTraffic . . . 272
— Show . . . 273
— SwitchTraffic . . . 273
Throttling . . . 274
VReplication V1 Commands . . . 274
DropSources . . . 275
MoveTables . . . 275
Reshard . . . 278
SwitchReads . . . 279
SwitchWrites . . . 280
VReplication V2 commands . . . 281
— Cancel . . . 281
— Complete . . . 281
“— Create” . . . 282
MoveTables . . . 284
— Progress . . . 285
Reshard . . . 286
— ReverseTraffic . . . 286
— Show . . . 287
— SwitchTraffic . . . 287
VDiff . . . 288
VExec . . . 291
Overview . . . 291
Feature description . . . 292
VReplicationExec . . . 293
Other properties of VReplication . . . 296
Monitoring and troubleshooting . . . 297
VReplicationExec . . . 297
Workflow . . . 298
Resources . . . 299
Presentations and Videos . . . 299
Rawkode Live . . . 299
Percona FOSS Talks 2021 . . . 299
FOSDEM 2021 . . . 299
KubeCon NA 2020 . . . 299
Perconalive Europe Online 2020 . . . 299

CNCF Webinars 2020 . . . 300
Quarantine 2020 Database Talks, Carnegie Mellon University . . . 300
KubeCon Europe 2020 . . . 300
Online Demo . . . 300
Perconalive NA Online 2020 . . . 300
CNCF Webinar 2020 . . . 300
MySQL Pre-FOSDEM Day 2020 . . . 300
QCon 2019 . . . 301
KubeCon San Diego 2019 . . . 301
Highload 2019 . . . 301
Utah Kubernetes Meetup 2019 . . . 301
CNCF Meetup Paris 2019 . . . 301
Percona Live Europe 2019 . . . 301
Vitess Meetup 2019 @ Slack HQ . . . 302
Cloud Native Show 2019 . . . 302
CNCF Webinar 2019 . . . 302
Kubecon China 2019 . . . 302
RootConf 2019 . . . 302
Kubecon 19 Barcelona . . . 303
Percona Live Austin 2019 . . . 303
Velocity New York 2018 . . . 303
Percona Live Europe 2017 . . . 303
Vitess Deep Dive sessions . . . 304
Percona Live 2016 . . . 304
CoreOS Meetup, January 2016 . . . 304
Oracle OpenWorld 2015 . . . 304
Percona Live 2015 . . . 304
Google I/O 2014 - Scaling with Go: YouTube's Vitess . . . 304
Vitess Roadmap . . . 305
Short Term (1-3 months) . . . 305
Medium Term (3-9 months) . . . 305
Troubleshoot . . . 306
Elevated query latency on master . . . 306
Elevated query latency on master . . . 306
Master starts up read-only . . . 307
Master starts up read-only . . . 307
Vitess sees the wrong tablet as master . . . 307
Vitess sees the wrong tablet as master . . . 307

User Guides . . . 307
Advanced Configuration . . . 307
Authorization . . . 307
VTTablet parameters for table ACLs . . . 308
Format of the table ACL config file . . . 308
Example . . . 309
CreateLookupVindex . . . 310
Integration with Orchestrator . . . 319
Orchestrator configuration . . . 320
VTTablet configuration . . . 320
LDAP authentication . . . 320
Requirements . . . 321
Configuration . . . 321
Query Consolidation . . . 322
Consistency . . . 323
Region-based Sharding . . . 323
Preparation . . . 323
Schema . . . 323
Region Vindex . . . 323
Start the Cluster . . . 324
Aliases . . . 325
Connect to your cluster . . . 326
Insert some data into the cluster . . . 326
Examine the data we just inserted . . . 326
Prepare for resharding . . . 327
Perform Resharding . . . 331
Cutover . . . 332
Drop source . . . 333
Teardown . . . 333
Reparenting . . . 333
MySQL requirements . . . 333
External Reparenting . . . 334
Fixing Replication . . . 335
Resharding . . . 335
Preparation . . . 335
Apply VSchema . . . 337
Create new shards . . . 338
Start the Reshard . . . 338

Validate Correctness . . . 339
Switch Non-Primary Reads . . . 339
Switch Writes and Primary Reads . . . 339
Note . . . 339
Finalize and Cleanup . . . 340
Tracing . . . 340

Vitess tracing **341**
Configuring tracing . . . 341
Unmanaged Tablet . . . 343
Ensure all components are up . . . 343
Start a tablet to correspond to legacy . . . 343
Connect via VTGate . . . 344
Move legacytable to the commerce keyspace . . . 344
User Management and Authentication . . . 345
Authentication . . . 345
Password format . . . 346
UserData . . . 347
Multiple passwords . . . 347
Other authentication methods . . . 348
Running in Production . . . 348
Add or Delete a Cell . . . 348
Add . . . 348
Delete . . . 348
Backups and Restores . . . 348
Creating a cell . . . 349
Mapping cells to zones and regions . . . 349
Checklist . . . 350
Delete a Keyspace . . . 351
Exporting data from Vitess . . . 351
Global TopoServer . . . 352
Choosing a TopoRoot . . . 352
Moving to a different TopoServer . . . 353
Backups . . . 353
Checklist . . . 353
InitShardMaster . . . 353
Keyspaces and Shards . . . 353
Monitoring . . . 355
Tools . . . 355

VTTablet . . . 355
VTGate . . . 360
Planning . . . 361
External tools . . . 361
Provisioning . . . 362
Environment variables . . . 362
Backups . . . 362
Logging . . . 362
Ports . . . 362

Ports and Network interactions in Vitess **362**

Reparenting . . . 363
Troubleshooting . . . 365
Understanding the Components . . . 365
Query Serving . . . 365
vtctld . . . 369
vtctlclient . . . 369
vtgate . . . 369
vtorc (experimental) . . . 371
VTTablet and MySQL . . . 373
Starting MySQL . . . 373
Starting vttablet . . . 376
Tablet Records . . . 379
Legacy . . . 380
description: User guides for features in older version of Vitess . . . 380
Horizontal Sharding . . . 380
Preparation . . . 380
Create new shards . . . 382
SplitClone . . . 383
Cut over . . . 384
Clean up . . . 385
Next Steps . . . 385
Vertical Split . . . 385
Create Keyspace . . . 386
Customer Tablets . . . 386
VerticalSplitClone . . . 386
Cut over . . . 387
Clean up . . . 388
Next Steps . . . 388

Migration . . . 388
Materialize . . . 388
Planning to use Materialize . . . 389
Create the destination tables . . . 389
Start the Materialize (first copy) . . . 390
Viewing the workflow while in progress . . . 391
Start the Materialize (redacted copy) . . . 392
What happened under the covers . . . 393
Cleanup . . . 394
Recap . . . 395
Migrating data into Vitess . . . 395

Introduction **395**

Overview . . . 395
Method 1: “Stop-the-world”: . . . 396
Method 2: VReplication from Vitess setup in front of the existing external MySQL database . . . 396
Method 3: Application-level migration . . . 397
MoveTables . . . 397
Planning to Move Tables . . . 398
Show our current tablets . . . 398
Create new tablets . . . 399
Show our old and new tablets . . . 400
Start the Move . . . 400
Check routing rules (optional) . . . 401
Monitoring Progress (optional) . . . 402
Validate Correctness (optional) . . . 402
Phase 1: Switch Non-Primary Reads . . . 402
Interlude: check the routing rules (optional) . . . 402
Phase 2: Switch Writes and Primary Reads . . . 404
Note . . . 404
Interlude: check the routing rules (optional) . . . 405
Reverse workflow . . . 405
Finalize and Cleanup . . . 406
Next Steps . . . 406
Operational . . . 406
Backup and Restore . . . 406
Backup and Restore . . . 407
Concepts . . . 407
VTTablet Configuration . . . 407

Creating a Backup . . . 410
Creating a backup . . . 410
Restoring a backup . . . 411
Managing backups . . . 411
Bootstrapping a new tablet . . . 412
Common Errors and Resolutions . . . 412
Backing up Topology Server . . . 413
Upgrading Vitess . . . 413
Compatibility . . . 413
Upgrade Order . . . 413
Canary Testing . . . 414
Rolling Upgrades . . . 414
Upgrading the Master Tablet . . . 414
Making Schema Changes . . . 414
The schema change problem . . . 414
ALTER TABLE solutions . . . 415
Schema change cycle and operation . . . 415
Schema change and Vitess . . . 416
The various approaches . . . 416
Applying, auditing, and controlling Online DDL . . . 417
Running migrations . . . 417
Tracking migrations . . . 418
Showing migration logs . . . 422
Cancelling a migration . . . 422
Cancelling all keyspace migrations . . . 425
Retrying a migration . . . 426
Reverting a migration . . . 428
VExec commands for greater control and visibility . . . 429
Online DDL strategies . . . 432
Specifying a DDL strategy . . . 432
Choosing a DDL strategy . . . 432
Vitess functionality comparison . . . 435
Declarative migrations . . . 435
Usage . . . 436
Implementation details . . . 437
Managed, Online Schema Changes . . . 437
Syntax . . . 438
ddl_strategy . . . 439

Running, tracking and controlling Online DDL . . . 439
Migration flow and states . . . 440
Configuration . . . 440
Auto resume after failure . . . 441
Auto retry after failure . . . 441
Throttling . . . 441
Table cleanup . . . 442
Revertible migrations . . . 442
Behavior and limitations . . . 442
REVERT syntax . . . 443
Usage & walkthrough . . . 443
Implementation details . . . 446
Unmanaged Schema Changes . . . 447
ApplySchema . . . 447
VTGate . . . 448
Directly to MySQL . . . 448
SQL Statement Analysis . . . 448
Analyzing SQL statements in bulk . . . 449

Introduction **449**

Prerequisites . . . 449
Overview . . . 449
1. Gather the queries from your current MySQL database environment . . . 449
2. Filter out specific queries . . . 449
3. Populate fake values for your queries . . . 450
4. Run the VTexplain tool via a script . . . 451
5. Add your SQL schema to the output file . . . 451
6. Add your VSchema . . . 451
7. Run the VTexplain tool and capture the output . . . 452
8. Check your output . . . 452
Analyzing a SQL statement . . . 453

Introduction **453**

Prerequisites . . . 453
Overview . . . 453
1. Identify a SQL schema for tables in the statement . . . 453
2. Identify a VSchema for the statement's source tables . . . 454
3. Run the VTExplain tool . . . 455
See also . . . 458

VSchema and Query Serving . . . 458
Advanced VSchema Properties . . . 458
Multi-Column Vindexes . . . 458
Reference Tables . . . 459
Column List . . . 459
Routing Rules . . . 461
Lookup as Primary Vindex . . . 461
Non-Unique Lookup Vindexes . . . 463
Overview . . . 466
Pictorial Summary . . . 467
Sequences . . . 467
Sharded Keyspace . . . 470
Sharding Guidelines . . . 472
Shared Vindexes and Foreign Keys . . . 474
Unique Lookup Vindexes . . . 477
Unsharded Keyspace . . . 480

Concepts

description: Learn core Vitess concepts and terminology

Cell

description: Data center, availability zone or group of computing resources

A *cell* is a group of servers and network infrastructure collocated in an area, and isolated from failures in other cells. It is typically either a full data center or a subset of a data center, sometimes called a *zone* or *availability zone*. Vitess gracefully handles cell-level failures, such as when a cell is cut off the network.

Each cell in a Vitess implementation has a local topology service, which is hosted in that cell. The topology service contains most of the information about the Vitess tablets in its cell. This enables a cell to be taken down and rebuilt as a unit.

Vitess limits cross-cell traffic for both data and metadata. While it may be useful to also have the ability to route read traffic to individual cells, Vitess currently serves reads only from the local cell. Writes will go cross-cell when necessary, to wherever the master for that shard resides.

Execution Plans

Vitess parses queries at both the VTGate and VTTablet layer in order to evaluate the best method to execute a query. This evaluation is known as query planning, and results in a *query execution plan*.

The Execution Plan is dependent on both the query and the associated VSchema. One of the underlying goals of Vitess' planning strategy is to push down as much work as possible to the underlying MySQL instances. When this is not possible, Vitess will use a plan that collects input from multiple sources and merges the results to produce the correct query result.

Evaluation Model An execution plan consists of operators, each of which implements a specific piece of work. The operators combine into a tree-like structure, which represents the overall execution plan. The plan represents each operator as a node in the tree. Each operator takes as input zero or more rows, and produces as output zero or more rows. This means that the output from one operator becomes the input for the next operator. Operators that join two branches in the tree combine input from two incoming streams and produce a single output.

Evaluation of the execution plan begins at the leaf nodes of the tree. Leaf nodes pull in data from VTTablet, the Topology Service, and in some cases are also able to evaluate expression values locally. Each leaf node will not have input from other operators, and pipe in any nodes they produce into their parent nodes. The parents nodes will then pipe in nodes to their parent nodes, all the way up to a root node. The root node produces the final results of the query and delivers the results to the user.

Observing Execution Plans Cached execution plans can be observed at the VTGate level by browsing the `/queryz` end point.

Starting with Vitess 6, individual statement plans can also be observed with `EXPLAIN FORMAT=vitess <query>`.

Related Vitess Documentation

- VTGate

Keyspace ID

The *keyspace ID* is the value that is used to decide on which shard a given row lives. Range-based Sharding refers to creating shards that each cover a particular range of keyspace IDs.

Using this technique means you can split a given shard by replacing it with two or more new shards that combine to cover the original range of keyspace IDs, without having to move any records in other shards.

The keyspace ID itself is computed using a function of some column in your data, such as the user ID. Vitess allows you to choose from a variety of functions (vindexes) to perform this mapping. This allows you to choose the right one to achieve optimal distribution of the data across shards.

Keyspace

A *keyspace* is a logical database. If you're using sharding, a keyspace maps to multiple MySQL databases; if you're not using sharding, a keyspace maps directly to a MySQL database name. In either case, a keyspace appears as a single database from the standpoint of the application.

Reading data from a keyspace is just like reading from a MySQL database. However, depending on the consistency requirements of the read operation, Vitess might fetch the data from a master database or from a replica. By routing each query to the appropriate database, Vitess allows your code to be structured as if it were reading from a single MySQL database.

MoveTables

MoveTables is a new workflow based on VReplication. It enables you to relocate tables between keyspaces, and therefore physical MySQL instances, without downtime.

Identifying Candidate Tables

It is recommended to keep tables that need to join on each other in the same keyspace, so typical candidates for a MoveTables operation are a set of tables which logically group together or are otherwise isolated.

If you have multiple groups of tables as candidates, which makes the most sense to move may depend on the specifics of your environment. For example, a larger table will take more time to move, but in doing so you might be able to utilitize additional or newer hardware which has more headroom before you need to perform additional operations such as sharding.

Similarly, tables that are updated at a more frequent rate could increase the move time.

Impact to Production Traffic Internally, a MoveTables operation is comprised of both a table copy and a subscription to all changes made to the table. Vitess uses batching to improve the performance of both table copying and applying subscription changes, but you should expect that tables with lighter modification rates to move faster.

During the active move process, data is copied from replicas instead of the master server. This helps ensure minimal production traffic impact.

During the `SwitchWrites` phase of the MoveTables operation, Vitess may be briefly unavailable. This unavailability is usually a few seconds, but will be higher in the event that your system has a high replication delay from master to replica(s).

Related Vitess Documentation

- MoveTables User Guide

Query Rewriting

Vitess works hard to create an illusion of the user having a single connection to a single database. In reality, a single query might interact with multiple databases and may use multiple connections to the same database. Here we'll go over what Vitess does and how it might impact you.

Query splitting A complicated query with a cross shard join might need to first fetch information from a tablet keeping vindex lookup tables. Then use this information to query two different shards for more data and subsequently join the incoming results into a single result that the user receives. The queries that MySQL gets are often just pieces of the original query, and the final result will get assembled at the vtgate level.

Connection Pooling When a tablet talks with a MySQL to execute a query on behalf of a user, it does not use a dedicated connection per user, and instead will share the underlying connection between users. This means that it's not safe to store any state in the session as you can't be sure it will continue executing queries on the same connection, and you can't be sure if this connection will be used by other users later on.

User-Defined Variables User defined variables are kept in the session state when working with MySQL. You can assign values to them using SET:

```
SET @my_user_variable = 'foobar'
```

And later there can be queries using for example SELECT:

```
> SELECT @my_user_variable;
+--------------------+
| @my_user_variable |
+--------------------+
| foobar             |
+--------------------+
```

If you execute these queries against a VTGate, the first `SET` query is not sent to MySQL. Instead, it is evaluated in the VTGate, and VTGate will keep this state for you. The second query is also not sent down. Trivial queries such as this one are actually fully executed on VTGate.

If we try a more complicated query that requires data from MySQL, VTGate will rewrite the query before sending it down. If we were to write something like:

```
WHERE col = @my_user_variable
```

What MySQL will see is:

```
WHERE col = 'foobar'
```

This way, no session state is needed to evaluate the query in MySQL.

Server System Variables A user might also want to change one of the many different system variables that MySQL exposes. Vitess handles system variables in one of four different ways:

- *No op*. For some settings, Vitess will just silently ignore the setting. This is for system variables that don't make much sense in a sharded setting, and don't change the behaviour of MySQL in an interesting way.
- *Check and fail if not already set*. These are settings that should not change, but Vitess will allow SET statements that try to set the variable to whatever it already is.
- *Not supported*. For these settings, attempting to change them will always result in an error.
- *Vitess aware*. These are settings that change Vitess' behaviour, and are not sent down to MySQL
- *Reserved connection*. For some settings, it makes sense to allow them to be set, but it also means that we can't use a shared connection for this user. What this means is that every connection done on this users behalf will need to first have these system variables set, and then keep the connection dedicated. Connection pooling is important for the performance of Vitess, and reserved connections can't be pooled, so this should not be the normal way to run applications on Vitess. Just make sure that the global variable is set to the same value the application will set it to, and Vitess can use connection pooling.

In addition to this, Vitess makes sure that @@version includes both the emulated MySQL version and the Vitess version, such as: `5.7.9-vitess-10.0.0-SNAPSHOT`. This value can be changed by using the vtgate flag `-mysql_server_version`.

Special functions There are a few special functions that Vitess handles without delegating to MySQL.

- `DATABASE()` - The keyspace name and the underlying database names do not have to be equal. Vitess will rewrite these calls to use the literal string for the keyspace name. (This also applies to the synonym `SCHEMA()`)
- `ROW_COUNT()` and `FOUND_ROWS()` - These functions returns how many rows the last query affected/returned. Since this might have been executed on a different connection, these get rewritten to use the literal value of the number of returned rows.
- `LAST_INSERT_ID()` - Much like `FOUND_ROWS()`, we can't trust a pooled connection for these function calls, so they get rewritten before hitting MySQL.

Reference Here is a list of all the system variables that are handled by Vitess and how they are handled.

System variable	*Handled*
autocommit	VitessAware
client_found_rows	VitessAware
skip_query_plan_cache	VitessAware
tx_read_only	VitessAware
transaction_read_only	VitessAware
sql_select_limit	VitessAware
transaction_mode	VitessAware
workload	VitessAware
charset	VitessAware
names	VitessAware
big_tables	NoOp
bulk_insert_buffer_size	NoOp
debug	NoOp
default_storage_engine	NoOp
default_tmp_storage_engine	NoOp
innodb_strict_mode	NoOp
innodb_support_xa	NoOp
innodb_table_locks	NoOp
innodb_tmpdir	NoOp
join_buffer_size	NoOp
keep_files_on_create	NoOp
lc_messages	NoOp

System variable	*Handled*
long_query_time	NoOp
low_priority_updates	NoOp
max_delayed_threads	NoOp
max_insert_delayed_threads	NoOp
multi_range_count	NoOp
net_buffer_length	NoOp
new	NoOp
query_cache_type	NoOp
query_cache_wlock_invalidate	NoOp
query_prealloc_size	NoOp
sql_buffer_result	NoOp
transaction_alloc_block_size	NoOp
wait_timeout	NoOp
audit_log_read_buffer_size	NotSupported
auto_increment_increment	NotSupported
auto_increment_offset	NotSupported
binlog_direct_non_transactional_updates	NotSupported
binlog_row_image	NotSupported
binlog_rows_query_log_events	NotSupported
innodb_ft_enable_stopword	NotSupported
innodb_ft_user_stopword_table	NotSupported
max_points_in_geometry	NotSupported
max_sp_recursion_depth	NotSupported
myisam_repair_threads	NotSupported
myisam_sort_buffer_size	NotSupported
myisam_stats_method	NotSupported
ndb_allow_copying_alter_table	NotSupported
ndb_autoincrement_prefetch_sz	NotSupported
ndb_blob_read_batch_bytes	NotSupported
ndb_blob_write_batch_bytes	NotSupported
ndb_deferred_constraints	NotSupported
ndb_force_send	NotSupported
ndb_fully_replicated	NotSupported
ndb_index_stat_enable	NotSupported
ndb_index_stat_option	NotSupported
ndb_join_pushdown	NotSupported
ndb_log_bin	NotSupported
ndb_log_exclusive_reads	NotSupported
ndb_row_checksum	NotSupported
ndb_use_exact_count	NotSupported
ndb_use_transactions	NotSupported
ndbinfo_max_bytes	NotSupported
ndbinfo_max_rows	NotSupported
ndbinfo_show_hidden	NotSupported
ndbinfo_table_prefix	NotSupported
old_alter_table	NotSupported
preload_buffer_size	NotSupported
rbr_exec_mode	NotSupported
sql_log_off	NotSupported
thread_pool_high_priority_connection	NotSupported
thread_pool_prio_kickup_timer	NotSupported
transaction_write_set_extraction	NotSupported
default_week_format	ReservedConn
end_markers_in_json	ReservedConn
eq_range_index_dive_limit	ReservedConn

System variable	*Handled*
explicit_defaults_for_timestamp	ReservedConn
foreign_key_checks	ReservedConn
group_concat_max_len	ReservedConn
max_heap_table_size	ReservedConn
max_seeks_for_key	ReservedConn
max_tmp_tables	ReservedConn
min_examined_row_limit	ReservedConn
old_passwords	ReservedConn
optimizer_prune_level	ReservedConn
optimizer_search_depth	ReservedConn
optimizer_switch	ReservedConn
optimizer_trace	ReservedConn
optimizer_trace_features	ReservedConn
optimizer_trace_limit	ReservedConn
optimizer_trace_max_mem_size	ReservedConn
transaction_isolation	ReservedConn
tx_isolation	ReservedConn
optimizer_trace_offset	ReservedConn
parser_max_mem_size	ReservedConn
profiling	ReservedConn
profiling_history_size	ReservedConn
query_alloc_block_size	ReservedConn
range_alloc_block_size	ReservedConn
range_optimizer_max_mem_size	ReservedConn
read_buffer_size	ReservedConn
read_rnd_buffer_size	ReservedConn
show_create_table_verbosity	ReservedConn
show_old_temporals	ReservedConn
sort_buffer_size	ReservedConn
sql_big_selects	ReservedConn
sql_mode	ReservedConn
sql_notes	ReservedConn
sql_quote_show_create	ReservedConn
sql_safe_updates	ReservedConn
sql_warnings	ReservedConn
tmp_table_size	ReservedConn
transaction_prealloc_size	ReservedConn
unique_checks	ReservedConn
updatable_views_with_limit	ReservedConn
binlog_format	CheckAndIgnore
block_encryption_mode	CheckAndIgnore
character_set_client	CheckAndIgnore
character_set_connection	CheckAndIgnore
character_set_database	CheckAndIgnore
character_set_filesystem	CheckAndIgnore
character_set_results	CheckAndIgnore
character_set_server	CheckAndIgnore
collation_connection	CheckAndIgnore
collation_database	CheckAndIgnore
collation_server	CheckAndIgnore
completion_type	CheckAndIgnore
div_precision_increment	CheckAndIgnore
innodb_lock_wait_timeout	CheckAndIgnore
interactive_timeout	CheckAndIgnore
lc_time_names	CheckAndIgnore

System variable	*Handled*
lock_wait_timeout	CheckAndIgnore
max_allowed_packet	CheckAndIgnore
max_error_count	CheckAndIgnore
max_execution_time	CheckAndIgnore
max_join_size	CheckAndIgnore
max_length_for_sort_data	CheckAndIgnore
max_sort_length	CheckAndIgnore
max_user_connections	CheckAndIgnore
net_read_timeout	CheckAndIgnore
net_retry_count	CheckAndIgnore
net_write_timeout	CheckAndIgnore
session_track_gtids	CheckAndIgnore
session_track_schema", boolean:	CheckAndIgnore
session_track_state_change", boolean:	CheckAndIgnore
session_track_system_variables	CheckAndIgnore
session_track_transaction_info	CheckAndIgnore
sql_auto_is_null", boolean:	CheckAndIgnore
time_zone	CheckAndIgnore
version_tokens_session	CheckAndIgnore

Related Vitess Documentation

- VTGate

Replication Graph

The *replication graph* identifies the relationships between master databases and their respective replicas. During a master failover, the replication graph enables Vitess to point all existing replicas to a newly designated master database so that replication can continue.

Shard

A *shard* is a subset of a keyspace. A keyspace will always contain one or more shards. A shard typically contains one MySQL master and potentially many MySQL replicas.

Each MySQL instance within a shard has identical data (if we ignore any replication lag). The replicas can serve read-only traffic (with eventual consistency guarantees), execute long-running data analysis queries, or perform administrative tasks (backup, restore, diff, etc.).

An *unsharded* keyspace is a keyspace with only a single shard. Vitess names the shard `0` (or sometimes `-`) by convention. When sharded, a keyspace has `N` shards with non-overlapping data. The number of shards in a keyspace can vary depending on the use-case and load characteristics, some Vitess users have hundreds of shards in some keyspaces.

Shard Naming

Shard names have the following characteristics:

- They represent a range in unsigned integer space, where the left number is included, but the right is not.
- Their notation is hexadecimal.
- They are left justified, and right-padded with zeros.
- A `-` prefix means: anything less than the right value.
- A `-` postfix means: anything greater than or equal to the LHS value.

- A plain `-` denotes the full keyrange.

Thus: `-80` == `00-80` == `0000-8000` == `000000-800000` == `0000000000000000-8000000000000000`

`80-` is not the same as `80-FF`. This is why:

`80-FF` == `8000-FF00`. Therefore `FFFF` will be out of the `80-FF` range.

`80-` means: 'anything greater than or equal to `0x80`

A `hash` type vindex produces an unsigned 64 bit integer as output. This means that all integers **less** than `0x8000000000000000` will fall in shard `-80`. Any number with the highest bit set will be >= `0x8000000000000000`, and will therefore belong to shard `80-`.

This left-justified approach allows you to have keyspace ids of arbitrary length. However, the most significant bits are the ones on the left.

For example an `md5` hash produces 16 bytes. That can also be used as a keyspace id.

A `varbinary` of arbitrary length can also be mapped as is to a keyspace id. This is what the `binary` vindex does.

Resharding

Vitess supports resharding, in which the number of shards is changed on a live cluster. This can be either splitting one or more shards into smaller pieces, or merging neighboring shards into bigger pieces.

During resharding, the data in the source shards is copied into the destination shards, allowed to catch up on replication, and then compared against the original to ensure data integrity. Then the live serving infrastructure is shifted to the destination shards, and the source shards are deleted.

Related Vitess Documentation

- Resharding User Guide

Tablet

A *tablet* is a combination of a `mysqld` process and a corresponding `vttablet` process, usually running on the same machine. Each tablet is assigned a *tablet type*, which specifies what role it currently performs.

Queries are routed to a tablet via a VTGate server.

Tablet Types

See the user guide VTTablet Modes for more information.

- **master** - A *replica* tablet that happens to currently be the MySQL master for its shard.
- **replica** - A MySQL replica that is eligible to be promoted to *master*. Conventionally, these are reserved for serving live, user-facing requests (like from the website's frontend).
- **rdonly** - A MySQL replica that cannot be promoted to *master*. Conventionally, these are used for background processing jobs, such as taking backups, dumping data to other systems, heavy analytical queries, MapReduce, and resharding.
- **backup** - A tablet that has stopped replication at a consistent snapshot, so it can upload a new backup for its shard. After it finishes, it will resume replication and return to its previous type.
- **restore** - A tablet that has started up with no data, and is in the process of restoring itself from the latest backup. After it finishes, it will begin replicating at the GTID position of the backup, and become either *replica* or *rdonly*.
- **drained** - A tablet that has been reserved by a Vitess background process (such as rdonly tablets for resharding).

Topology Service

description: Also known as the TOPO or lock service

The *Topology Service* is a set of backend processes running on different servers. Those servers store topology data and provide a distributed locking service.

Vitess uses a plug-in system to support various backends for storing topology data, which are assumed to provide a distributed, consistent key-value store. The default topology service plugin is `etcd2`.

The topology service exists for several reasons:

- It enables tablets to coordinate among themselves as a cluster.
- It enables Vitess to discover tablets, so it knows where to route queries.
- It stores Vitess configuration provided by the database administrator that is needed by many different servers in the cluster, and that must persist between server restarts.

A Vitess cluster has one global topology service, and a local topology service in each cell.

Global Topology

The global topology service stores Vitess-wide data that does not change frequently. Specifically, it contains data about keyspaces and shards as well as the master tablet alias for each shard.

The global topology is used for some operations, including reparenting and resharding. By design, the global topology service is not used a lot.

In order to survive any single cell going down, the global topology service should have nodes in multiple cells, with enough to maintain quorum in the event of a cell failure.

Local Topology

Each local topology contains information related to its own cell. Specifically, it contains data about tablets in the cell, the keyspace graph for that cell, and the replication graph for that cell.

The local topology service must be available for Vitess to discover tablets and adjust routing as tablets come and go. However, no calls to the topology service are made in the critical path of serving a query at steady state. That means queries are still served during temporary unavailability of topology.

VSchema

A VSchema allows you to describe how data is organized within keyspaces and shards. This information is used for routing queries, and also during resharding operations.

For a Keyspace, you can specify if it's sharded or not. For sharded keyspaces, you can specify the list of vindexes for each table.

Vitess also supports sequence generators that can be used to generate new ids that work like MySQL auto increment columns. The VSchema allows you to associate table columns to sequence tables. If no value is specified for such a column, then VTGate will know to use the sequence table to generate a new value for it.

VStream

VStream is a change notification service accessible via VTGate. The purpose of VStream is to provide equivalent information to the MySQL binary logs from the underlying MySQL shards of the Vitess cluster. gRPC clients, including Vitess components like VTTablets, can subscribe to a VStream to receive change events from other shards. The VStream pulls events from one or more VStreamer instances on VTTablet instances, which in turn pulls events from the binary log of the underlying MySQL instance. This allows for efficient execution of functions such as VReplication where a subscriber can indirectly receive events from the

binary logs of one or more MySQL instance shards, and then apply it to a target instance. An user can leverage VStream to obtain in-depth information about data change events for given Vitess keyspace, shard, and position. A single VStream can also consolidate change events from multiple shards in a keyspace, making it a convenient tool to feed a CDC (Change Data Capture) process downstream from your Vitess datastore.

For reference, please refer to the diagram below:

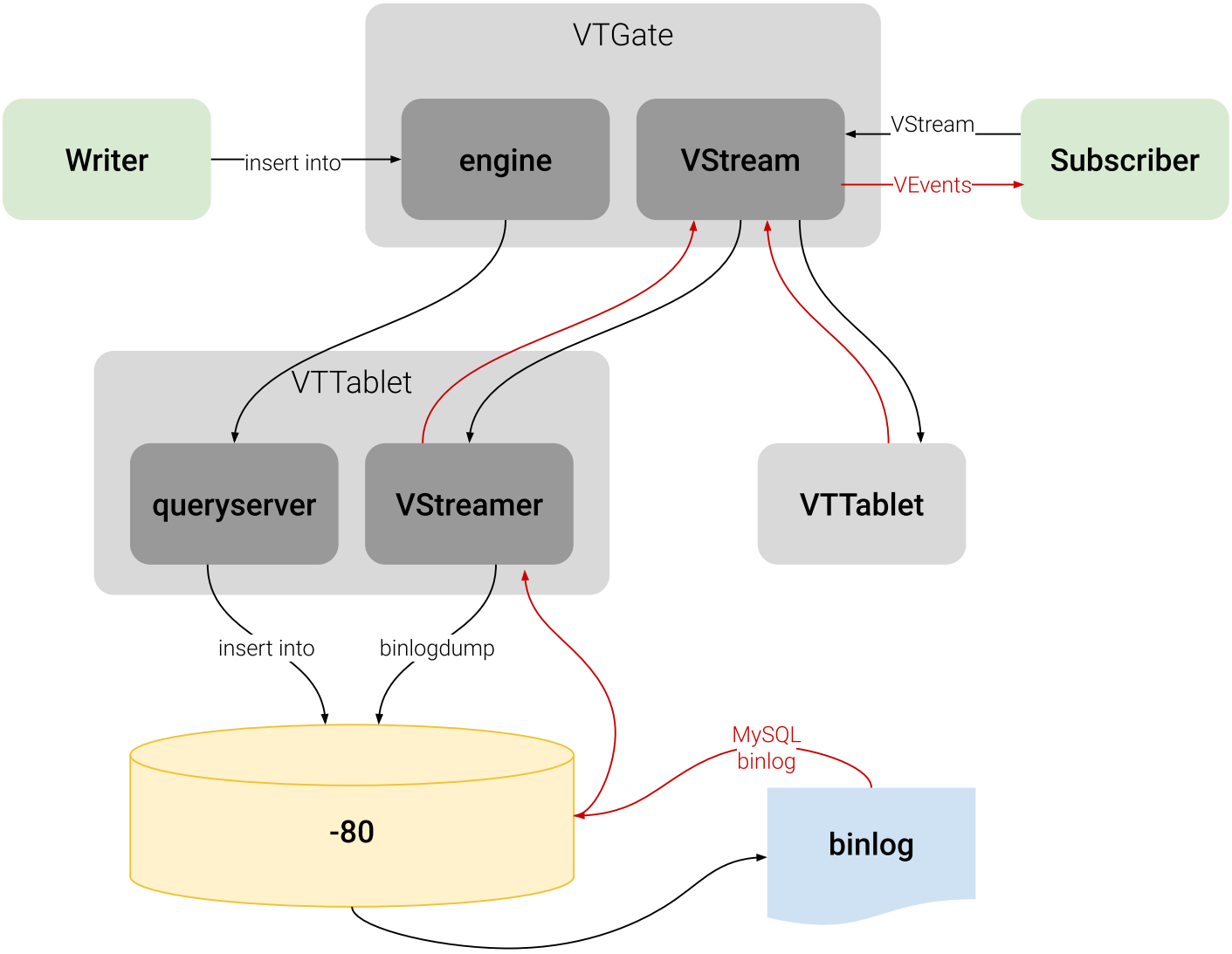


Figure 1: VStream diagram

Note: A VStream is distinct from a VStreamer. The former is located on the VTGate and the latter is located on the VTTablet.

vtctl

vtctl is a command-line tool used to administer a Vitess cluster. It is available as both a standalone tool (`vtctl`) and client-server (`vtctlclient` in combination with `vtctld`). Using client-server is recommended, as it provides an additional layer of security when using the client remotely.

Using vtctl, you can identify master and replica databases, create tables, initiate failovers, perform resharding operations, and so forth.

As vtctl performs operations, the Topology Service is updated as needed. Other Vitess servers observe those changes and react accordingly. For example, if you use vtctl to fail over to a new master database, vtgate sees the change and directs future write operations to the new master.— ## vtctld

vtctld is an HTTP server that lets you browse the information stored in the Topology Service. It is useful for troubleshooting or for getting a high-level overview of the servers and their current states.

`vtctld` also acts as the server for `vtctlclient` connections.— ## VTGate

VTGate is a lightweight proxy server that routes traffic to the correct VTTablet servers and returns consolidated results back to the client. It speaks both the MySQL Protocol and the Vitess gRPC protocol. Thus, your applications can connect to VTGate as if it is a MySQL Server.

When routing queries to the appropriate VTTablet servers, VTGate considers the sharding scheme, required latency and the availability of tables and their underlying MySQL instances.

Related Vitess Documentation

- Execution Plans

Contribute

description: Get involved with Vitess development

You want to contribute to Vitess? That's awesome!

In the past we have reviewed and accepted many external contributions. Examples are the Java JDBC driver, the PHP PDO driver or VTGate v3 improvements.

We're looking forward to any contribution! Before you start larger contributions, make sure to reach out first and discuss your plans with us.

This page describes for new contributors how to make yourself familiar with Vitess and the programming language Go.

Learning Go

Vitess was one of the early adaptors of Google's programming language Go. We love it for its simplicity (e.g. compared to C++ or Java) and performance (e.g. compared to Python).

Contributing to our server code will require you to learn Go. We recommend that you follow the Go Tour to get started.

The Go Programming Language Specification is also useful as a reference guide.

Learning Vitess

Before diving into the Vitess codebase, make yourself familiar with the system and run it yourself:

- Read the What is Vitess page, in particular the architecture section.
- Read the Concepts and Sharding pages.
  - We also recommend to look at our latest presentations. They contain many illustrations which help understanding how Vitess works in detail.
  - After studying the pages, try to answer the following question (click expand to see the answer):
    Let's assume a keyspace with 256 range-based shards: What is the name of the first, the second and the last shard?
    -01, 01-02, ff-
- Go through the Kubernetes and local get started guides.
  - While going through the tutorial, look back at the architecture and match the processes you start in Kubernetes with the boxes in the diagram.

Build on CentOS

description: Instructions for building Vitess on your machine for testing and development purposes

{{< info >}} If you run into issues or have questions, we recommend posting in our Slack channel, click the Slack icon in the top right to join. This is a very active community forum and a great place to interact with other users. {{< /info >}}

The following has been verified to work on **CentOS 7**. If you are new to Vitess, it is recommended to start with the local install guide instead.

Install Dependencies

Install Go 1.16+ Download and install Golang 1.16. For example, at writing:

```
curl -LO https://golang.org/dl/go1.16.5.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.16.5.linux-amd64.tar.gz
```

Make sure to add go to your bashrc:

```
# Additions to ~/.bashrc file

# Add go PATH
export PATH=$PATH:/usr/local/go/bin

# Add GOROOT
export GOROOT=/usr/local/go/

# Add GOPATH
export GOPATH=/home/<user>/go
```

Packages from CentOS repos The MariaDB version included with CentOS 7 (5.5) is not supported by Vitess. First install the MySQL 5.7 repository from Oracle:

```
sudo yum localinstall -y
   https://dev.mysql.com/get/mysql57-community-release-el7-9.noarch.rpm
sudo yum install -y mysql-community-server
```

Install additional dependencies required to build and run Vitess:

```
sudo yum install -y make unzip g++ etcd curl git wget
```

Notes:

- We will be using etcd as the topology service. The command `make tools` can also install Zookeeper or Consul for you, which requires additional dependencies.
- Vitess currently has some additional tests written in Python, but we will be skipping this step for simplicity.

Disable SELinux SELinux will not allow Vitess to launch MySQL in any data directory by default. You will need to disable it:

```
sudo setenforce 0
```

Build Vitess

Navigate to the directory where you want to download the Vitess source code and clone the Vitess GitHub repo:

```
cd ~
git clone https://github.com/vitessio/vitess.git
cd vitess
```

Set environment variables that Vitess will require. It is recommended to put these in your `.bashrc`:

```
# Additions to ~/.bashrc file

#VTDATAROOT
export VTDATAROOT=/tmp/vtdataroot

# Vitess binaries
export PATH=~/vitess/bin:${PATH}
```

Build Vitess:

```
make build
```

Testing your Binaries

The unit tests require the following additional packages:

```
sudo yum install -y ant maven zip gcc
```

You can then install additional components from `make tools`. If your machine requires a proxy to access the Internet, you will need to set the usual environment variables (e.g. `http_proxy`, `https_proxy`, `no_proxy`) first:

```
make tools
make unit_test
```

In addition to running tests, you can try running the local example.

Common Build Issues

Key Already Exists This error is because etcd was not cleaned up from the previous run of the example. You can manually fix this by running `./401_teardown.sh`, removing vtdataroot and then starting again:

```
Error:  105: Key already exists (/vitess/zone1) [6]
Error:  105: Key already exists (/vitess/global) [6]
```

MySQL Fails to Initialize This error is most likely the result of SELinux enabled:

```
1027 18:28:23.462926   19486 mysqld.go:734] mysqld --initialize-insecure failed:
   /usr/sbin/mysqld: exit status 1, output: mysqld: [ERROR] Failed to open required
   defaults file: /home/morgo/vitess/vtdataroot/vt_0000000102/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!

could not stat mysql error log (/home/morgo/vitess/vtdataroot/vt_0000000102/error.log):
   stat /home/morgo/vitess/vtdataroot/vt_0000000102/error.log: no such file or directory
E1027 18:28:23.464117   19486 mysqlctl.go:254] failed init mysql: /usr/sbin/mysqld: exit
   status 1, output: mysqld: [ERROR] Failed to open required defaults file:
   /home/morgo/vitess/vtdataroot/vt_0000000102/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!
```

```
E1027 18:28:23.464780   19483 mysqld.go:734] mysqld --initialize-insecure failed:
   /usr/sbin/mysqld: exit status 1, output: mysqld: [ERROR] Failed to open required
   defaults file: /home/morgo/vitess/vtdataroot/vt_0000000101/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!
```

Build on macOS

description: Instructions for building Vitess on your machine for testing and development purposes

{{< info >}} If you run into issues or have questions, we recommend posting in our Slack channel, click the Slack icon in the top right to join. This is a very active community forum and a great place to interact with other users. {{< /info >}}

The following has been verified to work on **macOS Mojave**. If you are new to Vitess, it is recommended to start with the local install guide instead.

Install Dependencies

Install Xcode Install Xcode.

Install Homebrew and Dependencies Install Homebrew. From here you should be able to install:

```
brew install go@1.16 automake git curl wget mysql@5.7
```

Add `mysql@5.7` and `go@1.16` to your `PATH`:

```
echo 'export PATH="/usr/local/opt/mysql@5.7/bin:$PATH"' >> ~/.bash_profile
echo 'export PATH="/usr/local/opt/go@1.16/bin:$PATH"' >> ~/.bash_profile
```

Do not install etcd via brew otherwise it will not be the version that is supported. Let it be installed when running make build.

Do not set up MySQL or etcd to restart at login.

Build Vitess

Navigate to the directory where you want to download the Vitess source code and clone the Vitess GitHub repo:

```
cd ~
git clone https://github.com/vitessio/vitess.git
cd vitess
```

Set environment variables that Vitess will require. It is recommended to put these in your `~/.bash_profile` file:

```
# Vitess binaries
export PATH=~/vitess/bin:${PATH}
```

Build Vitess:

```
make build
```

Testing your Binaries

The unit tests require that you first install a Java runtime. This is required for running ZooKeeper tests:

```
brew tap adoptopenjdk/openjdk
brew install adoptopenjdk8
brew info java
```

You will also need to install `ant` and `maven`:

```
brew install ant maven
```

You can then install additional components from `make tools`. If your machine requires a proxy to access the Internet, you will need to set the usual environment variables (e.g. `http_proxy`, `https_proxy`, `no_proxy`) first:

```
make tools
make unit_test
```

In addition to running tests, you can try running the local example.

Common Build Issues

Key Already Exists This error is because etcd was not cleaned up from the previous run of the example. You can manually fix this by running `./401_teardown.sh`, removing vtdataroot and then starting again:

```
Error:  105: Key already exists (/vitess/zone1) [6]
Error:  105: Key already exists (/vitess/global) [6]
```

/tmp/mysql.sock Already In Use This error occurs because mysql is serving on the same port that vttgate requires. To solve this issue stop mysql service. If you have installed mysql via brew as specified above you should run:

```
brew services stop mysql@5.7
```

Build on Ubuntu/Debian

description: Instructions for building Vitess on your machine for testing and development purposes

{{< info >}} If you run into issues or have questions, we recommend posting in our Slack channel, click the Slack icon in the top right to join. This is a very active community forum and a great place to interact with other users. {{< /info >}}

The following has been verified to work on **Ubuntu 19.10** and **Debian 10**. If you are new to Vitess, it is recommended to start with the local install guide instead.

Install Dependencies

Install Go 1.16+ Download and install Golang 1.16. For example, at writing:

```
curl -LO https://dl.google.com/go/go1.16.5.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.16.5.linux-amd64.tar.gz
```

Make sure to add go to your bashrc:

```
# Additions to ~/.bashrc file

# Add go PATH
export PATH=$PATH:/usr/local/go/bin

# Add GOROOT
export GOROOT=/usr/local/go/

# Add GOPATH
export GOPATH=/home/<user>/go
```

Packages from apt repos Install dependencies required to build and run Vitess:

```
# Ubuntu
sudo apt-get install -y mysql-server mysql-client make unzip g++ etcd curl git wget

# Debian
sudo apt-get install -y default-mysql-server default-mysql-client make unzip g++ etcd curl
   wget
```

The services `mysqld` and `etcd` should be shutdown, since `etcd` will conflict with the `etcd` started in the examples, and `mysqlctl` will start its own copies of `mysqld`:

```
sudo service mysql stop
sudo service etcd stop
sudo systemctl disable mysql
sudo systemctl disable etcd
```

Notes:

- We will be using etcd as the topology service. The command `make tools` can also install Zookeeper or Consul for you, which requires additional dependencies.
- Vitess currently has some additional tests written in Python, but we will be skipping this step for simplicity.

Disable mysqld AppArmor Profile The `mysqld` AppArmor profile will not allow Vitess to launch MySQL in any data directory by default. You will need to disable it:

```
sudo ln -s /etc/apparmor.d/usr.sbin.mysqld /etc/apparmor.d/disable/
sudo apparmor_parser -R /etc/apparmor.d/usr.sbin.mysqld
```

The following command should return an empty result:

```
sudo aa-status | grep mysqld
```

Build Vitess

Navigate to the directory where you want to download the Vitess source code and clone the Vitess GitHub repo:

```
cd ~
git clone https://github.com/vitessio/vitess.git
cd vitess
```

Set environment variables that Vitess will require. It is recommended to put these in your `.bashrc`:

```
# Additions to ~/.bashrc file

#VTDATAROOT
export VTDATAROOT=/tmp/vtdataroot

# Vitess binaries
export PATH=~/vitess/bin:${PATH}
```

Build Vitess:

```
make build
```

Testing your Binaries

The unit test requires that you first install the following packages:

```
sudo apt-get install -y ant maven default-jdk zip
```

You can then install additional components from `make tools`. If your machine requires a proxy to access the Internet, you will need to set the usual environment variables (e.g. `http_proxy`, `https_proxy`, `no_proxy`) first:

```
make tools
make unit_test
```

In addition to running tests, you can try running the local example.

Common Build Issues

Key Already Exists This error is because etcd was not cleaned up from the previous run of the example. You can manually fix this by running `./401_teardown.sh`, removing vtdataroot and then starting again:

```
Error:  105: Key already exists (/vitess/zone1) [6]
Error:  105: Key already exists (/vitess/global) [6]
```

MySQL Fails to Initialize This error is most likely the result of an AppArmor enforcing profile being present:

```
1027 18:28:23.462926   19486 mysqld.go:734] mysqld --initialize-insecure failed:
   /usr/sbin/mysqld: exit status 1, output: mysqld: [ERROR] Failed to open required
   defaults file: /home/morgo/vitess/vtdataroot/vt_0000000102/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!

could not stat mysql error log (/home/morgo/vitess/vtdataroot/vt_0000000102/error.log):
   stat /home/morgo/vitess/vtdataroot/vt_0000000102/error.log: no such file or directory
E1027 18:28:23.464117   19486 mysqlctl.go:254] failed init mysql: /usr/sbin/mysqld: exit
   status 1, output: mysqld: [ERROR] Failed to open required defaults file:
   /home/morgo/vitess/vtdataroot/vt_0000000102/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!
E1027 18:28:23.464780   19483 mysqld.go:734] mysqld --initialize-insecure failed:
   /usr/sbin/mysqld: exit status 1, output: mysqld: [ERROR] Failed to open required
   defaults file: /home/morgo/vitess/vtdataroot/vt_0000000101/my.cnf
mysqld: [ERROR] Fatal error in defaults handling. Program aborted!
```

The following command disables the AppArmor profile for `mysqld`:

```
sudo ln -s /etc/apparmor.d/usr.sbin.mysqld /etc/apparmor.d/disable/
sudo apparmor_parser -R /etc/apparmor.d/usr.sbin.mysqld
```

The following command should now return an empty result:

```
sudo aa-status | grep mysqld
```

If this doesn't work, you can try making sure all lurking processes are shutdown, and then restart the example again in the `/tmp` directory:

```
for process in `pgrep -f '(vtdataroot|VTDATAROOT)'`; do
 kill -9 $process
done;

export VTDATAROOT=/tmp/vtdataroot
./101_initial_cluster.sh
```

Coding Standards

Backwards Compatibility

Vitess is being used to power many mission-critical production workloads at very large scale. Moreover, many users deploy directly from the main branch. It is very important the changes made by contributors do not break any existing workloads.

In order to avoid disruption, the following concerns need to be kept in mind:

- Does the change affect any external APIs? If so, make sure the change satisfies the compatibility rules.
- Can the change introduce a performance regression? If so, it will be good to measure the impact using benchmarks.
- If the change is substantial or is a breaking change, you must publish the proposal as an issue with a title like `RFC: Changing behavior of feature xxx`. Following this, sufficient time has to be given for others to give feedback. A breaking change must still satisfy the compatibility rules.
- New features that affect existing behavior must be introduced "behind a flag". Users will then be encouraged to enable them, but will have the option to fallback to the old behavior if issues are found.

What does a good PR look like?

Every GitHub pull request must go through a code review and get approved before it will be merged into the main branch.

Every pull request should meet the following requirements:

- Use the Pull Request Template
- Adhere to the Go coding guidelines and watch out for these common errors.
- Contain a description message that is as detailed as possible. Here is a great example https://github.com/vitessio/vitess/pull/6543.
- Pass all CI tests that run on PRs.
- For bigger changes, it is a good idea to start by creating an RFC (Request for Commnent) issue - this is where you can discuss the feature and why it's important. Once that is in place, you can create the PR, as a solution to the problem described in the issue. Separating the need and the solution this way makes discussions easier and more focused.

Testing We use unit tests both to test the code and to describe it for other developers.

- Unit tests should:
  - Demonstrate every use case the change covers.
  - Involve all important units being added or changed.
  - Attempt to cover every corner case the change introduces. The thumb rule is: if it can happen in production, it must be covered.
- Integration tests should ensure that the feature works end-to-end. They must cover all the important use cases of the feature.
- A separate pull request into `vitessio/website` that updates the documentation is required if the feature changes or adds to existing behavior.

Bug fixes If you are creating a PR to fix a bug, make sure to create an end-to-end test that fails without your change. This is the important reproduction case that will make sure this particular bug does not show up again, and that clearly shows on your PR what bug you are fixing.

While you are fixing the bug, it's valuable if you take the time to step back and try to think of other places where this problem could have impacted. It's often possible to infer that other similar problems in this or other parts of the code base can be prevented.

Some additional points to keep in mind:

- Does this change match an existing design / bug?

- Is this change going to log too much? (Error logs should only happen when the component is in bad shape, not because of bad transient state or bad user queries)
- Does this match our current patterns? Example include RPC patterns, Retries / Waits / Timeouts patterns using Context, …

We also recommend that every author look over their code change before committing and ensure that the recommendations below are being followed. This can be done by skimming through `git diff --cached` just before committing.

- Scan the diffs as if you're the reviewer.
  - Look for files that shouldn't be checked in (temporary/generated files).
  - Look for temporary code/comments you added while debugging.
  - Example: `fmt.Println("AAAAAAAAAAAAAAAAAA")`
  - Look for inconsistencies in indentation.
  - Use 2 spaces in everything except Go.
  - In Go, just use goimports.
- Commit message format:
  - ```
    <component>: This is a short description of the change.

    If necessary, more sentences follow e.g. to explain the intent of the change, how it fits
       into the bigger picture or which implications it has (e.g. other parts in the system
       have to be adapted.)

    Sometimes this message can also contain more material for reference e.g. benchmark numbers
       to justify why the change was implemented in this way.
    ```
- Comments
  - `// Prefer complete sentences when possible.`
  - Leave a space after the comment marker `//`.

If your reviewer leaves comments, make sure that you address them and then click "Resolve conversation". There should be zero unresolved discussions when the PR merges.

Assigning a Pull Request

Vitess uses code owners to auto-assign reviewers to a particular PR. If you have been granted membership to the Vitess team, you can add additional reviewers using the right-hand side pull request menu.

During discussions, you can also refer to somebody using the *@username* syntax and they'll receive an email as well.

If you want to receive notifications even when you aren't mentioned, you can go to the repository page and click *Watch*.

Approving a Pull Request

As a reviewer you can approve a pull request through two ways:

- Approve the pull request via GitHub's code review system
- Reply with a comment that contains *LGTM* (Looks Good To Me)

Merging a Pull Request

The Vitess team will merge your pull request after the PR has been approved and CI tests have passed.
    ```

GitHub Workflow

If you are new to Git and GitHub, we recommend to read this page. Otherwise, you may skip it.

Our GitHub workflow is a so called triangular workflow:

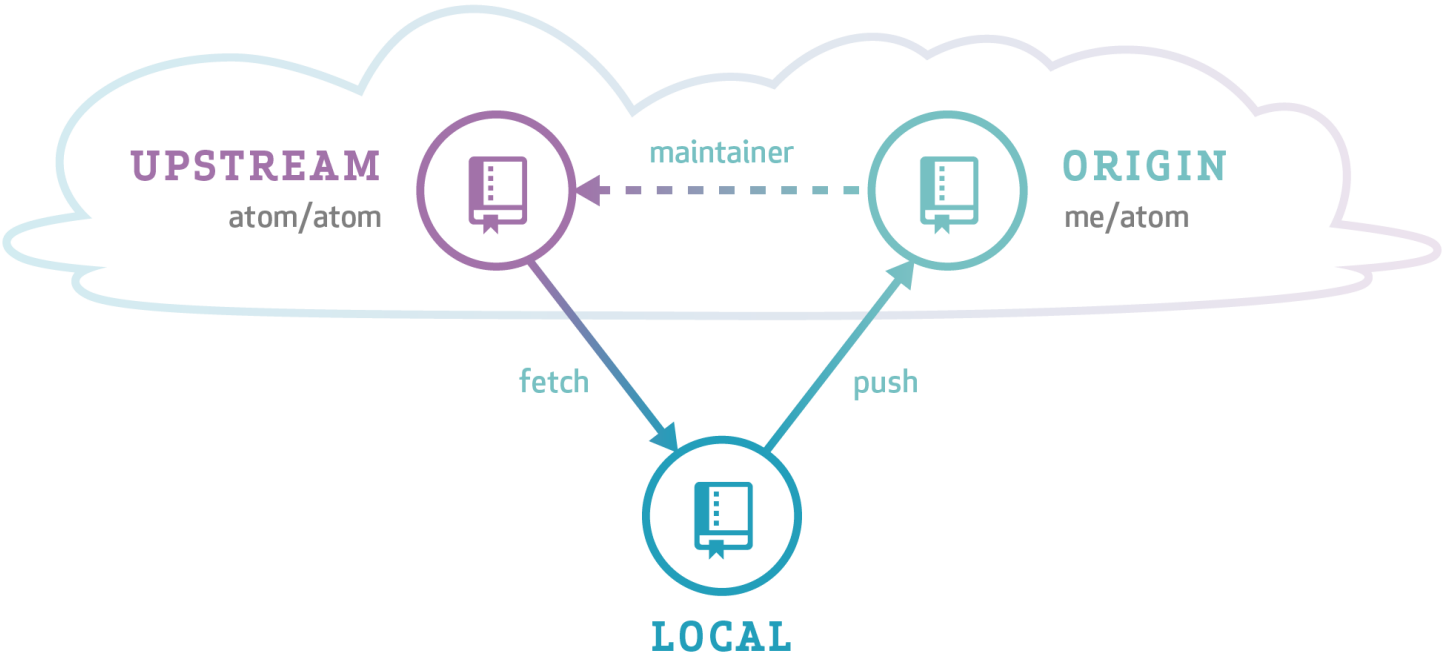


Image Source

The Vitess code is hosted on GitHub. This repository is called *upstream*. You develop and commit your changes in a clone of our upstream repository (shown as *local* in the image above). Then you push your changes to your forked repository (*origin*) and send us a pull request. Eventually, we will merge your pull request back into the *upstream* repository.

Remotes

Since you should have cloned the repository from your fork, the `origin` remote should look like this:

```
$ git remote -v
origin  git@github.com:<yourname>/vitess.git (fetch)
origin  git@github.com:<yourname>/vitess.git (push)
```

To help you keep your fork in sync with the main repo, add an `upstream` remote:

```
$ git remote add upstream git@github.com:vitessio/vitess.git
$ git remote -v
origin  git@github.com:<yourname>/vitess.git (fetch)
origin  git@github.com:<yourname>/vitess.git (push)
upstream        git@github.com:vitessio/vitess.git (fetch)
upstream        git@github.com:vitessio/vitess.git (push)
```

Now to sync your local `main` branch, do this:

```
$ git checkout main
(main) $ git pull upstream main
```

Note: In the example output above we prefixed the prompt with `(main)` to stress the fact that the command must be run from the branch `main`.

You can omit the `upstream main` from the `git pull` command when you let your `main` branch always track the main `vitessio/vitess` repository. To achieve this, run this command once:

```
(main) $ git branch --set-upstream-to=upstream/main
```

Now the following command syncs your local `main` branch as well:

```
(main) $ git pull
```

Topic Branches

Before you start working on changes, create a topic branch:

```
$ git checkout main
(main) $ git pull
(main) $ git checkout -b new-feature
(new-feature) $ # You are now in the new-feature branch.
```

Try to commit small pieces along the way as you finish them, with an explanation of the changes in the commit message. Please see the Code Review page for more guidance.

As you work in a package, you can run just the unit tests for that package by running `go test` from within that package.

When you're ready to test the whole system, run the full test suite with `make test` from the root of the Git tree. If you haven't installed all dependencies for `make test`, you can rely on the Travis CI test results as well. These results will be linked on your pull request.

Committing your work

When running `git commit` use the `-s` option to add a Signed-off-by line. This is needed for the Developer Certificate of Origin.

Sending Pull Requests

Push your branch to the repository (and set it to track with `-u`):

```
(new-feature) $ git push -u origin new-feature
```

You can omit `origin` and `-u new-feature` parameters from the `git push` command with the following two Git configuration changes:

```
$ git config remote.pushdefault origin
$ git config push.default current
```

The first setting saves you from typing `origin` every time. And with the second setting, Git assumes that the remote branch on the GitHub side will have the same name as your local branch.

After this change, you can run `git push` without arguments:

```
(new-feature) $ git push
```

Then go to the repository page and it should prompt you to create a Pull Request from a branch you recently pushed. You can also choose a branch manually.

Addressing Changes

If you need to make changes in response to the reviewer's comments, just make another commit on your branch and then push it again:

```
$ git checkout new-feature
(new-feature) $ git commit
(new-feature) $ git push
```

That is because a pull request always mirrors all commits from your topic branch which are not in the `main` branch.

Once your pull request is merged:

- close the GitHub issue (if it wasn't automatically closed)
- delete your local topic branch (`git branch -d new-feature`)

Submitting Issues

If you have a significant change to add, you need to create an issue prior to creating a Pull Request. This issue should be used to explain what you're planning to work on, to track progress, and design decisions.

Or if you'd like to report a bug you've found within Vitess you can also create an issue.

Design Docs

description: Collection of design docs for some sections of Vitess

Benchmark

description: Design docs related to the benchmarks of Vitess

arewefastyet

description: Nightly Benchmarking project for Vitess

Background

With the codebase of Vitess becoming larger and complex changes getting merged, we need to ensure our changes are not degrading the performance of Vitess.

Benchmarking Tool

To solve the aforementioned issue, we use a tool named arewefastyet that automatically tests the performance of Vitess. The performance are measured through a set of benchmarks divided into two categories: `micro` and `macro`, the former focuses on unit-level functions, and the latter targets system-wide performance changes. Those benchmarks are run every night if there are new commits, release or pull request needing benchmarks.

The GitHub repository where lies all of arewefastyet's code can be found on vitessio/arewefastyet.

Pull Request needing benchmarks When a pull request affect the perfomance of Vitess, one might wish to benchmark it before merging it. The latter can be done by setting the `Benchmark me` label to your pull request. Each night, at midnight central european time, the head commit of your pull request will be benchmarked and compared against the pull request's base.

Website

The performances of Vitess can be observed throughout different releases, git SHAs, and nightly builds on arewefastyet's website at https://benchmark.vitess.io.

The website lets us:

- See previous nightly benchmarks.
- Search results for a specific git SHA.
- Compare two results for two git SHAs.
- See micro and macro benchmarks results throughout different releases.
- Compare performance between VTGate's v3 planner and Gen4 planner.

Build And Process

description: Design docs related to the build framework and our development process

PR Naming Conventions

description: A naming convention for GitHub pull requests

Background

There's a clear need of improving GitHub naming conventions for the following sections.

- Pull Request Naming
- Branch Naming
- Commit Message Naming
- Tag Naming
- Label Addition

Feature

The most problematic area we see is the Pull Request Naming Convention and Labels hence we'd like to come up with guidelines and once agreed by maintainers provide a Template that will help streamline the above areas.

For Issue Templates please refer to this section.

Solution

Naming Convention The suggested solution would be creating general guidelines for this naming convention update.

- Short and descriptive summary
- End with corresponding ticket/story id (e.g., GitHub issue, etc.)
- Should be capitalized and written in imperative present tense
- Not end with a period

Consists of three parts:

- Title: Short informative summary of the pull request
- #[Issue_ID]
- Description: More detailed explanatory text describing the PR for the reviewer

Suggested Format: Description #[Ticket_ID] Example: `Fix where clause in information schema with correct database name #6599`

Description:

- Separated with a blank line from the subject
- Explain what, why, etc.
- Max 72 chars
- Each paragraph capitalized
- Example and/or Reproduce steps

```
Overview of the Issue
The query sent down from VTGate to Vttablet does not replace the where clause of the
    information schema queries for table_schema = 'keyspace' to table_schema = 'databasename'

Reproduction Steps
Steps to reproduce this issue, for example:

Deploy the following vschema:

{
  "sharded": false,
  "tables": {
  }
}
```

Labels Two class of labels have been created and 1 label from each class must be added to all the pull requests. The two classes and the corresponding labels are as follows :-

- **Component**
  - Build/CI
  - Cluster Management
  - Java
  - Observability
  - Query Serving
  - VReplication
  - VTAdmin
  - vttestserver
- **Type**
  - Announcement
  - Bug
  - CI/Build
  - Documentation
  - Enhancement
  - Feature Request
  - Internal Cleanup
  - Performance
  - Question
  - RFC
  - Testing

Additionally, other labels may be added as appropriate.

Other Suggestions

- How to write good pull requests via GitHub templates link

Call for feedback

We're looking for the community's feedback on the above suggestions/flow. Thank you for taking the time to read and respond!

Query Serving

description: Query Serving related design docs

Consistent lookup vindexes

description:

Problem

Vitess supports the concept of a lookup vindex, or what is also commonly known as a cross-shard index. It's implemented as a mysql table that maps a column value to the keyspace id. This is usually needed when someone needs to efficiently find a row using a where clause that does not contain the primary vindex.

This lookup table can be sharded or unsharded. No matter which option one chooses, the lookup row is most likely not going to be in the same shard as the keyspace id it points to.

Vitess allows the transparent population of these rows by assigning an owner table, which is the main table that requires this lookup. When a row is inserted into the main table, the lookup row for it is created in the lookup table. The lookup row is also deleted on a delete of the main row. These essentially result in distributed transactions, which require 2PC to guarantee atomicity.

There have been requests to see if we can avoid 2PC and find a solution that could still give the same guarantees. This proposal is for such a solution.

Existing solution

The best existing solution is one where the lookup row is created immediately (autocommited) irrespective of whether the original transaction completes or not. This can generally work if values are not reused by the original table. For example, it's a perfect use case where the column is an auto-increment. In the worst case, the lookup row will have a dangling reference. A select on such a value will result in an extra lookup on the target shard, but will result in no rows being found, as expected.

However, if we want to allow for values to be reused, then the presence of this row will prevent a new row from being created. We cannot confidently overwrite the row because it may be referencing an actual row in the main table.

At the same time, we also cannot allow deletes. This has to do with the order of operations: a delete is currently performed before the main row is deleted. In autocommit mode, this delete will be committed first. If the original transaction is later rolled back, then we have data that is not referenceable from the lookup, which is a more serious problem.

The above use case assumes that we're dealing with a unique lookup. Similar concerns exist for non-unique lookups also.

Approach

The proposed approach is to extend the existing behavior to better handle the case of a reuse. When we find that a lookup row already exists, we don't know if it's dangling or not. But it's possible to find out: all we have to do is look for the owner row. If it doesn't exist, then we can conclude that it's dangling and allow for it to be overwritten. Otherwise, we can return a duplicate key error.

Implementation

Even though the approach looks simple, the implementation gets tricky because of race conditions. For example, if we autocommitted the lookup row. A new insert can immediately find the row, but it could appear as dangling because the main row for the original transaction might not have been created yet. Even if it was created, it might not have been committed.

Similar race conditions exist with various combinations including update and delete operations.

To prevent these race conditions, we have to go through an elaborate sequence of operations, transactions, and row locking as explained below:

Extend the VTGate transaction model The VTGate transaction currently has a Session variable that keeps track of the current transactions it has open with the vttablets. To solve the above problem correctly, we need more fine-grained control over the order in which those transactions are committed. Additionally, even if some operations end up on the same shards, they may have to be in different transactions. To satisfy the needs that a consistent lookup needs, we need to introduce two independent transactions: they would ideally be names as prequel and sequel. However, since `sequel` can be confusing in this context, we'll call them PreTransaction and PostTransaction. A PreTransaction is always committed first and a PostTransaction is always committed last.

We can now dive into the details of how each vindex operation will utilize the above new features:

Create The Create will start with a `BeginPreTransaction` (if one is not already started), and insert the lookup row. If it succeeds, we return success.

Why do we need a separate transaction instead of autocommiting: If a new insert comes in, it will find this row, but won't find the owner because it has yet to be created. If we tried to lock that row it will succeed with zero rows returned, and make us think that it's ok for us to update it. If we were in a transaction, the new insert will wait till the PreTransaction is committed, at which time the owner row would have already been created.

If the insert results in a duplicate key error, then it issues a `select for update` for the lookup row as well as the owner. The locking of the owner row is performed using the main transaction (not pre or post). If this select returns a row, then we return a duplicate key error.

Why do we need to lock the owner row: if a previous operation just committed the PreTransaction and is yet to commit the main transaction, then a simple select of the row will result in a row not found, which will make the current operation think that the row doesn't exist.

Why do we need to also lock the lookup row: two inserts could race and try to lock the owner row, and they will both succeed, which will cause them to both try to update the lookup row. This also means that the owner row must be updated using values obtained from the select for update.

If the row is not found, the existing row is updated to point at the new keyspace id and return success. If this operation fails with a dup key error, we return that error also.

In the above sequence when the combined transaction is committed, it's safe to commit the PreTransaction first. The correct behavior will be observed irrespective of whether the main transaction succeeds or not.

It's important to note that the most common code path where the insert succeeds is as efficient as it was before. Things are less efficient in the case where a row already exists, but it's the less common case.

We can also write a watchdog process later that harvests dangling rows.

Delete The Delete operation will start with a `BeginPostTransaction`. The delete of lookup rows must happen after the main transaction that's deleting the rows is committed. If the main transaction commits and the PostTransaction fails, we'll just be in a dangling row situation that we already know how to handle.

The delete operation already selects the owned rows for an update before issuing the delete. So, there are no other corner cases to handle for the delete. Even otherwise, row locks will eventually be obtained when the actual delete is performed.

Update The update operation is currently implemented as a delete and insert. This means that an update that changes a vindex column will result in three transactions. However, it's a small price to pay for the consistency gained.

Special case: the case where the update does not change the column must be detected and be converted to a no-op. Otherwise, we'll end up in an indefinite lock wait because we'll be trying to change the same row via two different transactions (pre and post).

Extending the Vindex API

Apart from the pre and post-transaction changes, we'll need to extend the vindex API: A Vindex will export an optional `SetOwnerColumns` function. If present, the vschema builder will call this function at the time of creation and supply the column names of the owner table. This will allow the `Create` to issue the `select for update` to lock the owner row.

New vindexes

In order to remain backward compatible. We'll implement these new vindexes under different names, likely `consistent_lookup` and `consistent_lookup_unique`.

Corner cases to keep in mind

Non-unique lookups: the above algorithm will work equally well for non-unique lookup. The only change will be the definition of a duplicate key. In the unique case, the `from` rows are the primary key, and in the non-unique case, the entire row is the pk.

Multi-column vindexes: These will also work as long as the primary keys are correctly defined. Within a transaction, multiple different rows could be inserted that have owned lookups. This means that the pre and post transactions are full sessions that may themselves open multiple vttablet transactions (ShardSessions).

Commit order within a session: Previously there was an implicit commit order that was dictated by the order in which DMLs were issues. This was instituted as a best-effort attempt to cause the lookup rows to be committed first. Although not guaranteed, this would have been the most common commit order. In the new scheme, since we explicitly control the commit order, there's no need to follow this implicit order. Instead, we can commit the shard sessions in parallel.

Bulk insert: Vitess supports bulk inserts. This results in a corresponding bulk insert for lookup rows also. If the lookup insert results in a duplicate key error, the algorithm will have to fall back to individual inserts in order to implement the above algorithm for rows that yield dup keys. For simplicity, we may not do bulk insert in the initial implementation.

What won't work

Autocommit of the main transaction will not work. For the lookup to be consistent, we have to start two transactions, create the rows in both transactions and commit in a specific order. If the lookup transaction was created with the row inserted without a commit, and if the main row is autocommitted, then the commit of the lookup row can later fail, which will result queries returning incorrect results.

The feature should be used only with 2PC disabled. Of course, you don't need this feature if you're using 2PC already.

Locking functions

description:

Locking Functions

Supporting the advisory locking functions in MySQL is important, given that they are used by common applications and frameworks.

Functions covered

- **GET_LOCK()**
- **IS_FREE_LOCK()**
- **IS_USED_LOCK()**
- **RELEASE_ALL_LOCKS()**
- **RELEASE_LOCK()**

Restrictions

Vitess will initially only support locking functions with these limitations:

- Can only be used in SELECT queries
- The queries can either have only the table `dual`, or have no `FROM` clause.

Functionality

Locking function evaluation will have a simple and consistent routing scheme, making sure all requests happen at the same target. This way, locks will be executed on the same `mysqld`. The locking function evaluation always is routed to the first shard in the first keyspace known to the VTGate, sorted alphabetically.

Using any of the locking functions will force the session to use a reserved connection - a dedicated connection to `mysqld` for that session, so that `get_lock()/release_lock()` happen on the same connection, and so that `COM_QUIT` releases any lingering locks.

Examples of valid queries

```
SELECT GET_LOCK('lock1',10);
SELECT RELEASE_LOCK('lock1');
SELECT GET_LOCK('lock1',10), GET_LOCK('lock2',10);
SELECT RELEASE_ALL_LOCKS()
SELECT GET_LOCK(@customVariable, 10);
```

Examples of queries not supported in the first implementation

```
SELECT GET_LOCK(user_name,10) FROM users;
INSERT INTO T (id) VALUES (GET_LOCK('lock2',10));
DO GET_LOCK('lock1',10);
```

Owned Primary Vindexes

description:

This proposal presents a way to allow for owned lookup vindexes to also be the Primary Vindex.

Problem statement

An owned lookup vindex is a vindex for which a mapping is created from the input column value(s) to a keyspace id at the time of insert. This necessarily means that the keyspace id has to be known before this mapping can be created.

This also means that an owned vindex cannot be the primary vindex. This is because the primary vindex is the one that determines the keyspace id of a row based on the input value. The essential thinking is: if one can compute the keyspace id for a row, then there is no need to store a mapping for it.

There are, however, situations where these seemingly contradictory situations can co-exist. Here are two use cases:

1. I want to generate a random keyspace id at the time of row insertion, but I want to remember it for later.
2. I want to use multiple column values to compute a keyspace id, but would like to save a mapping from just one column to the computed keyspace id. This way, the lookup can be used in situations where only the mapped column was provided in the where clause. This is an upcoming use case for those who wish to geo-partition their data based on regions.

In the above cases, the vindex is capable of generating a keyspace id, and at the same time, it needs to save that data so that it can be used later when `Map` is called.

Solution

In order to support the new use cases, the following changes can be made:

1. Extend the Vindex API where a vindex can export a `MapNew` function. This function will generate a keyspace ID.
2. Allow owned lookup vindexes that support the `MapNew`function to be the primary vindex.
3. If a `MapNew` function exists for the primary vindex, the insert will call it. Otherwise, it will use the regular `Map` function.
4. If the vindex is owned, then the regular code for an owned vindex is executed, even if it's the primary vindex. This is because `MapNew` would have returned us the keyspace id for the row.

Why not a `MapAndCreate` function instead? A `MapAndCreate` will end up duplicating the work performed by an owned Vindex for the Create part. Having `MapNew` only generate the keyspace id keeps the functionality more orthogonal and composable.

Why do we need a separate `MapNew` function instead of just reusing the `Map` function? This is to address the first use case. In the first use case, the `MapNew` function will generate a random keyspace id, whereas the `Map` function will perform the lookup.

Resharding

We currently don't support resharding through a lookup vindex. This is because a vttablet is not able to read from a lookup table that may be distributed across different keyspaces and shards. Also, performing a lookup for each vreplication row may be a performance bottleneck.

For the first use case of random keyspace id generation, there is no recourse; Looking up the keyspace id may be the only way to reshard. However, this is currently only a hypothetical use case. So, there's no need to solve this problem immediately.

For the second use case, which addresses the geo-partitioning problem, `Map` and `MapNew` have the same implementation. The only difference is that `MapNew` takes multiple columns as input, whereas `Map` takes only one column.

We have always wanted to extend `Map` to accept multiple columns as input. We now have the opportunity to do so. In such cases, a resharding will be able to use `Map` with all column values as input, thereby avoiding the need to read from lookup tables.

The VTGate itself can continue to just send the first column's value for the vindex. However, VReplication can use all column values to allow for `Map` to return efficiently.

Replica Transactions

description:

Feature Description Vitess currently supports transactions through vtgate only on MASTER tablets. We would like to extend transaction support to REPLICA (or other tablet types).

Use Case(s)

- Consistent reads
- Sqoop integration

Proposed Solution

- When vtgate chooses a tablet to execute a query on, it should return the tablet alias.
- tablet alias and transactionID will be stored on the shard session struct.
- if the session object has a tablet alias set, then the query will target the specific tablet.
- if the transaction is committed or rolled back, the session should end.

In order to use this feature, a client will need to issue `use @replica` followed by `BEGIN`.

Prerequisite: #5750

Reserved Connections

description:

Feature Description

Vitess uses connection pooling to minimize the memory usage of the underlying MySQL servers. This means that different users connecting to a vtgate can be sharing a connection session to MySQL. To make this as invisible as possible to users, Vitess works hard removing all query constructs that would normally need to change the state in the MySQL connection. A simple example are user defined variables. When a user sets or evaluates an UDV, the vtgate will rewrite the query so that it doesn't actually do anything with user variables, and keep the state on the Vitess layer.

For some things a user might want to do, this is not enough, and in those cases, Vitess will use something called reserved connections. This means a dedicated connection from the vttablet to the MySQL server. Reserved connections are used when changing system variables, using temporary tables, or when a user uses the locking functions to acquire advisory locks.

System variables and reserved connections If a user does change a system variable, the user connection will be marked as needing reserved connections, and for all sub-sequent calls to Vitess, connection pooling is turned off for this particular session. This only applies to some system settings. See more details here. Any queries to a tablet from this session will create a reserved connection on that tablet that is reserved for the user and no one else.

Connection pooling is an important part of what makes Vitess fast, so using constructs that turn it off should only be done in rare circumstances. If you are using an application or library that is issues these kind of `SET` statements, the best way to avoid reserved connections is to make sure the global MySQL settings match the one the application is trying to set. When Vitess discovers that you are changing a system setting to the global value, Vitess just ignores those `SET`s.

Once a session has been marked for reserved connections, it will stay as such until the user disconnects.

Temporary tables and reserved connections Temporary tables exist only in the context of a particular MySQL connection. If a user uses temporary tables, Vitess will mark the session as needing a reserved connection. It will continue to require a reserved connection until the user disconnects - removing the temp table is not enough. More info can be found here.

GET_LOCK() and reserved connections The MySQL locking functions allows users to work with user level locks. Since the locks are tied to the connection, and freeing lock has to be done in the same connection as the lock was acquired, use of these functions will force a connection to become a reserved connection. This connection is also kept alive so it does not time out due to inactivity. More information can be found here..

Shutting down reserved connections Whenever a connection gets transformed into a reserved connection, a fresh, clean connections is returned to the connection pool to replace it. Once the vtgate session that initiated the reserved connections disconnects, all reserved connections between the vttablets and MySQL are terminated.—

Set Statements

description:

Use Case

There are multiple applications/frameworks that tweak session system variables at the start of the connection to configure the system in specific ways. Some do it for all connections it uses, and some do it only for specific workloads, such as import, installation, etc.

To support these use cases, Vitess needs to allow SET statements to be set on a per-connection basis.

The recommendation is to change the setting at a global level during Vitess tablet setup. When this is done, sessions can use the normal connection pools, which is the most efficient way to operate. However, this design allows for sessions with VTgate changing these system variables and Vitess will make sure that they are set for all connections with MySQL for that session.

System variable types

Vitess internally classifies system variables into three types - Ignore, Reserve, and Vitess-aware.

Ignore modification These are the system variables for which Vitess validates the current setting and notify users via warning that settings are either ignored or modification was not allowed. E.g. default_storage_engine, debug, etc.

Vitess Aware These are the system variables for which Vitess needs to change behavior based on the settings. E.g. autocommit, client_found_rows, etc.

Reserve connection These are the system variables that Vitess will send down to the storage engine and modify the settings at the session connection level. E.g. sql_mode, sql_safe_updates, etc.

This document contains the design for this type of system variables.

Design

VTTablet Reserve Pool VTTablet will have a new pool named reserved pool. Connections in the reserved pool are tied to a specific session and are closed when the session is closed.

QueryService interface changes:

- ReserveExecute: Reserve a connection, execute the set statements, and the query.
- ReserveBeginExecute: Reserve a connection, execute the set statements, move the reserve connection to active transaction pool and execute begin and the query.
- ReserveRelease: This closes the reserved connection and also rollback the transaction if it is in an open state.
- BeginExecute: ReserveId will be added as a parameter to the request.
- Execute: ReserveId will be added as a parameter to the request.

Connections that have updated session system variables will at the end of a transaction (Commit/Rollback) be moved to the reserve pool - they can't be reused by other sessions.

VTGate When a session issues a SET statement to change a system variable, VTGate will compare the provided value with the already configured value. If it differs, it is stored in the session.

A connection with the vttablet is reserved when a query is sent from the client connection. The stored settings are applied first to the reserved connection and then the query is executed. Storing the changed setting in the session enables VTGate to apply settings to any new connection that is opened with different shards.

State	Reserved	Transaction	Recorded SETs
clear	F	F	F
remember-sets	F	F	T

State	Reserved	Transaction	Recorded SETs
reserved	T	F	T
inTx	F	T	F
reserved-inTx	T	T	T

From State	Query	Vtgate Action	API on tablet	To State
clear	SET system var	Record statement	No call	remember-sets
remember-sets	SET system var	Record statement	No call	remember-sets
reserved	SET system var	Record statement	Execute	reserved
inTx	SET system var	Record statement	ReserveExecute	reserved-inTx
reserved-inTx	SET system var	Record statement	Execute	reserved-inTx
remember-sets	Query		ReserveExecute	reserved
reserved	Query		Execute	reserved
reserved-inTx	Query		Execute	reserved-inTx
remember-sets	Begin + Query		ReserveBeginExecute	reserved-inTx
reserved	Begin + Query		BeginExecute	reserved-inTx
remember-sets	Commit/Rollback	No-op	No call	remember-sets
reserved	Commit/Rollback	No-op	No call	reserved
reserved-inTx	Commit/Rollback		Commit/Rollback	reserved
remember-sets	Close Connection	Close Session	No call	clear
reserved	Close Connection	Close Session	ReserveRelease	clear
reserved-inTx	Close Connection	Close Session	ReserveRelease	clear

Release Plan

The release plan would be to update vttablet < vtctl < vtgate in this particular order.

Assumption

The order of the SET statements is not maintained. Different Vitess shards might see settings being applied in any order.

Table Lifecycle

description: Table lifecycle related documents

Safe, lazy DROP TABLE

description:

Problem statement

`DROP TABLE` is a risky MySQL operation in production. There seem to be multiple components involved, the major being that if the table has pages in InnoDB's buffer pool (one or many), then the buffer pool is locked for the duration of the `DROP`. The duration of the `DROP` is also related with the time it takes the operating system to delete the `.ibd` file associated with the table (assuming `innodb_file_per_table`).

Noteworthy that the problem is in particular on the `primary` MySQL server; replicas are not affected as much.

Different companies solve `DROP TABLE` in different ways. An interesting discussion is found on `gh-ost`'s repo and on mysql bugs:

- https://github.com/github/gh-ost/issues/307

- https://bugs.mysql.com/bug.php?id=91977

Solutions differ in implementation, but all suggest *waiting* for some time before actually dropping the table. That alone requires management around `DROP TABLE` operations. As explained below, *waiting* enables reverting the operation.

Vitess should automate table drops and make the problem transparent to the user as much as possible. Breakdown of the suggested solution follows.

Illustrating custom DROP workaround steps

We can make the `DROP` management stateful or stateless. Opting for stateless: no meta tables to describe the progress of the `DROP`. The state should be inferred from the tables themselves. Specifically, we will encode hints in the table names.

We wish to manage `DROP` requests. Most managed `DROP` requests will *wait* before destroying data. If the user issued a `DROP TABLE` only to realize the app still expects the table to exist, then we make it possible to revert the operation.

This is done by first issuing a `RENAME TABLE my_table TO something_else`. To the app, it seems like the table is gone; but the user may easily restore it by running the revert query: `RENAME TABLE something_else TO my_table`.

That `something_else` name can be e.g. `_vt_HOLD_2201058f_f266_11ea_bab4_0242c0a8b007_20200910113042`.

At some point we decide that we can destroy the data. The "hold" period can either be determined by vitess or explicitly by the user. e.g. On a successful schema migration completion, Vitess can choose to purge the "old" table right away. At that stage we rename the table to e.g. `_vt_PURGE_63b5db0c_f25c_11ea_bab4_0242c0a8b007_20200911070228`. A table by that name is eligible to have its data purged.

By experience (see `gh-ost` issue above), a safe method to purge data is to slowly remove rows, until the table is empty. Note:

- This operation needs to be throttled (see #6661, #6668, https://github.com/vitessio/website/pull/512)
- We can `SET SQL_LOG_BIN=0` and only purge the table on the `primary`. This reduces binlog size and also does not introduce replication lag. One may argue that lag-based throttling is not needed, but in my experience it's still wise to use, since replication lag can imply load on `primary`, and it's best to not overload the `primary`.
- We issue a `DELETE FROM my_table LIMIT 50` ; 10-100 are normally good chunk sizes. Order of purging does not matter.

It's important to note that the `DELETE` statement actually causes table pages to *load into the buffer pool*, which works against our objective.

Once all rows are purged from a table, we rename it again to e.g. `_vt_DROP_8a797518_f25c_11ea_bab4_0242c0a8b007_20210211234156`. At this time we point out that `20200911234156` is actually a readable timestamp, and stands for `2021-02-11 23:41:56`. That timestamp can tell us when the table was last renamed.

Vitess can then run an actual `DROP TABLE` for `_vt_DROP_...` tables whose timestamp is older than, say, 2 days. As mentioned above, purging the table actually caused the table to load onto the buffer pool, and we need to wait for it to naturally get evicted, before dropping it.

Suggested implementation: table lifecycle aka table garbage collection

The actual implementation will be in `vttablet`. A `primary` vttablet will routinely check for tables that need work:

- A once per hour check is enough; we are going to spend *days* in dropping a tables, so no need to check frequently
- Look for tables called `_vt_(HOLD|PURGE|DROP)_<UUID>_<TIMESTAMP>` (be very strict about the pattern search, because why not)
- If time is right, `RENAME` to the next step (`HOLD` renames to `PURGE`, `PURGE` renames to `DROP`)
- Continuously purge rows from tables in `PURGE` state. We may want to purge only one table at a time, e.g. purge oldest table, until it is empty, immediately rename it to `DROP` state, move on to the next table, begin purging it.
  - Use throttler in between purge chunks

This means a user-initiated `DROP ... TABLE` on a sharded cluster, is handled by each shard independently. The data is purged at different times, and the disk space reclaimed at different time, each shard will do its best, but will likely have somewhat different workloads to make the operations run faster on some, and slower on others.

We need to allow for alternate methods/flows for dropping tables:

- some users just wait and drop
- some need to purge rows
- some operate independently on replicas.

The way to support the above is by introducing user-defined states. The general table lifecycle flow is this, and in this order:

1. Real table (*alive*)
2. `HOLD`
3. `PURGE`
4. `EVAC`
5. `DROP`
6. *gone*

Vitess will transition the table through these states, *in order*. But it will also support skipping some states. Let's first explain the meaning of the states:

- Real table (alive): Table is in use in production.
- `HOLD`: Table is renamed to something like `_vt_HOLD_6ace8bcef73211ea87e9f875a4d24e90_20200915120410`. Vitess will not make changes to the table, will not drop data. The table is put away for safe keeping for X hours/days. If it turns out the app still needs the table, the user can `RENAME` is back to its original name, taking it out of this game.
- `PURGE`: Table renamed to e.g `_vt_PURGE_6ace8bcef73211ea87e9f875a4d24e90_20200916080539`. Vitess purges (or will purge, based on workload and prior engagements) rows from `PURGE` tables. Data is lost and the user may not resurrect the table anymore. Most likely we will settle for a `SQL_LOG_BIN=0`, ie purging will not go through replication. The replicas are not so badly affected by `DROP` statements as a `primary`.
- `EVAC`: Table renamed to e.g. `_vt_EVAC_6ace8bcef73211ea87e9f875a4d24e90_20200918192031`. The table sits still for Y houtrs/days. I'm thinking this period will be pre-defined by vitess. The purpose of this state is to wait a *reasonable* amount of time so that tabe's pages are evacuated from the innodb buffer pool by the natural succession of production IO/memory activity.
- `DROP`: Table renamed to e.g. `_vt_DROP_6ace8bcef73211ea87e9f875a4d24e90_20200921193202`. Vitess will `DROP TABLE` this table *imminently*.

- gone: end of lifecycle

Transitioning and skipping of states

The above lifecycle will be the default, and safest cycle. It is also the longest. Some users will use this sequence. Others have no issue dropping a table with millions of rows, and don't want to pay the IO+time of purging the data.

We introduce a `vttablet` command line flag: `-table_gc_lifecycle="hold,purge,evac,drop"`, that's the default value. `drop` is implicit, if you don't specify it, it's automatically appended. Otherwise, consider the following flows:

- some people just happy to drop tables with no wait, no purge, no nothing. They will set `-table_gc_lifecycle="drop"`
- some people want to keep the table around for a few days, then just go ahead and drop it: `-table_gc_lifecycle="hold,drop"`
- some want to keep the table, then purge it, then immediately drop: `-table_gc_lifecycle="hold,purge,drop"`

This is both user-customizable and simple to implement.

Asking for a safe DROP

Safe `DROP TABLE`s participate in the Online DDL flow. When a user chooses an online `ddl_strategy`, the user's `DROP TABLE` statement implicitly translates to a `RENAME TABLE` statement which sends the table into our lifecycle mechanism, from where it is garbage collected.

VReplication

description: VReplication related design docs skip_sections: true

File:Position based VReplication

description:

Problem Statement

In order to support migration from legacy databases that may not have GTID turned on, there is a need for VReplication to support file:position based tracking.

It is understood that this type of tracking will work only for a single physical mysql source. Replication cannot be automatically resumed from a different source. In the worst case, it is possible to manually figure out the continuation point, and change the VReplication parameters to resume replication from a different source.

Supporting file:position based vreplication will allow for one-time "shoveling" of data from an existing source to a destination. It could also be used to continuously keep the target data up-to-date just like we do for resharding. In this situation, it should also be possible to reverse the replication after a cut-over from legacy to vitess. This will give us the option to rollback a migration should anything go wrong.

Requirements

The VReplication design is entrenched in using the concept of GTID. So, it will be substantial work to change its DNA to recognize file:position based binlog tracking. However, Vitess has the design elements needed to support multiple GTID formats. This means that we can build a low level abstraction layer that encapsulates a file:position as a GTID. There are situations where this abstraction breaks down. We'll have to handle those corner cases.

Forcing file:position based tracking Although not anticipated, it's possible that a user may want to use file:position based tracking even if the source has GTID turned on. This means that file:position based tracking should not be inferred through auto-detection.

An environment variable (like MYSQL_FLAVOR) should not be used because the tracking mode depends on a specific source. Since VReplication can have multiple sources, this should be dictated by the source.

Current position In GTID mode, the current position is guaranteed to be at a transaction boundary. But a file:position based binlog can report a position for every event, including stray ones that are not material for replication. In such cases, the mechanism of getting the current position from the source and asking the target to stop at that position should work no matter what that position is.

Single Source As mentioned above, only a fixed mysql instance will be supported as source.

Design

A prototype work was done by PlanetScale for file:position based tracking. This was developed as an "RDS" flavor. This is because RDS did not support GTIDs when this was developed. With PlanetScale's permission, this work will be leveraged to implement the new proposal. The work will be published as part of the Vitess license. The following high level tasks will need to be performed.

- **Rename rdsFlavor->filePosFlavor**: This rename will more accurately represent the functionality.
- **Flavor as connection parameter**: Since we need to control the flavor on a per-source basis, the best way to achieve this is to extend the `ConnParams` structure to include a `Flavor`. If empty, auto-detection will be used. If specified, the name will map to a registered flavor implementation. This approach will retain backward compatibility.
- **Track sub-flavor**: Since we want to support file:position based tracking even if GTID is turned on, we need the ability to recognize GTID events. This means that we have to understand MySQL and MariaDB flavors under the covers.
- **Standardize on when to send GTID**: Currently, the binlog dictates when to report a GTID. In some cases, it's before the next transaction, and sometimes it's within. We'll change the design to report the GTID just before the "COMMIT". This is the point where it's actually useful. Knowing exactly when a GTID will be received will simplify the design of the vplayer.
- **Introduce Pseudo-GTID**: In order to report positions outside of transaction boundaries, one possibility is to report them as pseudo-GTIDs. Although it's possible to convert them to fake empty transactions, it may be more readable to use a separate category.
- **Stop Position**: The vplayer currently uses ambiguous rules about how it handles the case where a stop position was exceeded. As part of this change, we'll standardize on: *A stop position is considered to be successfully reached if the new position is greater than or equal to the specified position.* The main motivation for this change is that the possibility of position mismatch is higher in the case of file:pos tracking. We're likely to hit many false positives if we're too strict.

Future improvements

Once a GTID gets more implicitly associated with saveable events, we can later deprecate GTID as an explicit event. Instead, we can send the GTID as part of the COMMIT or DDL message, which is where it's actually used. This will allow us to tighten some code in vplayer. Without this, the association of GTID with a COMMIT is more ambiguous, and there's extra code to glue them together.

Vstream

description: Change notification service accessible via VTGate

References:

Streaming Vitess at Bolt

VStream Skew Minimization

description: Aligning streams from different shards in the VStream API

VStream Skew Detection

Motivation When the VStream API is streaming from multiple shards we have multiple sources of events: one primary or replica tablet for each shard in the provided VGTID. The rate at which the events will be streamed from the underlying sources can vary depending on various factors, like:

- the replication lag on the source tablets (if a replica is selected as the source for the VStream)
- the cpu load on the source tablet
- possible network partitions or network delays

This can result in the events in the VStream from some shards being well ahead of other shards. So, for example, if a row moves from the faster shard to a slower shard we might see the delete event in the VStream from the faster shard long before the insert from the second. This would result in the row going "invisible" for the duration of the skew. This can affect user experience in applications where the VStream events are used to refresh UI, for example.

For most applications where VStream API events feed into change data capture systems for auditing or reporting purposes these delays may be acceptable. However for applications which are using these events for user-facing functions this can cause unexpected behavior. See https://github.com/vitessio/vitess/issues/7402 for one such case.

Goal It is not practically possible to provide exact ordering of events across Vitess shards. The VStream API will inherently stream events from one shard independently of another. However, VStreamer events do keep track of the binlog event timestamps which we can use to loosely coordinate the streams. Since binlog timestamp granularity is only to the nearest second, we attempt to align the streams to within a second.

Implementation The skew minimization feature adds a flag that the client can set. This flag enables skew detection between the various streams. Once a skew is detected, events for streams that are ahead are held back until the lagging streams catch up causing the skew to reach an acceptable level.

Each VStreamer event (*vevent*) contains two timestamps: one when the database transaction occurred, and the other, the current time on the source tablet where the vevent was created. This lets us compute how far in the past the event we just received was created. We use this to determine which shard has the most recent event and which one has the oldest event. Note that, for shards where there are no activity, VStreamer sends a heartbeat event every second. The transaction time for an heartbeat is the same as the current time on the source. (These heartbeats are not forwarded to the VStream since they are synthetic vreplication events.)

If the difference between the fastest and slowest streams is greater than a threshold, we declare that we have detected a skew. MySQL binlogs store the transaction timestamp in seconds. Also, on the VTGate serving the VStream, we adjust this time for clock skews between the VTGate and the source MySQL server. When the user sets the `MinimizeSkew` flag we want to keep the events across shards to be in the same second: each transaction timestamp is within 1 second of each other. To account for rounding-off of the transaction timestamp and the clock-skew we set the threshold to be 2 seconds, instead of 1 second, so that we don't keep stalling the streams due to cumulative round-offs.

Possible unexpected behavior If there are no events for a second in a shard then a heartbeat is sent. On receiving a heartbeat we reset the skew. This is necessary to avoid shards with no events starving other shards. The current logic will align streams only if they are all getting events faster than the heartbeat frequency.

This means that we cannot guarantee the skew alignment feature will work as expected in certain conditions. This could happen mainly while streaming from replicas with high replication lags, say, due to high write qps or a network partition.

Thus it is recommended that you stream from primaries when using this feature. Note, however, that even primaries with skewed loads could trigger such a situation.

API This is how you would turn on the skew detection and alignment feature in a VStream client:

```
import vtgatepb "vitess.io/vitess/go/vt/proto/vtgate"
...
...
flags := &vtgatepb.VStreamFlags{};
flags.MinimizeSkew = true;

reader, err := conn.VStream(ctx, topodatapb.TabletType_MASTER, vgtid, filter, flags)
```

VStream API and Resharding

description: How VStream API handles a reshard

Stream migration on a resharding operation

While subscribing to the VStream API you need to specify the shards from which to stream events. While streaming it is possible that the underlying keyspace is resharded. Thus some or all of the shards which were specified may be replaced by new shards after the resharding is completed.

Stream migration logic within VReplication handles this transparently within VTGate. Event sending will be put on hold momentarily during the actual cutover (when writes are switched) and you will start getting the events (and vgtids) for the new set of shards once the cutover is completed.

An illustration Here is a sample session using the scripts from the local example.

Run the steps up to and including `205_clean_commerce.sh`. Now start a vstream api client in a separate terminal to stream events from the customer table in the customer keyspace, which is currently unsharded.

```
{
  ShardGtids: []*binlogdatapb.ShardGtid{
            {
            Keyspace: "customer",
            Shard:    "0",

          },
  },
}
```

Initial events will be streamed:

```
[type:BEGIN  type:FIELD field_event:<table_name:"customer.customer"
   fields:<name:"customer_id" type:INT64 table:"customer" org_table:"customer"
   database:"vt_customer" org_name:"customer_id" column_length:20 charset:63 flags:49667 >
   fields:<name:"email" type:VARBINARY table:"customer" org_table:"customer"
   database:"vt_customer" org_name:"email" column_length:128 charset:63 flags:128 > > ]
[type:VGTID vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-45" > > ]
[type:ROW row_event:<table_name:"customer.customer" row_changes:<after:<lengths:1
   lengths:16 values:"1alice@domain.com" > > >  type:ROW
   row_event:<table_name:"customer.customer" row_changes:<after:<lengths:1 lengths:14
   values:"2bob@domain.com" > > >  type:ROW row_event:<table_name:"customer.customer"
   row_changes:<after:<lengths:1 lengths:18 values:"3charlie@domain.com" > > >  type:ROW
   row_event:<table_name:"customer.customer" row_changes:<after:<lengths:1 lengths:14
   values:"4dan@domain.com" > > >  type:ROW row_event:<table_name:"customer.customer"
   row_changes:<after:<lengths:1 lengths:14 values:"5eve@domain.com" > > >  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-45"
   table_p_ks:<table_name:"customer" lastpk:<rows:<lengths:1 values:"5" > > > > >
   type:COMMIT ]
[type:BEGIN  type:VGTID vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-45" > >  type:COMMIT ]
```

Now run the resharding scripts and switch reads (301, 302, 303, 304). The following events are now seen:

```
[type:VGTID vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-46" > >  type:DDL
   timestamp:1616748652 statement:"alter table customer change customer_id customer_id
   bigint not null" current_time:1616748652480051077 ]
[type:VGTID vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-47" > >  type:OTHER
   timestamp:1616748652 current_time:1616748652553883482 ]
```

Run the 305 script to switch writes. You will see that vgtids will include the new shards `-80` and `80-` instead of `0`

```
[type:BEGIN timestamp:1616748733 current_time:1616748733480901644  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-48" > >  type:COMMIT
   timestamp:1616748733 current_time:1616748733480932466 ]
[type:BEGIN timestamp:1616748733 current_time:1616748733486715446  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"0"
   gtid:"MySQL56/060a409d-8e10-11eb-9bb5-04ed332e05c2:1-49" > >  type:COMMIT
   timestamp:1616748733 current_time:1616748733486749728 ]

[type:BEGIN timestamp:1616748733 current_time:1616748733519198641  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"80-"
   gtid:"MySQL56/6a60d315-8e10-11eb-b894-04ed332e05c2:1-76" >
   shard_gtids:<keyspace:"customer" shard:"-80"
   gtid:"MySQL56/629442b7-8e10-11eb-a0bb-04ed332e05c2:1-75" > >  type:COMMIT
   timestamp:1616748733 current_time:1616748733519244822 ]
[type:BEGIN timestamp:1616748733 current_time:1616748733520355854  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"80-"
   gtid:"MySQL56/6a60d315-8e10-11eb-b894-04ed332e05c2:1-76" >
   shard_gtids:<keyspace:"customer" shard:"-80"
   gtid:"MySQL56/629442b7-8e10-11eb-a0bb-04ed332e05c2:1-76" > >  type:COMMIT
   timestamp:1616748733 current_time:1616748733520403210 ]
```

Insert new rows: this will result in row events from the new shards. Shards will only stream changes from the point of resharding.

```
$ mysql -u root --host=127.0.0.1 -P 15306 -e "insert into customer(customer_id, email)
   values(6,'sougou@planetscale.com'), (7, 'deepthi@planetscale.com');"
```

```
[type:BEGIN timestamp:1616749631 current_time:1616749631516372189  type:FIELD
   timestamp:1616749631 field_event:<table_name:"customer.customer"
   fields:<name:"customer_id" type:INT64 table:"customer" org_table:"customer"
   database:"vt_customer" org_name:"customer_id" column_length:20 charset:63 flags:53251 >
   fields:<name:"email" type:VARBINARY table:"customer" org_table:"customer"
   database:"vt_customer" org_name:"email" column_length:128 charset:63 flags:128 > >
   current_time:1616749631517765487  type:ROW timestamp:1616749631
   row_event:<table_name:"customer.customer" row_changes:<after:<lengths:1 lengths:22
   values:"6sougou@planetscale.com" > > row_changes:<after:<lengths:1 lengths:23
   values:"7deepthi@planetscale.com" > > > current_time:1616749631517779353  type:VGTID
   vgtid:<shard_gtids:<keyspace:"customer" shard:"80-"
   gtid:"MySQL56/6a60d315-8e10-11eb-b894-04ed332e05c2:1-77" >
   shard_gtids:<keyspace:"customer" shard:"-80"
   gtid:"MySQL56/629442b7-8e10-11eb-a0bb-04ed332e05c2:1-76" > >  type:COMMIT
   timestamp:1616749631 current_time:1616749631517789376 ]
```

Schema Tracker

description: Tracking schema changes in Vstreams

Tracking schema changes in Vstreams

Motivation

Currently, Vstreams work with a single (the latest) database schema. On every DDL the schema engine reloads the schema from the database engine.

All Vstreams on a tablet share a common engine. Vstreams that are lagging might be seeing a newer (and hence incorrect) version of the schema in case ddls were applied in between.

In addition reloading schemas is an expensive operation. If there are multiple Vstreams each of them will separately receive a DDL event resulting in multiple reloads for the same DDL.

{{< info >}} For full functionality, schema tracking relies on non-default Vitess vttablet options: `-watch_replication_stream` and `-track_schema_versions`. Specifically, performing a Vstream from a non-master tablet while concurrently making DDL changes to the keyspace without one or both of these tablet options will result in incorrect Vstream results. {{< /info >}}

Goals

1. Provide a mechanism for maintaining versions of the schema
2. Reduce the number of redundant schema loads

Model

We add a new schema_version table in _vt with columns, including, the gtid position, the schema as of that position, and the ddl that led to this schema. Inserting into this table generates a Version event in Vstream.

Actors

Schema Engine Schema engine gets the schema from the database and only keeps the last (latest) copy it loaded. It notifies subscribers if the schema changes. It polls for the latest schema at intervals or can be explicitly requested to load the schema.

Replication watcher Replication watcher is a Vstream that is started by the tabletserver. It notifies subscribers when it encounters a DDL

Version Tracker Version tracker runs on the master. It subscribes to the replication watcher and inserts a new row into the schema_version table with the latest schema.

Version Historian Version historian runs on both master and replica and handles DDL events. For a given GTID it looks up its cache to check if it has a schema valid for that GTID. If not, on the replica, it looks up the schema_version table. If no schema is found then it provides the latest schema which is updated by subscribing to the schema engine's change notification.

Notes

- Schema Engine is an existing service
- Replication Watcher already exists and is used as an optional Vstream that the user can run. It doesn't do anything specific: it is used for the side-effect that a Vstream loads the schema on a DDL, to proactively load the latest schema.

Basic Flow for version tracking

Master

Version tracker:

1. When the master comes up the replication watcher (a Vstream) is started from the current GTID position. Tracker subscribes to the watcher.
2. Say, a DDL is applied
3. The watcher Vstream sees the DDL and

1. asks the schema engine to reload the schema, also providing the corresponding gtid position
2. notifies the tracker of a schema change

4. Tracker stores its latest schema into the _vt.schema_version table associated with the given GTID and DDL

Historian/Vstreams:

1. Historian warms its cache from the schema_version table when it loads
2. When the tracker inserts the latest schema into _vt.schema_version table, the Vstream converts it into a (new) Version event
3. For every Version event the Vstream registers it with the Historian
4. On the Version event, the tracker loads the new row from the _vt.schema_version table
5. When a Vstream needs a new TableMap it asks the Historian for it along with the corresponding GTID.
6. Historian looks up its cache for a schema version for that GTID. If not present just provides the latest schema it has received from the schema engine.

Replica

1. Version tracker does not run: the tracker can only store versions on the master since it is writing to the database.
2. Historian functionality is identical to that on the master.

Flags

Master Schema version snapshots are stored only on the master. This is done when the Replication Watcher gets a DDL event resulting in a SchemaUpdated(). There are two independent flows here:

1. Replication Watcher is running
2. Schema snapshots are saved to _vt.schema_version when SchemaUpdated is called

Point 2 is performed only when the flag TrackSchemaVersions is enabled. This implies that #1 also has to happen when TrackSchemaVersions is enabled independently of the WatchReplication flag

However if the WatchReplication flag is enabled but TrackSchemaVersions is disabled we still need to run the Replication Watcher since the user has requested it, but we should not store schema versions.

So the logic is:

1. WatchReplication==true
   => Replication Watcher is running
2. TrackSchemaVersions==false
   => SchemaUpdated is a noop
3. TrackSchemaVersions=true
   => Replication Watcher is running
   => SchemaUpdated is handled

The Historian behavior is identical to that of the replica: of course if versions are not stored in _vt.schema_versions it will always provide the latest version of the scheme.

Replica Schema versions are never stored on replicas, so SchemaUpdated is always a Noop. Versions are provided as appropriate by the historian. The historian provides the latest schema if there is no appropriate version.

So the logic is:

1. WatchReplication==true
   => Replication Watcher is running
2. TrackSchemaVersions==false || true //noop
   => Historian tries to get appropriate schema version

Caveat

Only best-effort versioning can be provided due to races between DDLs and DMLs. Some examples below:

Situation 1 If multiple DDLs are applied in a quick sequence we can end up with the following binlog.

T1: DDL 1 on table1

T2: DDL 2 on table1

T3: Version Event DDL1 // gets written because of the time taken by tracker processing DDL1

T4: DML1 on table1

T5: Version Event DDL2 // gets written AFTER DML1

So now on the replica, at T4, the version historian will incorrectly provide the schema from T1 after DDL1 was applied.

Situation 2 If version tracking is turned off on the master for some time, correct versions may not be available to the historian which will always return the latest schema. This might result in an incorrect schema when a Vstream is processing events in the past.

Possible new features around this functionality

- Schema tracking Vstream client for notifications of all ddls
- Raw history of schema changes for auditing, root cause analysis, etc.

VStream Copy

description: Streaming events from the beginning

VStream Copy

Allow vstreams to stream entire databases or tables

Motivation

Currently, the vstream API streams events starting either from the current position of the binlog or from a position specified by the client. The VStream Copy feature adds support to send all events starting from the first position of the binlog.

A naive extension of the current mechanism is to stream from the starting position. However, this is impractical for any database/table of a reasonable size. We will extend VStream to make use of the bulk copy based mechanism similar to vreplication streams, used in MoveTables or Reshard sharding workflows.

Note that with vstream copy the client vstream will not faithfully reproduce the events from the binlog. The aim is to be eventually (and rapidly) consistent with the current database snapshot. This improves performance since we will be merging multiple row updates into a single transaction. Once we have caught up (i.e. the replication lag is small) binlog events will again be directly streamed similar to the current implementation.

Previous API Clients create vstreams by grpc-ing to VTGate using the Vstream API call. In golang:

```
conn, _ :=  VTGate.Dial(ctx, "localhost:15991")
// tabletType is one of replica/master/rdonly, filter,vgtid: see below
reader, _ :=    VStream(ctx, tabletType, vgtid, filter)
e, _  :=    reader.Recv() //receive VEvents in a loop until io.EOF
```

It is possible for network errors to occur or for the client process to fail. In addition, the vstreamer itself might fail at VTGate or VTTablet. Thus, VTGate needs to send state frequently allowing VTGate to be stateless and clients to recover properly from failures.

Also, while creating the stream the client can specify multiple shards and/or keyspaces from which to stream events.

The vgtid structure facilitates both: determining the stream sources and maintaining state. vgtid is a list of tuples: (keyspace, shard, gtid). When a stream is created, gtid can either be "current" or a valid binlog position at which the vstream starts streaming events.

Some examples:

```
// stream from current position from two shards
vgtid := &binlogdatapb.VGtid{
                ShardGtids: []*binlogdatapb.ShardGtid{{
                    Keyspace: "ks",
                    Shard:    "-40",
                    Gtid:     "current",
                },{
                    Keyspace: "ks",
                    Shard:    "80-c0",
                    Gtid:     "current",
                }}
        }

// stream from specific position from all shards in keyspace ks
vgtid := &binlogdatapb.VGtid{
                ShardGtids: []*binlogdatapb.ShardGtid{{
                    Keyspace: "ks",
                    Gtid:     "MariaDB/0-41983-20",
                }}
        }

// stream from current position from all keyspaces
vgtid := &binlogdatapb.VGtid{
                ShardGtids: []*binlogdatapb.ShardGtid{{
                    Gtid:     "current",
                }}
        }
```

The data streamed is sourced from the list of keyspace/shards after applying the specified filter.

To achieve this VTGate sends a vgtid event whenever it encounters a gtid event with the current vgtid state at VTGate. Thus if the stream is broken, for any reason, the client needs to simply create a new vstream using the last vgtid that it received.

Architecture/Design

During a copy there will two distinct phases:

1. Copy phase: where the vstreamer is sending row data in bulk using the primary key to "paginate" the table
2. Replication phase: once copying is completed and going forward we only stream events

The copy phase is nuanced: we copy a batch of rows until a particular PK using a consistent snapshot. However, once the copy is completed the binlog position would have moved possibly containing updates to the rows already transmitted. Hence we need to perform a "catchup" where we play the events up to the current position. We can only send updates to rows that we have already sent to the stream.

After the catchup, we send the next batch of rows and perform the related catchup. This copy-catchup loop continues until all tables are copied, after which it is business as usual and events are streamed as they appear in the binlog.

API Changes for VStream Copy To use VStream Copy you just need to pass an empty string as the position. The only other change is in the vgtid structure. It now becomes a list of

`(keyspace, shard, gtid,[]LastTablePK)`

While the copy is in progress, the LastPK list contains the last seen primary key for each table in that shard. Once the copy is completed and we are replicating the stream this parameter will be nil.

Note that the vgtid is opaque to the consumer of the vstream API once the vstream starts and the ongoing state does not need to be interpreted on the client.

To start a VStream Copy user is expected to provide an empty gtid along with a list of tables to copy (essentially a LastTablePK list with a nil PK for each). Some examples (see https://github.com/vitessio/contrib/blob/master/vstream_client/vstream_client.go for a sample client):

```
// vstream copy two tables table from two shards
filter := &binlogdatapb.Filter{
        Rules: []*binlogdatapb.Rule{{
            Match: "t2",
            Filter: "select id, val from t2",
        },{
            Match: "t1",
            Filter: "select * from t1",
        }},
    }
vgtid := &binlogdatapb.VGtid{
                ShardGtids: []*binlogdatapb.ShardGtid{{
                    Keyspace: "ks",
                    Shard:    "-40",
                    Gtid:     "",
                },{
                    Keyspace: "ks",
                    Shard:    "80-c0",
                    Gtid:     "",
                }}
         }

// stream the entire database: vstream copy from all tables in all keyspaces
filter := &binlogdatapb.Filter{
        Rules: []*binlogdatapb.Rule{{
            Match: "/.*/",
        }},
    }
vgtid := &binlogdatapb.VGtid{
                ShardGtids: []*binlogdatapb.ShardGtid{{
                    Gtid:     "",
                }}
         }
```

VTTablet

description: VTTablet related documents

Componentizing TabletServer

description:

Introduction

As Vitess adoption expands, several feature requests have been popping up that will benefit from multiple instances of TabletServer (or its sub-components) co-existing within the same process.

The following features drive this refactor, in order of priority:

- Multi-schema
- VTShovel: multiple data import sources for vreplication
- VTDirect: allow VTGate to directly send queries to mysql

Beyond these features, componentizing TabletServer will make the vitess architecture more flexible. There are many places in the code where we instantiate a QueryService. All those places can now explore the benefit of instantiating a TabletServer or its sub-components.

Features

This section describes the use cases and their features. An important prerequisite: In order to retain backward compatibility, the new features should not cause existing behavior to be affected.

Multi-schema

There has been a steady inflow of enquiries about use cases where a mysql instance has a large number of schemas (1000+). We currently support multi-schema by requiring the user to launch one vttablet process per schema. This, however, does not scale for the number of schemas we are beginning to see.

To enable this, we need the ability for a single vttablet to host multiple TabletServers. Requirements are:

- Grouped or consolidated Stats (/debug/vars).
- Segregated or consolidated HTTP endpoints, like (/debug/status), with sub-page links working.
- A better way to specify flags: the existing approach of command line flags may not scale.
- A tablet manager that can represent multiple tablet ids.

Other parts of TabletServer have already been modified to point at a shared mysql instance due to work done by @deepthi in #4727, and other related changes.

VTShovel

VTShovel is a data migration feature that extends VReplication to allow a user to specify an external mysql as the source. This can be used to import data and also keep the targets up-to-date as the source is written to.

VTShovel currently has two limitations:

- It supports only one external source per vttablet. The need for multiple sources was voiced as a requirement by one of the adopters.
- It does not support VDiff, which also requires multiple sources.

If the TabletServer refactor is architected correctly, VTShovel should inherit the multi-instance ability without any major impact. In particular:

- Leverage the flags refactor to support more than one external source.
- Observability features implemented for multi-schema like stats and HTTP endpoints should naturally extend to vtshovel.
- VDiff should work.

VTDirect

The excessive use of CPU due to gRPC continues to be a concern among some adopters. Additionally, Vitess is now being deployed against externally managed databases like RDS and CloudSQL. Such users are reluctant to pay the latency cost of the extra hop.

VTDirect is the ability of VTGate to directly send queries to the mysql instances.

This feature adds the following requirements over the previous ones:

- Some features of TabletServer (like sequences) should be disabled or redirected to an actual vttablet.
- TabletServers can have a life-cycle as tablets are added and removed from the topo. The variables and end-points need to reflect these changes.

In previous discussions, alternate approaches that did not require a TabletServer refactor were suggested. Given that the TabletServer refactor brings us much closer to this feature, we'll need to re-evaluate our options for the best approach. This will be a separate RFC.

Requirements

This section describes the requirements dictated by the features.

Stats

Stats (/debug/vars) should be reported in such a way that the variables from each TabletServer can be differentiated. Idiomatic usage of vitess expects the monitoring tool to add the tablet id as a dimension when combining variables coming from different vttablets. Therefore, every TabletServer should be changed to add this dimension to its exported variables.

On the flip side, this may result in an extremely large number of variables to be exported. If so, it may be better to consolidate them. There is no right answer; We have to support both options.

Other options considered We could have each TabletServer export a brand new set of variables by appending the tablet id to the root variable name. However, this would make it very hard for monitoring tools because they are not very good at dealing with dynamic variable names.

HTTP endpoints

A TabletServer exports a variety of http endpoints. In general, it makes sense to have each TabletServer export a separate set of endpoints within the current process. However, in cases where the performance of the underlying mysql is concerned, it may be beneficial to consolidate certain pages.

We'll start with a separate set of pages, with each set prefixed by the tablet id. For example, what was previously `/debug/consolidations` will now become `/cell-100/debug/consolidations`.

Other options considered We could keep the existing set of endpoints unchanged, and have each TabletServer add its section. But this would make it hard to troubleshoot problems related to a specific TabletServer.

The best-case scenario would be the "why not both" option: the original set of pages continue to exist and provide a summary from all the tablet servers. This can still be implemented as an enhancement.

Command line flags

Extending the command line flags to be able to specify parameters for a thousand tablet servers is not going to be practical.

Using config files is a better option. To prevent verbosity, we can use a hierarchy where the root config specifies initial values for the flags, and the TabletServer specific configs can inherit and override the original ones.

The input file format could be yaml. Also, this is a good opportunity for us to come up with better names.

Since the config option is more powerful and flexible, specifying that file in the command line will supersede all legacy flags.

Other options considered These configs could be hosted in the topo. This is actually viable. There are two reasons why this option takes a backseat:

- We currently don't have good tooling for managing data in the topo. VTCtld is currently the only way, and people have found it inadequate sometimes.
- There are mechanisms to secure config files, which will allow it to contain secrets like the mysql passwords. This will not be possible in the case of topos.

Design

We propose to address the above requirements with the following design elements.

Dimension Dropper

The dimension dropper will be a new feature of the stats package. Its purpose is to remove specific dimensions from any multi-dimensional variable.

We'll introduce a new command-line flag that takes a list of labels as input, like '-drop_dimensions='Keyspace,ShardName". The stats package will then remove that dimension from any variable that refers to it.

In the case of the TabletServer, specifying `TabletID` in the list of dropped dimensions will have the effect of all TabletServers incrementing a common counter instead of different ones under their own tablet id.

The reason for this approach is that there are already other use cases where the number of exported variables is excessive. This allows us to address those use cases also.

It's possible that this feature is too broad. For example, one may not want to drop the `Keyspace` dimension from all variables. If we encounter such use cases, it should be relatively easy to extend this feature to accommodate more specific rules.

Exporter

The exporter will be a new feature that will layer between TabletServer and the singleton APIs: stats and http. It will allow you to create exporters that are either anonymous or named.

An anonymous exporter will behave as if you invoked the stats and http directly. Open issue: we'll need to see if we want to protect from panics due to duplicate registrations.

A named exporter will perform consolidations or segregations depending on the situation:

- In the case of a stats variable, it will create a common underlying variable, and will update the dimension that matches the name of the exporter.
- In the case of http, it will export the end point under a new URL rooted from the name of the exporter.

Currently, the connection pools have a different name based mechanism to export different stats. The exporter functionality should support this prefixing, which will eliminate the boiler-plate code in those components.

A prototype of this implementation (aka embedder) is present in the vtdirect branch. This needs to be cleaned up and ported to the latest code.

There is no need for the exporter to provide the option to consolidate without the name because the dimension dropper can cover that functionality.

It's possible to achieve backward compatibility for stats by creating an exporter with a name (tablet id), and then dropping that dimension in stats. However, it'll work only for stats and not for the http endpoints. For this reason, we need to support explicit code paths for the anonymous exporters. Plus, it makes things more explicit.

Config loader

The TabletServer already has most, if not all, of its input flags consolidated into a `Config` struct under tabletenv. The existing flags initialize a `DefaultConfig` global variable. If the command line specifies a newly defined flag, like '-tablet_config='filename.yaml", then we can branch off into code that reads the yaml file and initializes the configs from there.

The code will load the global part of the yaml into a "global" Config. For each tablet specific config, the global config will be copied first, and then the tablet specific overrides will be overwritten into the copied values.

This is an opportunity for us to rename the members of the Config struct to use better names and data types. The yaml tags will have to match these new names.

The most popular yaml reader seems to be https://github.com/go-yaml/yaml. We'll start with that and iterate forward.

The dbconfigs data structure will also be folded into the `Config`. This is because each tablet could potentially have different credentials.

Bonus points Given that vitess uses protos everywhere, we could look at standardizing on a generic way to convert yaml to and from protos. This will allow us to look at converting all formats to yaml. If this sounds viable, we can convert the `Config` struct to be generated from a proto, and then have yaml tags that can convert into it. This will future-proof us in case we decide to go this route.

On the initial search, there is no standard way to do this conversion. It would be nice if protos supported this natively as they do for json. We do have the option of using this code to build our own yaml to proto converter: https://github.com/golang/protobuf/blob/master/jsonpb/jsonpb.go.

TabletManager

The TabletManager change will put everything together for the multi-schema feature.

The ActionAgent data structure will be changed to support multiple tablet servers:

- QueryServiceControl will become a list (or map)
- UpdateStream will be deleted (deprecated)
- TabletAlias, VREngine, _tablet and _blacklistedTables will be added to the QueryServiceControl list

All other members of TabletManager seem unaffected.

The tablet manager API will be extended for cases where requests are specific to a tablet id. For example, `GetSchema` will now require the tablet id as an additional parameter. For legacy support: if tablet id is empty, then we redirect the request to the only tablet.

Note: VREngine's queries are actually tablet agnostic. The user is expected to restrict their queries to the dbname of the tablet. This is not a good user experience. We should tighten up the query analyzer of vrengine to add dbname as an additional constraint or fill in the correct value as needed.

VReplication

VReplication should have a relatively easy change. We already have a field named external mysql. This can be a key into the tablet id Config, which can then be used to pull the mysql credentials needed to connect to the external mysql.

The multi-instance capabilities of VStreamer will naturally extend to support all the observability features we'll add to it.

Fast vttablet state transitions

description: Fast and reliable vttablet state transitions

This issue is in response to #6645. When vttablet transitions from master to non-master, the following problems can occur under different circumstances:

- When a query is killed, it may take a long time for mysql to return the error if it has to do a lot of cleanup. This can delay a vttablet transition. Just closing the pools is not enough because the tablet shutdown also waits for all executing goroutines to return.
- It is possible that the query timeout is much greater than the transaction timeout. In such cases, the query timeout must be reduced to match the transaction timeout. Otherwise, a running query can hold a transaction hostage and prevent a vttablet from transitioning.
- The `transaction_shutdown_grace_period` must acquire a new meaning. It should be renamed to `shutdown_grace_period`, and must also apply to queries that are exceeding this time limit. This limit applies to all queries: streaming, oltp read, reserved, and in_transaction.
- The transaction shutdown code "waits for empty", but reserved connections (not in a transaction) are now part of this pool and will prevent the pool from going empty until they timeout. We need to close them more proactively during a shutdown.
- The transition from master to non-master uses the immediate flag. This should wait for transactions to complete. Fortunately, due to how PRS works, that code path is not used. We instead use the "dont_serve" code path. But this needs to be fixed for future-proofing.

Many approaches were discussed in #6645. Those approaches are all non-viable because they don't address all of the above concerns.

To fix all these problems, some refactoring will need to be done. Here's the proposal:

- The query killer (DBConn.Kill) will be changed to proactively close the connection. This will cause the execution to return immediately, thereby addressing the problem where slow kills delay a shutdown.
- Change the query execution to use the minimum of the transaction timeout and query timeout, but only if the request is part of a transaction.
- Build a list of all currently active queries by extending StreamQueryList. During a shutdown, the state manager will use this list to kill all active queries if shutdown_grace_period is hit. This, along with the Kill change, will cause all those executes to immediately return and the connections will be returned to their respective pools.
- tx_engine/tx_pool: Will now have two modes during shutdown:
  - "kill_reserved" will cause all reserved connections to be closed. If a reserved connection is returned to the pool during this state, it will also be closed. This will be the initial shutdown state until the shutdown grace period is hit.
  - "kill_all" will cause all reserved and transaction connections to be closed. We enter this state after the shutdown grace period is hit. While in this state, the state manager would have killed all currently executing queries. As these connections are returned to the pool, we just close them. This will cause the tx pool to close in a timely manner.
- Fix transition to not be immediate (dead code).

During a previous code review, I also found a (rare) race condition in tx_engine, which I forgot to document. I'll redo the analysis and fix the race if it's trivial.

The most important guarantee of this change is that a shutdown will not take longer than the shutdown_grace_period if it was specified.

VTTablet with RBR mode

description: Updating parameters without restarting a process

Summary

The deprecation of SBR will result in a fundamental change in how vttablet works. The most significant one is that vttablet does not need to compute which rows will be affected by a DML. Dropping this requirement allows us to rethink the implementation of its various features. Some of these changes will not be fully backward compatible. Here are the details:

Pass-through DMLs

Most DMLs can be just passed through.

There is, however, one valuable feature that we want to preserve: the ability to limit the number of rows a DML affects. In SBR mode, this was achieved as a side-effect of the fact that we had to run a "subquery" to identify all affected rows, which allowed us to count them and return an error if they exceeded the limit. We lose this ability in pass-through mode.

Instead of continuing to issue the subquery, the new proposal is to add a LIMIT clause to the DML itself. Then, if "RowsAffected" was greater than the limit, we return an error. The one downside of this implementation is that such a failure will leave the DML partially executed. This means that vttablet will have to force a rollback of the transaction. In most use cases, the application is expected to rollback the transaction on such an error. Therefore, this sounds like a reasonable trade-off for the level of simplicity achieved. There is also the added benefit of avoiding the extra roundtrip for the subquery.

This change has a subtle interaction with the "found rows" flag, which reports affected rows differently. This just means that users of this flag may have to set their limits differently.

The explicit pass-through mode flag `queryserver-config-passthrough-dmls` will continue to behave as is. In the new context, its meaning will change to: "do not limit affected rows".

We believe that inserts of a large number of rows are usually intentional. Therefore, limits will only be applied to updates and deletes.

Autocommits

With most DMLs being pass-through, we can now resurrect autocommit behavior in vttablet. This means that a transactionless DML can be treated as autocommit. In the case where a limit has to be enforced, vttablet can open a transaction, and then rollback if the limit is exceeded.

Sequences

Sequences will continue their existing behavior: normal statements will continue to be sent to mysql. If a 'select next …' is received, the sequence specific functionality will be triggered.

Messages

Messages are undergoing a significant overhaul as seen in #5913. Tracking of row changes has been changed to use VStreamer, which allows for any DML to be executed on those tables. Inserts remain an exception because columns like `time_created` and `time_scheduled` have to be populated by vttablet in order to maintain backward compatibility.

Plan IDs

Most plan ids were specific to SBR. After the refactor, the total number of plans will be greatly reduced, and we'll likely generate a new set of ids.

Schema

These changes greatly reduce our dependence on the schema. We only need to know the following info from a table:

- Field info
- PK columns
- Table comments

VTTablet also gathered table statistics from `information schema`. However, we don't think anyone is using them. So, it may be better to stop reporting them. If anyone still needs them, please speak up now.

Real-time parameter change

description: Updating parameters without restarting a process

Problem Statement

Vitess Components currently disallow changing of configuration parameters while a process is running. There are a few reasons why this was not allowed:

- The command line used to launch the process will cease to be an authoritative source.
- It is difficult to audit any values that were changed in real-time or keep track of history if changes were made multiple times.
- If a process is restarted for any reason, the changes will be reverted.

However, if there is an ongoing high severity incident, it may be beneficial to allow human operators to make temporary changes to a system. This, of course, comes with the caveat that they will eventually revert the changes or make them permanent by deploying the binaries with the new flags.

Proposal

Given that this should be generally discouraged as common practice, these capabilities should not be made openly available. For example, you should not be able to make such changes using SQL statements.

We'll export a `/debug/env` endpoint from vttablet, protected by `ADMIN` access. The URL will display the current values of the various parameters and will allow you to modify and submit any of those.

Values like connection pool sizes, query timeouts, query consolidation settings, etc. will be changeable. We'll iterate on this list as more use cases arise. The initial list will be in an upcoming PR.

Other options considered

SET Statements: These would be issued via VTGate. The problem with this approach is that the commands would be too widely available. Also, VTGate doesn't allow targeting of specific tablets.

vtctld ExecuteFetchAsDba SET statements: This could be made to work. However, this is currently implemented as a pass-through in tablet manager. Significant changes will be needed to parse the statements and make them do other things.

vtcltd alternate command, like `SetTabletEnv`: This could be made to work. However, it is a lot more work than what is proposed, and may not be worth it for a feature that is meant to be used so rarely. We still have the option of implementing this if the need arises in the future.

VTTablet YAML configuration

description: Tabletserver YAML parameters

Background

This issue is an expansion of #5791, and covers the details of how we'll use yaml to implement a hierarchy of parameters for initializing and customizing TabletServer components within a process.

Using the yaml approach, we intend to solve the following problems:

- Multiple tablet servers need to exist within a process.
- Badly named command line parameters: The new yaml specifications will have simpler and easier to remember parameter names.
- Sections for improved readability.
- Unreasonable default values: The new specs will introduce a simplified approach that will save the user from tweaking too many variables without knowing their consequences.
- Repetition: If multiple TabletServers have to share the same parameter values, they should not be repeated.
- Backward compatibility: The system must be backward compatible.

Proposed Approach

We will introduce two new command-line options that will be specified like this:

```
-defaults=defaults.yaml -tablets=tablets.yaml
```

The defaults option allows the user to specify the default settings for all tablets. The "tablets" option allows the user to specify tablet specific options.

We'll provide predefined "defaults" files that people can use as starting points. We'll start with something simple like small.yaml, medium.yaml, and large.yaml.

Protobuf We explored the use of protobufs for converting to and from yaml. However, the protobuf JSON implementation, which was required to correctly parse enums, was unable to preserve the original values for sub-objects. This was a showstopper for making defaults work.

The existing `tabletenv.TabletConfig` data structure will be converted into a struct with the new names and appropriate JSON tags.

Detailed specification

The following section lists all the properties that can be specified in a yaml file along with the existing defaults.

During implementation, we'll need to decide between creating a universal protobuf that represents this structure vs converting the original one into something more usable.

Convention used: same as the one followed by kubernetes, and we'll be making use of https://github.com/kubernetes-sigs/yaml for the implementation.

Note that certain properties (like tabletID) are only applicable to tablet-specific files.

```
tabletID: zone-1234

init:
  dbName:            # init_db_name_override
  keyspace:          # init_keyspace
  shard:             # init_shard
  tabletType:        # init_tablet_type
  timeoutSeconds: 60 # init_timeout

db:
  socket:     # db_socket
```

```
  host:       # db_host
  port: 0     # db_port
  charSet:    # db_charset
  flags: 0    # db_flags
  flavor:     # db_flavor
  sslCa:      # db_ssl_ca
  sslCaPath:  # db_ssl_ca_path
  sslCert:    # db_ssl_cert
  sslKey:     # db_ssl_key
  serverName: # db_server_name
  connectTimeoutMilliseconds: 0 # db_connect_timeout_ms
  app:
    user: vt_app      # db_app_user
    password:         # db_app_password
    useSsl: true      # db_app_use_ssl
    preferSocket: true
  dba:
    user: vt_dba      # db_dba_user
    password:         # db_dba_password
    useSsl: true      # db_dba_use_ssl
    preferSocket: true
  filtered:
    user: vt_filtered # db_filtered_user
    password:         # db_filtered_password
    useSsl: true      # db_filtered_use_ssl
    preferSocket: true
  repl:
    user: vt_repl     # db_repl_user
    password:         # db_repl_password
    useSsl: true      # db_repl_use_ssl
    preferSocket: true
  appdebug:
    user: vt_appdebug # db_appdebug_user
    password:         # db_appdebug_password
    useSsl: true      # db_appdebug_use_ssl
    preferSocket: true
  allprivs:
    user: vt_allprivs # db_allprivs_user
    password:         # db_allprivs_password
    useSsl: true      # db_allprivs_use_ssl
    preferSocket: true

oltpReadPool:
  size: 16                # queryserver-config-pool-size
  timeoutSeconds: 0       # queryserver-config-query-pool-timeout
  idleTimeoutSeconds: 1800 # queryserver-config-idle-timeout
  prefillParallelism: 0   # queryserver-config-pool-prefill-parallelism
  maxWaiters: 50000       # queryserver-config-query-pool-waiter-cap

olapReadPool:
  size: 200               # queryserver-config-stream-pool-size
  timeoutSeconds: 0       # queryserver-config-query-pool-timeout
  idleTimeoutSeconds: 1800 # queryserver-config-idle-timeout
  prefillParallelism: 0   # queryserver-config-stream-pool-prefill-parallelism
  maxWaiters: 0
```

```
txPool:
  size: 20                 # queryserver-config-transaction-cap
  timeoutSeconds: 1        # queryserver-config-txpool-timeout
  idleTimeoutSeconds: 1800 # queryserver-config-idle-timeout
  prefillParallelism: 0    # queryserver-config-transaction-prefill-parallelism
  maxWaiters: 50000        # queryserver-config-txpool-waiter-cap

oltp:
  queryTimeoutSeconds: 30 # queryserver-config-query-timeout
  txTimeoutSeconds: 30    # queryserver-config-transaction-timeout
  maxRows: 10000          # queryserver-config-max-result-size
  warnRows: 0             # queryserver-config-warn-result-size

hotRowProtection:
  mode: disable|dryRun|enable # enable_hot_row_protection, enable_hot_row_protection_dry_run
  # Default value is same as txPool.size.
  maxQueueSize: 20            # hot_row_protection_max_queue_size
  maxGlobalQueueSize: 1000    # hot_row_protection_max_global_queue_size
  maxConcurrency: 5           # hot_row_protection_concurrent_transactions

consolidator: enable|disable|notOnMaster # enable-consolidator, enable-consolidator-replicas
heartbeatIntervalMilliseconds: 0         # heartbeat_enable, heartbeat_interval
shutdownGracePeriodSeconds: 0            # transaction_shutdown_grace_period
passthroughDML: false                    # queryserver-config-passthrough-dmls
streamBufferSize: 32768                  # queryserver-config-stream-buffer-size
queryCacheSize: 5000                     # queryserver-config-query-cache-size
schemaReloadIntervalSeconds: 1800        # queryserver-config-schema-reload-time
watchReplication: false                  # watch_replication_stream
terseErrors: false                       # queryserver-config-terse-errors
messagePostponeParallelism: 4            # queryserver-config-message-postpone-cap
cacheResultFields: true                  # enable-query-plan-field-caching


# The following flags are currently not supported.
# enforce_strict_trans_tables
# queryserver-config-strict-table-acl
# queryserver-config-enable-table-acl-dry-run
# queryserver-config-acl-exempt-acl
# enable-tx-throttler
# tx-throttler-config
# tx-throttler-healthcheck-cells
# enable_transaction_limit
# enable_transaction_limit_dry_run
# transaction_limit_per_user
# transaction_limit_by_username
# transaction_limit_by_principal
# transaction_limit_by_component
# transaction_limit_by_subcomponent
```

There are also other global parameters. VTTablet has a total of 405 flags. We may have to later move some of them into these yamls.

Implementation

We'll use the unified tabletenv.TabletConfig data structure to load defaults as well as tablet-specific values. The following changes will be made:

- TabletConfig will be changed to match the above specifications.
- The existing flags will be changed to update the values in a global TabletConfig.
- In case of type mismatch (like time.Duration vs a "Seconds" variable), an extra step will be performed after parsing to convert the flag variables into TabletConfig members.
- The code will be changed to honor the new TabletConfig members.
- After the flag.Parse step, a copy of the global Config will be used as input to load the defaults file. This means that any command line flags that are not overridden in the yaml files will be preserved. This behavior is needed to support backward compatibility in case we decide to move more flags into the yaml.
- For each tablet entry, we create a copy of the result TabletConfig and read the tablet yaml into those, which will set the tablet-specific values. This will then be used to instantiate a TabletServer.

The exact format of the tablets.yaml file is not fully finalized. Our options are to allow a list of files where each is for a single tablet, or, to require only one file containing a dictionary of tablet-specific overrides.

The tabletmanager model

description:

Background

The current tabletmanager model treats the tablet record as authoritative. The tabletmanager polls the tablet record and reacts to changes there. There are also calls sprayed around the code that invoke "RefreshTablet" or "RefreshState", after they update the tablet record.

This model is not representative of how we currently operate vitess. There is actually no benefit to updating the tablet record and expecting the tablet to refresh itself. In fact, the approach is fragile because we're unnecessarily bringing additional components in our chain of action, thereby increasing the chances of failure.

We should instead change our model to say that the tablet process is the authoritative source of its current state. It publishes it to the tablet record, which is then used for discovery.

Details

While vttablet is up Every flow that needs to change something about a tablet directly issues an rpc request to it. The tablet will immediately execute the request, and will perform a best-effort action to update the tablet record, and will continue to retry until it succeeds.

The main advantage of this approach is that the request will succeed with a single round trip to the tablet. If the request fails, we treat it as a failure. If the request succeeds, but the tablet fails to update its record, we still succeed. The tablet record will eventually be updated.

If vttablet is down or is unreachable If vttablet is unreachable, we fail the operation. This failure mode is no worse than us not being able to update a tablet record.

Essentially, we generally assume that the tablet record may not be in sync with the vttablet. This has always been the case, and our code has to deal with this already.

Refresh State Refresh state will continue to exist as an API, but it's only for refreshing against state changes in the global topo.

Exception 1: Mastership In the case of flows that designate who the master is, the topo is the authority. For such requests, the tablet will first try to update its record, and only then succeed. This is required because of how the new mastership redesign works.

Exception 2: VTTablet startup When a vttablet starts up, it will treat the following info from the topo as authoritative:

- Keyspace
- Shard
- Tablet Type
- DBName

As an additional precaution, if this info does not match the init parameters, we should exit.

We can also consider the following: If the tablet type is master, we can force a sync against the shard record before we confirm ourselves to be the master.

Advantages

The main advantage of this approach is that a vttablet becomes the authoritative owner of a tablet record. This will greatly reduce its complexity because it doesn't have to continuously poll it, and it does not have to deal with possibly unexpected or invalid changes in the tablet record.

Since it can assume that nobody else is modifying the record, the vttablet can freely update the tablet record with its local copy without worrying about synchronizing with the existing info.

This will also simplify many flows because they will all become a single request instead of being two requests (change record and refresh).

Load on topos will be reduced because the tablets don't poll anymore.

Transition

We'll first change all call sites to a single round-trip to the tablets. The tabletmanager will continue to update the tablet record and rely on its existing poller.

This change will mean that the tabletmanager has become the sole owner of its tablet record. At this point, we can switch its behavior to non-polling.

FAQ

description: Frequently Asked Questions about Vitess disable_expand: true

Where can I ask questions about Vitess?

Our most popular channel and the one we recommend for asking questions you may have is our Slack located here.

We have a number of other options that can be used as well listed here.

Please do note that we request that you do not ask individual project members for support. Instead please use these channels where the whole community can help you and benefit from the solutions provided. Thanks!

What are the key slack channels to join?

There are many channels available and we encourage you to join as many or as few as interest you. Some of the most popular channels are listed below:

- #kubernetes
- #monitoring
- #operator
- #orchestrator-integration
- #releases
- #vreplication

How can I contribute a Pull Request to Vitess?

We always enjoy having new contributors to Vitess. Just be sure to read the information here to start.

If you are already familiar with Vitess and you'd like information on how to file a Pull Request or submit an Issue request check out the following links:

- Pull Requests
- Issue

What are good videos to watch to get started learning about Vitess?

We have a number of recorded presentations and videos that can be watched to start learning about Vitess.

- For a curated list please check out a PlanetScale blog post here.

Where can I read additional FAQs?

PlanetScale hosts a knowledge base for Vitess. This additional resource is available here.—

Get Started

description: Deploy Vitess on your favorite platform

Vitess supports binary deployment on the following platforms. See also Build On CentOS, Build on MacOS, or Build on Ubuntu if you are interesting in building your own binary, or contributing to Vitess.

Helm Chart (deprecated)

This tutorial is deprecated. We recommend that you use the Operator instead.

This tutorial demonstrates how Vitess can be used with Minikube to deploy Vitess clusters using Helm.

Prerequisites

Before we get started, let's get a few things out of the way:

1. Install Minikube and start a Minikube engine:

   ```
   minikube start
   ```

2. Install kubectl and ensure it is in your `PATH`. For example, on Linux:

   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s
      https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kube
   ```

3. Install Helm 3:

   ```
   wget https://get.helm.sh/helm-v3.2.1-linux-amd64.tar.gz
   tar -xzf helm-v3.*
   # copy linux-amd64/helm into your path
   ```

4. Install the MySQL client locally. For example, on Ubuntu:

   ```
   apt install mysql-client
   ```

5. Install vtctlclient locally:

   If you are familiar with Go development, the easiest way to do this is: `bash go get vitess.io/vitess/go/cmd/vtctlclient` If not, you can also download the latest Vitess release and extract `vtctlclient` from it.

Start a single keyspace cluster

So you searched keyspace on Google and got a bunch of stuff about NoSQL… what's the deal? It took a few hours, but after diving through the ancient Vitess scrolls you figure out that in the NewSQL world, keyspaces and databases are essentially the same thing when unsharded. Finally, it's time to get started.

Change to the helm example directory:

```
git clone git@github.com:vitessio/vitess.git
cd vitess/examples/helm
```

In this directory, you will see a group of yaml files. The first digit of each file name indicates the phase of example. The next two digits indicate the order in which to execute them. For example, `101_initial_cluster.yaml` is the first file of the first phase. We shall execute that now:

```
helm install vitess ../../helm/vitess -f 101_initial_cluster.yaml
```

You should see output similar to the following:

```
$ helm install vitess ../../helm/vitess -f 101_initial_cluster.yaml

NAME: vitess
LAST DEPLOYED: Tue Apr 14 20:32:18 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Release name: vitess

To access administrative web pages, start a proxy with:
  kubectl proxy --port=8001

Then use the following URLs:

      vtctld: http://localhost:8001/api/v1/namespaces/default/services/vtctld:web/proxy/app/
      vtgate:
         http://localhost:8001/api/v1/namespaces/default/services/vtgate-zone1:web/proxy/
```

Verify cluster You can check the state of your cluster with `kubectl get pods,jobs`. After a few minutes, it should show that all pods are in the status of running:

```
$ kubectl get pods,jobs
NAME                                          READY   STATUS      RESTARTS   AGE
pod/commerce-apply-schema-initial-vlc6k       0/1     Completed   0          2m42s
pod/commerce-apply-vschema-initial-9wb2k      0/1     Completed   0          2m42s
pod/vtctld-58bd955948-pgz7k                   1/1     Running     0          2m43s
pod/vtgate-zone1-c7444bbf6-t5xc6              1/1     Running     3          2m43s
pod/zone1-commerce-0-init-shard-master-gshz9  0/1     Completed   0          2m42s
pod/zone1-commerce-0-replica-0                2/2     Running     0          2m42s
pod/zone1-commerce-0-replica-1                2/2     Running     0          2m42s
pod/zone1-commerce-0-replica-2                2/2     Running     0          2m42s

NAME                                          COMPLETIONS   DURATION   AGE
job.batch/commerce-apply-schema-initial       1/1           94s        2m43s
job.batch/commerce-apply-vschema-initial      1/1           87s        2m43s
job.batch/zone1-commerce-0-init-shard-master  1/1           90s        2m43s
```

Setup Port-forward

For ease-of-use, Vitess provides a script to port-forward from kubernetes to your local machine. This script also recommends setting up aliases for `mysql` and `vtctlclient`:

```
./pf.sh &
sleep 5

alias vtctlclient="vtctlclient -server=localhost:15999"
alias mysql="mysql -h 127.0.0.1 -P 15306"
```

Setting up aliases changes `mysql` to always connect to Vitess for your current session. To revert this, type `unalias mysql && unalias vtctlclient` or close your session.

Connect to your cluster You should now be able to connect to the VTGate Server in your cluster with the MySQL client:

```
~/my-vitess-example> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.7.9-Vitess Percona Server (GPL), Release 29, Revision 11ad961

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW DATABASES;
+-----------+
| Databases |
+-----------+
| commerce  |
+-----------+
1 row in set (0.00 sec)
```

Summary In this example, we deployed a single unsharded keyspace named `commerce`. Unsharded keyspaces have a single shard named `0`. The following schema reflects a common ecommerce scenario that was created by the script:

```
create table product(
  sku varbinary(128),
  description varbinary(128),
  price bigint,
  primary key(sku)
);
create table customer(
  customer_id bigint not null auto_increment,
  email varbinary(128),
  primary key(customer_id)
);
create table corder(
  order_id bigint not null auto_increment,
  customer_id bigint,
  sku varbinary(128),
  price bigint,
```

```
  primary key(order_id)
);
```

The schema has been simplified to include only those fields that are significant to the example:

- The `product` table contains the product information for all of the products.
- The `customer` table has a `customer_id` that has an `auto_increment`. A typical customer table would have a lot more columns, and sometimes additional detail tables.
- The `corder` table (named so because `order` is an SQL reserved word) has an `order_id` auto-increment column. It also has foreign keys into `customer(customer_id)` and `product(sku)`.

Next Steps

You can now proceed with MoveTables.

Or alternatively, if you would like to teardown your example:

```
helm delete vitess
kubectl delete pvc -l "app=vitess"
kubectl delete vitesstoponodes --all
```

Congratulations on completing this exercise!

Local Install via Homebrew

description: Instructions for using Vitess on your macOS machine for testing purposes

This guide covers installing Vitess locally to macOS for testing purposes, from pre-compiled binaries. We will launch multiple copies of `mysqld`, so it is recommended to have greater than 4GB RAM, as well as 20GB of available disk space.

A Homebrew package manager is also available, which requires no dependencies on your local host.

```
/bin/bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Install Vitess with Homebrew

Vitess supports MySQL 5.7+ and MariaDB 10.3+. Homebrew will install latest tagged Vitess Release.

```
$ brew install vitess
Updating Homebrew...
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> Updated Formulae
Updated 19 formulae.
==> Updated Casks
Updated 1 cask.

==> Downloading https://homebrew.bintray.com/bottles/vitess-9.0.0.catalina.bottle.tar.gz
Already downloaded:
   /Users/askdba/Library/Caches/Homebrew/downloads/45991b27589a191910e89a1ce529fcdaa694bb5f36b99f1
==> Pouring vitess-9.0.0.catalina.bottle.tar.gz
/usr/local/Cellar/vitess/9.0.0: 268 files, 528.3MB
```

At this point Vitess binaries installed under default Homebrew install location at /usr/local/share/vitess.

Start a Single Keyspace Cluster

For testing purposes initiate following example;

```
$ cd /usr/local/share/vitess/examples/local/
$ ./101_initial_cluster.sh
add /vitess/global
add /vitess/zone1
add zone1 CellInfo
etcd start done...
Starting vtctld...
Starting MySQL for tablet zone1-0000000100...
Starting vttablet for zone1-0000000100...
HTTP/1.1 200 OK
Date: Wed, 17 Feb 2021 13:09:54 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000101...
Starting vttablet for zone1-0000000101...
HTTP/1.1 200 OK
Date: Wed, 17 Feb 2021 13:10:03 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000102...
Starting vttablet for zone1-0000000102...
HTTP/1.1 200 OK
Date: Wed, 17 Feb 2021 13:10:12 GMT
Content-Type: text/html; charset=utf-8
W0217 16:10:13.075530   65562 main.go:67] W0217 13:10:13.074510 reparent.go:188]
   master-elect tablet zone1-0000000100 is not the shard master, proceeding anyway as
   -force was used
W0217 16:10:13.076472   65562 main.go:67] W0217 13:10:13.075483 reparent.go:194]
   master-elect tablet zone1-0000000100 is not a master in the shard, proceeding anyway as
   -force was used
I0217 16:10:13.076498   65562 main.go:67] I0217 13:10:13.075729 reparent.go:225] resetting
   replication on tablet zone1-0000000102
I0217 16:10:13.076503   65562 main.go:67] I0217 13:10:13.075730 reparent.go:225] resetting
   replication on tablet zone1-0000000101
I0217 16:10:13.076508   65562 main.go:67] I0217 13:10:13.075732 reparent.go:225] resetting
   replication on tablet zone1-0000000100
I0217 16:10:13.649509   65562 main.go:67] I0217 13:10:13.649327 reparent.go:244]
   initializing master on zone1-0000000100
I0217 16:10:13.819282   65562 main.go:67] I0217 13:10:13.818887 reparent.go:277] populating
   reparent journal on new master zone1-0000000100
I0217 16:10:13.819341   65562 main.go:67] I0217 13:10:13.818928 reparent.go:284]
   initializing replica zone1-0000000101
I0217 16:10:13.819363   65562 main.go:67] I0217 13:10:13.819005 reparent.go:284]
   initializing replica zone1-0000000102
I0217 16:10:15.913013   65563 main.go:67] I0217 13:10:15.912595 tablet_executor.go:240]
   Received DDL request. strategy=direct
I0217 16:10:16.020590   65563 main.go:67] I0217 13:10:16.020461 tablet_executor.go:240]
   Received DDL request. strategy=direct
I0217 16:10:16.189170   65563 main.go:67] I0217 13:10:16.189044 tablet_executor.go:240]
   Received DDL request. strategy=direct
New VSchema object:
{
  "tables": {
```

```
    "corder": {

    },
    "customer": {

    },
    "product": {

    }
  }
}
If this is not what you expected, check the input data (as JSON parsing will skip
   unexpected fields).
Waiting for vtgate to be up...
vtgate is up!
Access vtgate at http://askdba:15001/debug/status
```

Verify your initial cluster:

```
$ mysql -e "show vitess_tablets"
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| Cell  | Keyspace | Shard | TabletType | State   | Alias            | Hostname  |
   MasterTermStartTime  |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| zone1 | commerce | 0     | MASTER     | SERVING | zone1-0000000100 | localhost |
   2021-02-17T13:10:13Z |
| zone1 | commerce | 0     | REPLICA    | SERVING | zone1-0000000101 | localhost |
                        |
| zone1 | commerce | 0     | RDONLY     | SERVING | zone1-0000000102 | localhost |
                        |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
```

You can also verify that the processes have started with `pgrep`:

```
$ pgrep -fl vtdataroot | awk '{print $2,$3}'
etcd --enable-v2=true
vtctld -topo_implementation
/bin/sh /usr/local/opt/mysql@5.7/bin/mysqld_safe
/usr/local/opt/mysql@5.7/bin/mysqld
   --defaults-file=/usr/local/Cellar/vitess/9.0.0/share/vitess/examples/local/vtdataroot/vt_000000
vttablet -topo_implementation
/bin/sh /usr/local/opt/mysql@5.7/bin/mysqld_safe
/usr/local/opt/mysql@5.7/bin/mysqld
   --defaults-file=/usr/local/Cellar/vitess/9.0.0/share/vitess/examples/local/vtdataroot/vt_000000
vttablet -topo_implementation
/bin/sh /usr/local/opt/mysql@5.7/bin/mysqld_safe
/usr/local/opt/mysql@5.7/bin/mysqld
   --defaults-file=/usr/local/Cellar/vitess/9.0.0/share/vitess/examples/local/vtdataroot/vt_000000
vttablet -topo_implementation
vtgate -topo_implementation
```

The exact list of processes will vary. For example, you may not see `mysqld_safe` listed.

If you encounter any errors, such as ports already in use, you can kill the processes and start over:

```
pkill -9 -f '(vtdataroot|VTDATAROOT)' # kill Vitess processes
rm -rf /usr/local/Cellar/vitess/9.0.0/share/vitess/examples/local/vtdataroot
```

Setup Aliases

For ease-of-use, Vitess provides aliases for `mysql` and `vtctlclient`:

```
source ./env.sh
```

Setting up aliases changes `mysql` to always connect to Vitess for your current session. To revert this, type `unalias mysql && unalias vtctlclient` or close your session.

Connect to your cluster

You should now be able to connect to the VTGate server that was started in `101_initial_cluster.sh`:

```
/usr/local/share/vitess/examples/local> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.9-Vitess (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+-----------------------+
| Tables_in_vt_commerce |
+-----------------------+
| corder                |
| customer              |
| product               |
+-----------------------+
3 rows in set (0.00 sec)
```

You can also browse to the vtctld console using the following URL:

```
http://localhost:15000
```

Summary

In this example, we deployed a single unsharded keyspace named `commerce`. Unsharded keyspaces have a single shard named `0`. The following schema reflects a common ecommerce scenario that was created by the script:

```
create table product (
  sku varbinary(128),
  description varbinary(128),
  price bigint,
  primary key(sku)
);
create table customer (
  customer_id bigint not null auto_increment,
  email varbinary(128),
  primary key(customer_id)
);
create table corder (
```

```
  order_id bigint not null auto_increment,
  customer_id bigint,
  sku varbinary(128),
  price bigint,
  primary key(order_id)
);
```

The schema has been simplified to include only those fields that are significant to the example:

- The `product` table contains the product information for all of the products.
- The `customer` table has a `customer_id` that has an `auto_increment`. A typical customer table would have a lot more columns, and sometimes additional detail tables.
- The `corder` table (named so because `order` is an SQL reserved word) has an `order_id` auto-increment column. It also has foreign keys into `customer(customer_id)` and `product(sku)`.

Next Steps

You can now proceed with MoveTables.

Or alternatively, if you would like to teardown your example:

```
pkill -9 -f '(vtdataroot|VTDATAROOT)' # kill Vitess processes
rm -rf /usr/local/Cellar/vitess/9.0.0/share/vitess/examples/local/vtdataroot
```

Local Install via Docker

description: Instructions for using Vitess on your machine for testing purposes

This guide illustrates how to run a local testing Vitess setup via Docker. The Vitess environment is identical to the local setup, but without having to install software on one's host other than Docker.

Check out the vitessio/vitess repository

Clone the GitHub repository via:

- SSH: `git clone git@github.com:vitessio/vitess.git`, or:
- HTTP: `git clone https://github.com/vitessio/vitess.git`

```
cd vitess
```

Build the docker image

In your shell, execute:

```
make docker_local
```

This creates a docker image named `vitess/local` (aka `vitess/local:latest`)

Run the docker image

Execute:

```
./docker/local/run.sh
```

This will set up a MySQL replication topology, as well as `etcd`, `vtctld` and `vtgate` services.

- `vtgate` listens on http://127.0.0.1:15001/debug/status
- `vtctld` listens on http://127.0.0.1:15000/debug/status
- Control panel is available at http://localhost:15000/app/

From within the docker shell, aliases are set up for your convenience. Try the following `mysql` commands to connect to various tablets:

- `mysql commerce`
- `mysql commerce@master`
- `mysql commerce@replica`
- `mysql commerce@rdonly`

You will find that Vitess runs a single keyspace, single shard cluster.

Summary

In this example, we deployed a single unsharded keyspace named `commerce`. Unsharded keyspaces have a single shard named `0`. The following schema reflects a common ecommerce scenario that was created by the script:

```
create table product (
  sku varbinary(128),
  description varbinary(128),
  price bigint,
  primary key(sku)
);
create table customer (
  customer_id bigint not null auto_increment,
  email varbinary(128),
  primary key(customer_id)
);
create table corder (
  order_id bigint not null auto_increment,
  customer_id bigint,
  sku varbinary(128),
  price bigint,
  primary key(order_id)
);
```

The schema has been simplified to include only those fields that are significant to the example:

- The `product` table contains the product information for all of the products.
- The `customer` table has a `customer_id` that has an `auto_increment`. A typical customer table would have a lot more columns, and sometimes additional detail tables.
- The `corder` table (named so because `order` is an SQL reserved word) has an `order_id` auto-increment column. It also has foreign keys into `customer(customer_id)` and `product(sku)`.

Next Steps

You can now proceed with MoveTables.

Exiting the docker shell terminates and destroys the vitess cluster.

Local Install

description: Instructions for using Vitess on your machine for testing purposes

This guide covers installing Vitess locally for testing purposes, from pre-compiled binaries. We will launch multiple copies of `mysqld`, so it is recommended to have greater than 4GB RAM, as well as 20GB of available disk space.

A docker setup is also available, which requires no dependencies on your local host.

Install MySQL and etcd

Vitess supports MySQL 5.6+ and MariaDB 10.0+. We recommend MySQL 5.7 if your installation method provides a choice:

```
# Ubuntu based
sudo apt install -y mysql-server etcd curl

# Debian
sudo apt install -y default-mysql-server default-mysql-client etcd curl

# Yum based
sudo yum -y localinstall
   https://dev.mysql.com/get/mysql57-community-release-el7-9.noarch.rpm
sudo yum -y install mysql-community-server etcd curl
```

On apt-based distributions the services `mysqld` and `etcd` will need to be shutdown, since `etcd` will conflict with the `etcd` started in the examples, and `mysqlctl` will start its own copies of `mysqld`:

```
# Debian and Ubuntu
sudo service mysql stop
sudo service etcd stop
sudo systemctl disable mysql
sudo systemctl disable etcd
```

Disable AppArmor or SELinux

AppArmor/SELinux will not allow Vitess to launch MySQL in any data directory by default. You will need to disable it:

AppArmor:

```
# Debian and Ubuntu
sudo ln -s /etc/apparmor.d/usr.sbin.mysqld /etc/apparmor.d/disable/
sudo apparmor_parser -R /etc/apparmor.d/usr.sbin.mysqld

# The following command should return an empty result:
sudo aa-status | grep mysqld
```

SELinux:

```
# CentOS
sudo setenforce 0
```

Install Vitess

Download the latest binary release for Vitess on Linux. For example with Vitess 6:

```
version=6.0.20-20200818
file=vitess-${version}-90741b8.tar.gz
wget https://github.com/vitessio/vitess/releases/download/v${version}/${file}
tar -xzf ${file}
cd ${file/.tar.gz/}
sudo mkdir -p /usr/local/vitess
sudo cp -r * /usr/local/vitess/
```

Make sure to add `/usr/local/vitess/bin` to the `PATH` environment variable. You can do this by adding the following to your `$HOME/.bashrc` file:

```
export PATH=/usr/local/vitess/bin:${PATH}
```

You are now ready to start your first cluster! Open a new terminal window to ensure your `.bashrc` file changes take effect.

Start a Single Keyspace Cluster

Start by copying the local examples included with Vitess to your preferred location. For our first example we will deploy a single unsharded keyspace. The file `101_initial_cluster.sh` is for example `1` phase `01`. Lets execute it now:

```
cp -r /usr/local/vitess/examples/local ~/my-vitess-example
cd ~/my-vitess-example
./101_initial_cluster.sh
```

You should see output similar to the following:

```
~/my-vitess-example> ./101_initial_cluster.sh
$ ./101_initial_cluster.sh
add /vitess/global
add /vitess/zone1
add zone1 CellInfo
etcd start done...
Starting vtctld...
Starting MySQL for tablet zone1-0000000100...
Starting vttablet for zone1-0000000100...
HTTP/1.1 200 OK
Date: Wed, 25 Mar 2020 17:32:45 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000101...
Starting vttablet for zone1-0000000101...
HTTP/1.1 200 OK
Date: Wed, 25 Mar 2020 17:32:53 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000102...
Starting vttablet for zone1-0000000102...
HTTP/1.1 200 OK
Date: Wed, 25 Mar 2020 17:33:01 GMT
Content-Type: text/html; charset=utf-8

W0325 11:33:01.932674   16036 main.go:64] W0325 17:33:01.930970 reparent.go:185]
   master-elect tablet zone1-0000000100 is not the shard master, proceeding anyway as
   -force was used
```

```
W0325 11:33:01.933188   16036 main.go:64] W0325 17:33:01.931580 reparent.go:191]
   master-elect tablet zone1-0000000100 is not a master in the shard, proceeding anyway as
   -force was used
..
```

You can also verify that the processes have started with `pgrep`:

```
~/my-vitess-example> pgrep -fl vtdataroot
14119 etcd
14176 vtctld
14251 mysqld_safe
14720 mysqld
14787 vttablet
14885 mysqld_safe
15352 mysqld
15396 vttablet
15492 mysqld_safe
15959 mysqld
16006 vttablet
16112 vtgate
```

The exact list of processes will vary. For example, you may not see `mysqld_safe` listed.

If you encounter any errors, such as ports already in use, you can kill the processes and start over:

```
pkill -9 -e -f '(vtdataroot|VTDATAROOT)' # kill Vitess processes
rm -rf vtdataroot
```

Setup Aliases

For ease-of-use, Vitess provides aliases for `mysql` and `vtctlclient`:

```
source ./env.sh
```

Setting up aliases changes `mysql` to always connect to Vitess for your current session. To revert this, type `unalias mysql && unalias vtctlclient` or close your session.

Connect to your cluster

You should now be able to connect to the VTGate server that was started in `101_initial_cluster.sh`:

```
~/my-vitess-example> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.9-Vitess (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+----------------------+
| Tables_in_vt_commerce |
```

```
+----------------------+
| corder               |
| customer             |
| product              |
+----------------------+
3 rows in set (0.00 sec)
```

You can also browse to the vtctld console using the following URL:

```
http://localhost:15000
```

Summary

In this example, we deployed a single unsharded keyspace named `commerce`. Unsharded keyspaces have a single shard named `0`. The following schema reflects a common ecommerce scenario that was created by the script:

```
create table product (
  sku varbinary(128),
  description varbinary(128),
  price bigint,
  primary key(sku)
);
create table customer (
  customer_id bigint not null auto_increment,
  email varbinary(128),
  primary key(customer_id)
);
create table corder (
  order_id bigint not null auto_increment,
  customer_id bigint,
  sku varbinary(128),
  price bigint,
  primary key(order_id)
);
```

The schema has been simplified to include only those fields that are significant to the example:

- The `product` table contains the product information for all of the products.
- The `customer` table has a `customer_id` that has an `auto_increment`. A typical customer table would have a lot more columns, and sometimes additional detail tables.
- The `corder` table (named so because `order` is an SQL reserved word) has an `order_id` auto-increment column. It also has foreign keys into `customer(customer_id)` and `product(sku)`.

Next Steps

You can now proceed with MoveTables.

Or alternatively, if you would like to teardown your example:

```
./401_teardown.sh
rm -rf vtdataroot
```

Vitess Operator for Kubernetes

PlanetScale provides a Vitess Operator for Kubernetes, released under the Apache 2.0 license. The following steps show how to get started using Minikube:

Prerequisites

Before we get started, let's get a few things out of the way:

1. Install Minikube and start a Minikube engine. We recommend using Kubernetes 1.14, as this is a common denominator across public clouds: `bash minikube start --kubernetes-version=v1.14.10 --cpus=8 --memory=11000 --disk-size=50g`

   If you do not have a machine with 11GB of memory to spare, you could also consider using GKE instead. An equivalent setup can be deployed from the Cloud Shell with: `bash gcloud container clusters create vitess --cluster-version 1.14 --zone us-east1-b --num-nodes 5`

2. Install kubectl and ensure it is in your `PATH`. For example, on Linux:

   ```
   curl -LO
      https://storage.googleapis.com/kubernetes-release/release/v1.14.9/bin/linux/amd64/kubectl
   ```

3. Install the MySQL client locally. For example, on Ubuntu:

   ```
   apt install mysql-client
   ```

4. Install vtctlclient locally:

   If you are familiar with Go development, the easiest way to do this is: `bash go get vitess.io/vitess/go/cmd/vtctlclient` If not, you can also download the latest Vitess release and extract `vtctlclient` from it.

Install the Operator

Change to the operator example directory:

```
git clone git@github.com:vitessio/vitess.git
cd vitess/examples/operator
```

Install the operator:

```
kubectl apply -f operator.yaml
```

Bring up an initial cluster

In this directory, you will see a group of yaml files. The first digit of each file name indicates the phase of example. The next two digits indicate the order in which to execute them. For example, `101_initial_cluster.yaml` is the first file of the first phase. We shall execute that now:

```
kubectl apply -f 101_initial_cluster.yaml
```

{{< info >}} We have supplied an example yaml for bringing up Vitess with the experimental vtorc component. You can try this out by using the following command: `kubectl apply -f vtorc_example.yaml`. Once `vtorc` is officially released, the examples will be updated accordingly. {{< /info >}}

Verify cluster You can check the state of your cluster with `kubectl get pods`. After a few minutes, it should show that all pods are in the status of running:

```
$ kubectl get pods
NAME                                        READY   STATUS    RESTARTS   AGE
example-etcd-faf13de3-1                     1/1     Running   0          78s
example-etcd-faf13de3-2                     1/1     Running   0          78s
example-etcd-faf13de3-3                     1/1     Running   0          78s
example-vttablet-zone1-2469782763-bfadd780  3/3     Running   1          78s
```

```
example-vttablet-zone1-2548885007-46a852d0       3/3     Running   1          78s
example-zone1-vtctld-1d4dcad0-59d8498459-kwz6b   1/1     Running   2          78s
example-zone1-vtgate-bc6cde92-6bd99c6888-vwcj5   1/1     Running   2          78s
vitess-operator-8454d86687-4wfnc                 1/1     Running   0          2m29s
```

Setup Port-forward

{{< warning >}} The port-forward will only forward to a specific pod. Currently, `kubectl` does not automatically terminate a port-forward as the pod disappears due to apply/upgrade operations. You will need to manually restart the port-forward. {{</ warning >}}

For ease-of-use, Vitess provides a script to port-forward from Kubernetes to your local machine. This script also recommends setting up aliases for `mysql` and `vtctlclient`:

```
./pf.sh &
alias vtctlclient="vtctlclient -server=localhost:15999"
alias mysql="mysql -h 127.0.0.1 -P 15306 -u user"
```

Setting up aliases changes `mysql` to always connect to Vitess for your current session. To revert this, type `unalias mysql && unalias vtctlclient` or close your session.

Create Schema

Load our initial schema:

```
vtctlclient ApplySchema -sql="$(cat create_commerce_schema.sql)" commerce
vtctlclient ApplyVSchema -vschema="$(cat vschema_commerce_initial.json)" commerce
```

Connect to your cluster You should now be able to connect to the VTGate Server in your cluster with the MySQL client:

```
~/vitess/examples/operator$ mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.9-Vitess MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----------+
| Databases |
+-----------+
| commerce  |
+-----------+
1 row in set (0.00 sec)
```

Summary In this example, we deployed a single unsharded keyspace named `commerce`. Unsharded keyspaces have a single shard named `0`. The following schema reflects a common ecommerce scenario that was created by the script:

```
create table product(
  sku varbinary(128),
  description varbinary(128),
  price bigint,
  primary key(sku)
);
create table customer(
  customer_id bigint not null auto_increment,
  email varbinary(128),
  primary key(customer_id)
);
create table corder(
  order_id bigint not null auto_increment,
  customer_id bigint,
  sku varbinary(128),
  price bigint,
  primary key(order_id)
);
```

The schema has been simplified to include only those fields that are significant to the example:

- The `product` table contains the product information for all of the products.
- The `customer` table has a `customer_id` that has an `auto_increment`. A typical customer table would have a lot more columns, and sometimes additional detail tables.
- The `corder` table (named so because `order` is an SQL reserved word) has an `order_id` auto-increment column. It also has foreign keys into `customer(customer_id)` and `product(sku)`.

Next Steps

You can now proceed with MoveTables.

Or alternatively, if you would like to teardown your example:

```
kubectl delete -f 101_initial_cluster.yaml
```

Congratulations on completing this exercise!

Vttestserver Docker Image

This guide covers using the vttestserver docker image for testing purposes. This is also the docker image that we use for testing in Vitess Framewok Testing.

Get the docker image

The first step is to get the docker image. There are two ways of doing this :

1. **From the vitessio/vitess repository**

Check out the vitessio/vitess repository Clone the GitHub repository via:

- SSH: `git clone git@github.com:vitessio/vitess.git`, or:
- HTTP: `git clone https://github.com/vitessio/vitess.git`

```
cd vitess
```

Build the docker image In your shell, execute:

```
make docker_vttestserver
```

This creates 2 docker images named `vitess/vttestserver:mysql57` and `vitess/vttestserver:mysql80`

2. Pulling from docker hub Alternately, you can get the latest docker images from the docker hub. In your shell, execute:

```
docker pull vitess/vttestserver:mysql57
docker pull vitess/vttestserver:mysql80
```

Run the docker image

At this point, you should have a docker image named `vitess/vttestserver:mysql57` or `vitess/vttestserver:mysql80`.

Environment variables The docker image expects some of the environment variables to be set to function properly. The following table lists all the environment variables available along with their usages.

Environment variable	Required	Use
KEYSPACES	yes	Specifies the names of the keyspaces to be created as a comma separated value.
NUM_SHARDS	yes	Specifies the number of shards in each keyspace. It is a comma separated value as well, read in conjunction with the KEYSPACES.
PORT	yes	The starting of the port addresses that vitess will use to register its components like vtgate, etc.
MYSQL_MAX_CONNECTIONS	no	Maximum number of connections that the MySQL instance will support. If unspecified, it defaults to 1000.
MYSQL_BIND_HOST	no	Which host to bind the MySQL listener to. If unspecified, it defaults to `127.0.0.1`.
MYSQL_SERVER_VERSION	no	MySQL server version to advertise. If unspecified, it defaults to `8.0.21-vitess` or `5.7.9-vitess` according to the version of vttestserver run.
CHARSET	no	Default charset to use. If unspecified, it defaults to `utf8mb4`.
FOREIGN_KEY_MODE	no	This is to provide how to handle foreign key constraint in create/alter table. Valid values are: allow (default), disallow.
ENABLE_ONLINE_DDL	no	Allow users to submit, review and control Online DDL. Valid values are: true (default), false.
ENABLE_DIRECT_DDL	no	Allow users to submit direct DDL statements. Valid values are: true (default), false.

Environment variables in docker can be specified using the `-e` aka `--env` flag.

Sending queries to vttestserver container from outside The vtgate listens for MySQL connections on 3 + the `PORT` environment variable specified. i.e. if you specify `PORT` to be 33574, then vtgate will be listening to connections on 33577, on the `MYSQL_BIND_HOST` which defaults to localhost. But this port will be on the docker container side. To connect to vtgate externally from a MySQL client, you will need to publish that port as well and specify the `MYSQL_BIND_HOST` to `0.0.0.0`. This can be done via the `-p` aka `--publish` flag to docker. For eg: adding `-p 33577:33577` to the `docker run` command will publish the container's 33577 port to your local 33577 port, which can now be used to connect to the vtgate.

Example

An example command to run the docker image is as follows :

```
docker run --name=vttestserver -p 33577:33577 -e PORT=33574 -e KEYSPACES=test,unsharded -e
   NUM_SHARDS=2,1 -e MYSQL_MAX_CONNECTIONS=70000 -e MYSQL_BIND_HOST=0.0.0.0
   --health-cmd="mysqladmin ping -h127.0.0.1 -P33577" --health-interval=5s
   --health-timeout=2s --health-retries=5 vitess/vttestserver:mysql57
```

Now, we can connect to the vtgate from a MySQL client as follows :

```
mysql --host 127.0.0.1 --port 33577 --user "root"
```

We have 2 keyspaces which we can use, `test` which has 2 shards and `unsharded` which has a single shard.

Overview

description: High-level information about Vitess

The Vitess overview documentation provides general information about Vitess that's less immediately practical than what you'll find in Get Started section and the User Guides.

Architecture

The Vitess platform consists of a number of server processes, command-line utilities, and web-based utilities, backed by a consistent metadata store.

Depending on the current state of your application, you could arrive at a full Vitess implementation through a number of different process flows. For example, if you're building a service from scratch, your first step with Vitess would be to define your database topology. However, if you need to scale your existing database, you'd likely start by deploying a connection proxy.

Vitess tools and servers are designed to help you whether you start with a complete fleet of databases or start small and scale over time. For smaller implementations, vttablet features like connection pooling and query rewriting help you get more from your existing hardware. Vitess' automation tools then provide additional benefits for larger implementations.

The diagram below illustrates Vitess' components:

For additional details on each of the components, see Concepts.

Cloud Native

Vitess is well-suited for Cloud deployments because it enables databases to incrementally add capacity. The easiest way to run Vitess is via Kubernetes.

Vitess on Kubernetes

Kubernetes is an open-source orchestration system for Docker containers, and Vitess can run as a Kubernetes-aware cloud native distributed database.

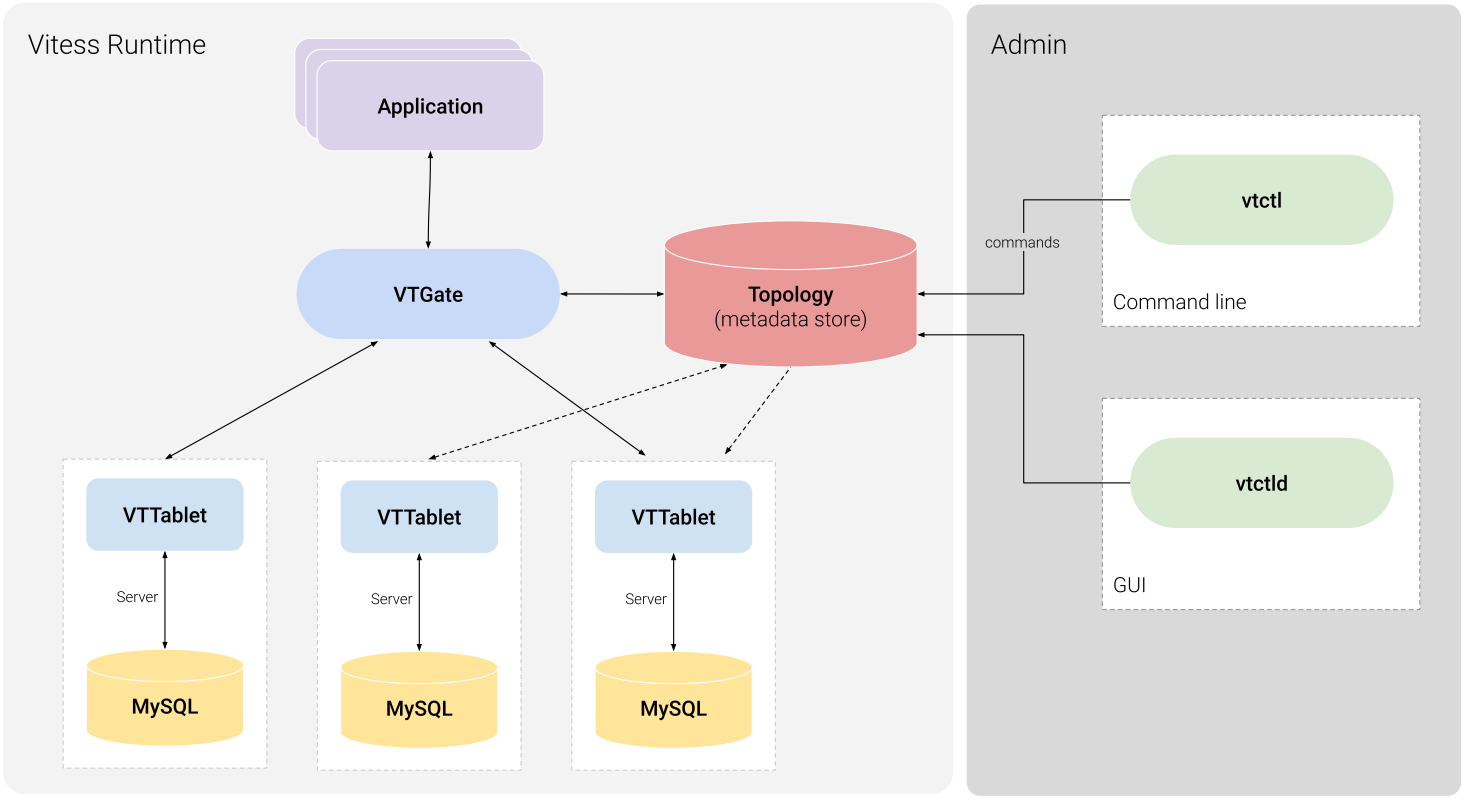


Figure 2: Architecture Diagram

Kubernetes handles scheduling onto nodes in a compute cluster, actively manages workloads on those nodes, and groups containers comprising an application for easy management and discovery. This provides an analogous open-source environment to the way Vitess runs in YouTube, on the predecessor to Kubernetes.

Related Vitess Documentation

- Kubernetes Quickstart

History

description: Born at YouTube, released as Open Source

Vitess was created in 2010 to solve the MySQL scalability challenges that the team at YouTube faced. This section briefly summarizes the sequence of events that led to Vitess' creation:

1. YouTube's MySQL database reached a point when peak traffic would soon exceed the database's serving capacity. To temporarily alleviate the problem, YouTube created a master database for write traffic and a replica database for read traffic.
2. With demand for cat videos at an all-time high, read-only traffic was still high enough to overload the replica database. So YouTube added more replicas, again providing a temporary solution.
3. Eventually, write traffic became too high for the master database to handle, requiring YouTube to shard data to handle incoming traffic. As an aside, sharding would have also become necessary if the overall size of the database became too large for a single MySQL instance.
4. YouTube's application layer was modified so that before executing any database operation, the code could identify the right database shard to receive that particular query.

Vitess let YouTube remove that logic from the source code, introducing a proxy between the application and the database to route and manage database interactions. Since then, YouTube has scaled its user base by a factor of more than 50, greatly

increasing its capacity to serve pages, process newly uploaded videos, and more. Even more importantly, Vitess is a platform that continues to scale.

Vitess becomes a CNCF project

The CNCF serves as the vendor-neutral home for many of the fastest-growing open source projects. In February 2018 the Technical Oversight Committee (TOC) voted to accept Vitess as a CNCF incubation project. Vitess became the eighth CNCF project to graduate in November 2019, joining Kubernetes, Prometheus, Envoy, CoreDNS, containerd, Fluentd, and Jaeger.

Scalability Philosophy

Scalability problems can be solved using many approaches. This document describes Vitess' approach to address these problems.

Small instances

When deciding to shard or break databases up into smaller parts, it's tempting to break them just enough that they fit in one machine. In the industry, it's common to run only one MySQL instance per host.

Vitess recommends that instances be broken up into manageable chunks (250GB per MySQL server), and not to shy away from running multiple instances per host. The net resource usage would be about the same. But the manageability greatly improves when MySQL instances are small. There is the complication of keeping track of ports, and separating the paths for the MySQL instances. However, everything else becomes simpler once this hurdle is crossed.

There are fewer lock contentions to worry about, replication is a lot happier, production impact of outages become smaller, backups and restores run faster, and a lot more secondary advantages can be realized. For example, you can shuffle instances around to get better machine or rack diversity leading to even smaller production impact on outages, and improved resource usage.

Durability through replication

Traditional data storage software treated data as durable as soon as it was flushed to disk. However, this approach is impractical in today's world of commodity hardware. Such an approach also does not address disaster scenarios.

The new approach to durability is achieved by copying the data to multiple machines, and even geographical locations. This form of durability addresses the modern concerns of device failures and disasters.

Many of the workflows in Vitess have been built with this approach in mind. For example, turning on semi-sync replication is highly recommended. This allows Vitess to failover to a new replica when a master goes down, with no data loss. Vitess also recommends that you avoid recovering a crashed database. Instead, create a fresh one from a recent backup and let it catch up.

Relying on replication also allows you to loosen some of the disk-based durability settings. For example, you can turn off `sync_binlog`, which greatly reduces the number of IOPS to the disk thereby increasing effective throughput.

Consistency model

Before sharding or moving tables to different keyspaces, the application needs to be verified (or changed) such that it can tolerate the following changes:

- Cross-shard reads may not be consistent with each other. Conversely, the sharding decision should also attempt to minimize such occurrences because cross-shard reads are more expensive.
- In "best-effort mode", cross-shard transactions can fail in the middle and result in partial commits. You could instead use "2PC mode" transactions that give you distributed atomic guarantees. However, choosing this option increases the write cost by approximately 50%.

Single shard transactions continue to remain ACID, just like MySQL supports it.

If there are read-only code paths that can tolerate slightly stale data, the queries should be sent to REPLICA tablets for OLTP, and RDONLY tablets for OLAP workloads. This allows you to scale your read traffic more easily, and gives you the ability to distribute them geographically.

This trade-off allows for better throughput at the expense of stale or possibly inconsistent reads, since the reads may be lagging behind the master, as data changes (and possibly with varying lag on different shards). To mitigate this, VTGate servers are capable of monitoring replica lag and can be configured to avoid serving data from instances that are lagging beyond X seconds.

For a true snapshot, queries must be sent to the master within a transaction. For read-after-write consistency, reading from the master without a transaction is sufficient.

To summarize, these are the various levels of consistency supported:

- `REPLICA/RDONLY` read: Servers can be scaled geographically. Local reads are fast, but can be stale depending on replica lag.
- `MASTER` read: There is only one worldwide master per shard. Reads coming from remote locations will be subject to network latency and reliability, but the data will be up-to-date (read-after-write consistency). The isolation level is `READ_COMMITTED`.
- `MASTER` transactions: These exhibit the same properties as MASTER reads. However, you get REPEATABLE_READ consistency and ACID writes for a single shard. Support is underway for cross-shard Atomic transactions.

As for atomicity, the following levels are supported:

- `SINGLE`: disallow multi-db transactions.
- `MULTI`: multi-db transactions with best effort commit.
- `TWOPC`: multi-db transactions with 2PC commit.

No multi-master Vitess doesn't support multi-master setup. It has alternate ways of addressing most of the use cases that are typically solved by multi-master:

- Scalability: There are situations where multi-master gives you a little bit of additional runway. However, since the statements have to eventually be applied to all masters, it's not a sustainable strategy. Vitess addresses this problem through sharding, which can scale indefinitely.
- High availability: Vitess integrates with Orchestrator, which is capable of performing a failover to a new master within seconds of failure detection. This is usually sufficient for most applications.
- Low-latency geographically distributed writes: This is one case that is not addressed by Vitess. The current recommendation is to absorb the latency cost of long-distance round-trips for writes. If the data distribution allows, you still have the option of sharding based on geographic affinity. You can then setup masters for different shards to be in different geographic location. This way, most of the master writes can still be local.

Multi-cell

Vitess is meant to run in multiple data centers / regions / cells. In this part, we'll use "cell" to mean a set of servers that are very close together, and share the same regional availability.

A cell typically contains a set of tablets, a vtgate pool, and app servers that use the Vitess cluster. With Vitess, all components can be configured and brought up as needed:

- The master for a shard can be in any cell. If cross-cell master access is required, vtgate can be configured to do so easily (by passing the cell that contains the master as a cell to watch).
- It is not uncommon to have the cells that can contain the master be more provisioned than read-only serving cells. These *master-capable* cells may need one more replica to handle a possible failover, while still maintaining the same replica serving capacity.
- Failing over from a master in one cell to a master in a different cell is no different than a local failover. It has an implication on traffic and latency, but if the application traffic also gets re-directed to the new cell, the end result is stable.

- It is also possible to have some shards with a master in one cell, and some other shards with their master in another cell. vtgate will just route the traffic to the right place, incurring extra latency cost only on the remote access. For instance, creating U.S. user records in a database with masters in the U.S. and European user records in a database with masters in Europe is easy to do. Replicas can exist in every cell anyway, and serve the replica traffic quickly.
- Replica serving cells are a good compromise to reduce user-visible latency: they only contain replica servers, and master access is always done remotely. If the application profile is mostly reads, this works really well.
- Not all cells need `rdonly` (or batch) instances. Only the cells that run batch jobs, or OLAP jobs, really need them.

Note Vitess uses local-cell data first, and is very resilient to any cell going down (most of our processes handle that case gracefully).

Supported Databases

Vitess deploys, scales and manages clusters of open-source SQL database instances. Currently, Vitess supports the MySQL, Percona and MariaDB databases.

The VTGate proxy server advertises its version as MySQL 5.7.

MySQL versions 5.6 to 8.0

Vitess supports the core features of MySQL versions 5.6 to 8.0, with some limitations. Vitess also supports Percona Server for MySQL versions 5.6 to 8.0.

With MySQL 5.6 reaching end of life in February 2021, it is recommended to deploy MySQL 5.7 and later.

MariaDB versions 10.0 to 10.3

Vitess supports the core features of MariaDB versions 10.0 to 10.3. Vitess does not yet support version 10.4 of MariaDB.

See also

- MySQL Compatibility

What Is Vitess

Vitess is a database solution for deploying, scaling and managing large clusters of open-source database instances. It currently supports MySQL and MariaDB. It's architected to run as effectively in a public or private cloud architecture as it does on dedicated hardware. It combines and extends many important SQL features with the scalability of a NoSQL database. Vitess can help you with the following problems:

1. Scaling a SQL database by allowing you to shard it, while keeping application changes to a minimum.
2. Migrating from baremetal to a private or public cloud.
3. Deploying and managing a large number of SQL database instances.

Vitess includes compliant JDBC and Go database drivers using a native query protocol. Additionally, it implements the MySQL server protocol which is compatible with virtually any other language.

Vitess served all YouTube database traffic for over five years. Many enterprises have now adopted Vitess for their production needs.

Features

- Performance
  - Connection pooling - Multiplex front-end application queries onto a pool of MySQL connections to optimize performance.
  - Query de-duping – Reuse results of an in-flight query for any identical requests received while the in-flight query was still executing.
  - Transaction manager – Limit number of concurrent transactions and manage timeouts to optimize overall throughput.
- Protection
  - Query rewriting and sanitization – Add limits and avoid non-deterministic updates.
  - Query blacklisting – Customize rules to prevent potentially problematic queries from hitting your database.
  - Query killer – Terminate queries that take too long to return data.
  - Table ACLs – Specify access control lists (ACLs) for tables based on the connected user.
- Monitoring
  - Performance analysis tools let you monitor, diagnose, and analyze your database performance.
- Topology Management Tools
  - Master management tools (handles reparenting)
  - Web-based management GUI
  - Designed to work in multiple data centers / regions
- Sharding
  - Virtually seamless dynamic re-sharding
  - Vertical and Horizontal sharding support
  - Multiple sharding schemes, with the ability to plug-in custom ones

Comparisons to other storage options

The following sections compare Vitess to two common alternatives, a vanilla MySQL implementation and a NoSQL implementation.

Vitess vs. Vanilla MySQL Vitess improves a vanilla MySQL implementation in several ways:

Vanilla MySQL	Vitess
Every MySQL connection has a memory overhead that ranges between 256KB and almost 3MB, depending on which MySQL release you're using. As your user base grows, you need to add RAM to support additional connections, but the RAM does not contribute to faster queries. In addition, there is a significant CPU cost associated with obtaining the connections.	Vitess creates very lightweight connections. Vitess' connection pooling feature uses Go's concurrency support to map these lightweight connections to a small pool of MySQL connections. As such, Vitess can easily handle thousands of connections.
Poorly written queries, such as those that don't set a LIMIT, can negatively impact database performance for all users.	Vitess employs a SQL parser that uses a configurable set of rules to rewrite queries that might hurt database performance.
Sharding is a process of partitioning your data to improve scalability and performance. MySQL lacks native sharding support, requiring you to write sharding code and embed sharding logic in your application.	Vitess supports a variety of sharding schemes. It can also migrate tables into different databases and scale up or down the number of shards. These functions are performed non-intrusively, completing most data transitions with just a few seconds of read-only downtime.
A MySQL cluster using replication for availability has a master database and a few replicas. If the master fails, a replica should become the new master. This requires you to manage the database lifecycle and communicate the current system state to your application.	Vitess helps to manage the lifecycle of your database scenarios. It supports and automatically handles various scenarios, including master failover and data backups.

Vanilla MySQL	Vitess
A MySQL cluster can have custom database configurations for different workloads, like a master database for writes, fast read-only replicas for web clients, slower read-only replicas for batch jobs, and so forth. If the database has horizontal sharding, the setup is repeated for each shard, and the app needs baked-in logic to know how to find the right database.	Vitess uses a topology backed by a consistent data store, like etcd or ZooKeeper. This means the cluster view is always up-to-date and consistent for different clients. Vitess also provides a proxy that routes queries efficiently to the most appropriate MySQL instance.

Vitess vs. NoSQL If you're considering a NoSQL solution primarily because of concerns about the scalability of MySQL, Vitess might be a more appropriate choice for your application. While NoSQL provides great support for unstructured data, Vitess still offers several benefits not available in NoSQL datastores:

NoSQL	Vitess
NoSQL databases do not define relationships between database tables, and only support a subset of the SQL language.	Vitess is not a simple key-value store. It supports complex query semantics such as where clauses, JOINS, aggregation functions, and more.
NoSQL datastores do not usually support transactions.	Vitess supports transactions.
NoSQL solutions have custom APIs, leading to custom architectures, applications, and tools.	Vitess adds very little variance to MySQL, a database that most people are already accustomed to working with.
NoSQL solutions provide limited support for database indexes compared to MySQL.	Vitess allows you to use all of MySQL's indexing functionality to optimize query performance.

Older Version Docs

description: PDFs of vitess.io/docs at the time of previous versions disable_expand: true

The Vitess Documentation currently posted should reflect the most recent code.

To view a PDF of the state of the Vitess Documentation on the date of the general availability release for previous versions please follow the links below:

- Vitess Docs 10.0 on April 27, 2021
- Vitess Docs 9.0 on January 26, 2021
- Vitess Docs 8.0 on October 27, 2020
- Vitess Docs 7.0 on July 28, 2020
- Vitess Docs 6.0 on April 29, 2020
- Vitess Docs 5.0 on February 4, 2020

Reference

description: Detailed information about specific Vitess functionality

Compatibility

description: Reference documents for MySQL compatibility with Vitess

MySQL Compatibility

VTGate servers speak both gRPC and the MySQL server protocol. This allows you to connect to Vitess as if it were a MySQL Server without any changes to application code. This document refers to known compatibility issues where Vitess differs from MySQL.

Transaction Model

Vitess provides `READ COMMITTED` semantics when executing cross-shard queries. This differs to MySQL, which defaults to `REPEATABLE READ`.

SQL Syntax

The following describes some of the major differences in SQL Syntax handling between Vitess and MySQL. For a list of unsupported queries, check out the test-suite cases.

DDL Vitess supports MySQL DDL, and will send `ALTER TABLE` statements to each of the underlying tablet servers. For large tables it is recommended to use an external schema deployment tool and apply directly to the underlying MySQL shard instances. This is discussed further in Applying MySQL Schema.

Join Queries Vitess supports `INNER JOIN` including cross-shard joins. `LEFT JOIN` is supported as long as there are not expressions that compare columns on the outer table to the inner table in sharded keyspaces.

Ordering Vitess supports the ordering for all the columns irrespective of the data type. However it further requests collation specific information (`weight_string`) from the underlying MySQL instance. This can be avoided for numeric columns which is discussed further in Advanced VSchema Properties

Aggregation Vitess supports a subset of `GROUP BY` operations, including cross-shard operations. The VTGate servers are capable of scatter-gather operations, but can only stream results. Thus, a query that performs a `GROUP BY colx ORDER BY coly` may be refused if the intermediate result set is larger than VTGate's in-memory limit.

Subqueries Vitess supports a subset of subqueries. For example, a subquery combined with a `GROUP BY` operation is not supported.

Stored Procedures Calling stored procedures using CALL is only supported for:

- unsharded keyspaces
- if you directly target a specific shard

There are further limitations to calling stored procedures using CALL:

- The stored procedure CALL cannot return any results
- Only IN parameters are supported
- If you use transactions, the transaction state cannot be changed by the stored procedure.

  For example, if there is a transaction open at the beginning of the CALL, a transaction must still be open after the procedure finishes. Likewise, if no transaction is open at the beginning of the CALL, the stored procedure must not leave an open transaction after execution finishes.

CREATE PROCEDURE is not supported. You have to create the procedure directly on the underlying MySQL servers and not through Vitess.

Window Functions and CTEs Vitess does not yet support Window Functions or Common Table Expressions.

Killing running queries Vitess does not yet support killing running shard queries via the `KILL` command through VTGate. Vitess does have strict query timeouts for OLTP workloads (see below). If you need a query, you can connect to the underlying MySQL shard instance and run `KILL` from there.

Cross-shard Transactions By default, Vitess does not support transactions that span across shards. While Vitess can support this with the use of Two-Phase Commit, it is usually recommended to design the VSchema in such a way that cross-shard modifications are not required.

OLAP Workload By default, Vitess sets some intentional restrictions on the execution time and number of rows that a query can return. This default workload mode is called `OLTP`. This can be disabled by setting the workload to `OLAP`:

```
SET workload='olap'
```

SELECT ... INTO Statement The `SELECT ... INTO` form of `SELECT` in MySQL enables a query result to be stored in variables or written to a file. Vitess supports `SELECT ... INTO DUMFILE` and `SELECT ... INTO OUTFILE` constructs for unsharded keyspaces but does not support storing results in variable. Moreover, the position of `INTO` must be towards the end of the query and not in the middle. An example of a correct query is as follows:

```
SELECT * FROM <tableName> INTO OUTFILE 'x.txt' FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED
   BY '"' ESCAPED BY '\t' LINES TERMINATED BY '\ n'
```

For sharded keyspaces this statement can still be used but only after specifying the exact shard with a USE Statement.

LOAD DATA Statement `LOAD DATA` is the complement of `SELECT ... INTO OUTFILE` that reads rows from a text file into a table at a very high speed. Just like `SELECT ... INTO` statement, `LOAD DATA` is also supported in unsharded keyspaces. An example of a correct query is as follows:

```
LOAD DATA INFILE 'x.txt' INTO REPLACE TABLE <tableName> FIELDS TERMINATED BY ';' OPTIONALLY
   ENCLOSED BY '"' ESCAPED BY '\t' LINES TERMINATED BY '\ n'
```

For sharded keyspaces this statement can still be used but only after specifying the exact shard with a USE Statement.

Network Protocol

Prepared Statements Starting with version 4.0, Vitess features experimental support for prepared statements via the MySQL protocol. Session-based commands using the `PREPARE` and `EXECUTE` SQL statements are not supported.

Authentication Plugins Vitess supports the `mysql_native_password` authentication plugin. Support for `caching_sha2_password` can be tracked in #5399.

Transport Security To configure VTGate to support `TLS` set `-mysql_server_ssl_cert` and `-mysql_server_ssl_key`. Client certificates can also be mandated by setting `-mysql_server_ssl_ca`. If there is no CA specified then TLS is optional.

Temporary Tables

Vitess has limited support for temporary tables. It works only for unsharded keyspaces.

If the user creates a temporary table then the session will start using reserved connections for any query sent on that session.

The query plans generated by this session will not be cached. It will still continue to use the query plan cached from other non-temporary table sessions.

Character Set and Collation

Vitess only supports `utf8` and variants such as `utf8mb4`.

SQL Mode

Vitess behaves similar to the `STRICT_TRANS_TABLES` sql mode, and does not recommend changing the SQL Mode setting.

Data Types

Vitess supports all of the data types available in MySQL. Using the `FLOAT` data type as part of a `PRIMARY KEY` is strongly discouraged, since features such as filtered replication and VReplication will not correctly be able to detect which rows should be included as part of a modification.

Auto Increment

Tables in sharded keyspaces do not support the `auto_increment` column attribute, as the values generated would be local only to each shard. Vitess Sequences are provided as an alternative, which have very close semantics to `auto_increment`.

Extensions to MySQL Syntax

SHOW Statements Vitess supports a few additional options with the SHOW statement.

- `SHOW keyspaces` – A list of keyspaces available.

```
Example Output:
+-----------+
| Database |
+-----------+
| commerce |
| customer |
+-----------+
```

- `SHOW vitess_tablets` – Information about the current Vitess tablets such as the keyspace, key ranges, tablet type, hostname, and status.

```
Example Output:
+-------+----------+-------+------------+---------+------------------+------------+---------------
| Cell  | Keyspace | Shard | TabletType | State   | Alias            | Hostname   |
   MasterTermStartTime  |
+-------+----------+-------+------------+---------+------------------+------------+---------------
| zone1 | commerce | 0     | MASTER     | SERVING | zone1-0000000100 | <redacted> |
   2021-04-22T04:10:29Z |
| zone1 | commerce | 0     | REPLICA    | SERVING | zone1-0000000101 | <redacted> |
                        |
| zone1 | commerce | 0     | RDONLY     | SERVING | zone1-0000000102 | <redacted> |
                        |
| zone1 | customer | -80   | MASTER     | SERVING | zone1-0000000300 | <redacted> |
   2021-04-22T04:12:23Z |
| zone1 | customer | -80   | REPLICA    | SERVING | zone1-0000000301 | <redacted> |
                        |
| zone1 | customer | -80   | RDONLY     | SERVING | zone1-0000000302 | <redacted> |
                        |
```

```
| zone1 | customer | 80-   | MASTER     | SERVING | zone1-0000000400 | <redacted> |
   2021-04-22T04:12:23Z |
| zone1 | customer | 80-   | REPLICA    | SERVING | zone1-0000000401 | <redacted> |
                        |
| zone1 | customer | 80-   | RDONLY     | SERVING | zone1-0000000402 | <redacted> |
                        |
+-------+----------+-------+------------+---------+------------------+------------+---------------
```

- `SHOW vitess_shards` – A list of shards that are available.

```
Example Output:
+--------------+
| Shards       |
+--------------+
| commerce/0   |
| customer/-80 |
| customer/80- |
+--------------+
```

- `SHOW vschema tables` – A list of tables available in the current keyspace's vschema.

```
Example Output for customer keyspace:
+----------+
| Tables   |
+----------+
| corder   |
| customer |
| dual     |
+----------+
```

- `SHOW vschema vindexes` – Information about the current keyspace's vindexes such as the keyspace, name, type, params, and owner. Optionally supports an "ON" clause with a table name.

```
Example Output:
+----------+------+------+--------+-------+
| Keyspace | Name | Type | Params | Owner |
+----------+------+------+--------+-------+
| customer | hash | hash |        |       |
+----------+------+------+--------+-------+
```

- `show global gtid_executed [FROM <keyspace>]` – retrieves the global gtid_executed from each shard in the keyspace either selected or provided in the query.

```
Example Output for customer keyspace:
+----------+------------------------------------------+-------+
| db_name  | gtid_executed                            | shard |
+----------+------------------------------------------+-------+
| customer | e9148eb0-a320-11eb-8026-98af65a6dc4a:1-43 | 80-   |
| customer | e0f64aca-a320-11eb-9be4-98af65a6dc4a:1-43 | -80   |
+----------+------------------------------------------+-------+
```

USE Statements Vitess allows you to select a keyspace using the MySQL `USE` statement, and corresponding binary API used by client libraries. SQL statements can refer to a table in another keyspace by using the standard *dot* notation:

```
SELECT * FROM my_other_keyspace.table;
```

Vitess extends this functionality further by allowing you to select a specific shard and tablet-type within a `USE` statement (backticks are important):

```
-- `KeyspaceName:shardKeyRange@tabletType`
USE `mykeyspace:-80@rdonly`
```

A similar effect can be achieved by using a database name like `mykeyspace:-80@rdonly` in your MySQL application client connection string.

Create/Drop Database Vitess does not support CREATE and DROP DATABASE queries out of the box.

But, to make it possible to provision databases, a plugin mechanism exists. The plugin has to take care of creating and dropping the database, and update the topology & VSchema so that Vitess can start receiving queries for the new keyspace.

The plugin should implement the `DBDDLPlugin` interface, and be saved into a new file in the `go/vt/vtgate/engine/` directory.

```
type DBDDLPlugin interface {
    CreateDatabase(ctx context.Context, name string) error
    DropDatabase(ctx context.Context, name string) error
}
```

It must then register itself calling `DBDDLRegister`. You can take a look at the `dbddl_plugin.go` in the engine package for an example of how it's done. Finally, you need to add a command line flag to vtgate to have it use the new plugin: `-dbddl_plugin=myPluginName`

Errors

description: Detailed information about errors that the user might run into

Query Serving

description: Errors a users might encounter while querying Vitess

{{< info >}} These error messages are internal to Vitess. If you are getting other errors from MySQL you can check them on this MySQL error page. {{< /info >}}

Error Number	Error State	Message	Meaning
1105	HY000	INSERT not supported when targeting a key range: %s	When targeting a range of shards, Vitess doesn't know which shard to send the INSERT to, so this is not supported.
1450	HY000	Changing schema from '%s' to '%s' is not allowed	Changing schema from rename command is not valid
1149	42000	aggregate functions take a single argument '%s'	This function only takes a single argument.

Error Number	Error State	Message	Meaning
1238	HY000	Variable ‘%s’ is a read only variable	Tried changing a read-only system variable
1105	HY000	column has duplicate set values: ‘%v’	Cannot assign multiple values to a column in an update statement
1231	42000	unexpected value type for ‘%s’: %v	Some system variables require a specific type to be used
1192	HY000	Can’t execute the given command because you have an active transaction	The provided statement cannot be executed inside a transaction.
1231	42000	invalid transaction_mode: %s	Valid transaction_mode values are ‘SINGLE’, ‘MULTI’ or ‘TWOPC’
1231	42000	invalid workload: %s	Valid workload values are ‘OLTP’, ‘OLAP’ or ‘DBA’
1231	42000	invalid DDL strategy: %s	Valid DDL strategies are gh-ost, pt-osc
1690	22003	%s value is out of range in %v [+,-,*,/] %v	Arithmetic operation lead to out of range value for the type
1047	42000	connection ID and transaction ID do not exist	The session is pointing to a transaction and/or reserved connection that is not valid
1105	HY000	%d is not a boolean	Tried setting a system variable to a value that could not be converted a boolean value
1105	HY000	%s is not a sequence	The given table is not a sequence table
1105	HY000	%s: system setting is not supported	Tried reading or setting a system variable that is not supported
1105	HY000	%v cannot be converted to a go type	This type can’t be represented as a golang value

Error Number	Error State	Message	Meaning
1105	HY000	2pc is not enabled	This functionality requires 2PC. Read more about 'transaction_mode' to learn how to enable it.
1105	HY000	Destination can only be a single shard for statement: %s, got: %v	This statement type can only be executed against a single shard. You need to change the target string so a single shard in targeted.
1105	HY000	GTIDSet Mismatch: requested source position:%v, current target vrep position: %v	The requested GTIDSet does not exist in the vrep stream.
1105	HY000	No target	TODO https://github.com/vitessio
1105	HY000	Unexpected error, DestinationKeyspaceID mapping to multiple shards	This is an internal error. If you see this error, please report it as a bug.
1105	HY000	auto sequence generation can happen through single shard only, it is getting routed to %d shards	A sequence query has to be routed to a single shard, but this query was not.
1105	HY000	cannot mix scope and user defined variables	If you use the `SET GLOBAL` form, specify the variable without any `@` symbols
1105	HY000	cannot use scope and @@	If you use the `SET GLOBAL` form, specify the variable without any `@` symbols
1105	HY000	column has duplicate set values: '%v'	An UPDATE query should only list a column to be updated once
1105	HY000	could not parse value: '%s'	Tried parsing a value as a number but failed
1105	HY000	disallowed due to rule: %s	The query was not permitted to execute because the session was lacking permissions to do so

Error Number	Error State	Message	Meaning
1105	HY000	expression is too complex ‘%v’	An expression was used that is not recognized by Vitess. Arithmetics and function calls are examples of expressions that are too complex in this context.
1105	HY000	invalid increment for sequence %s: %d	TODO
1105	HY000	invalid keyspace %v does not match expected %v	TODO
1105	HY000	invalid shard %v does not match expected %v	TODO
1105	HY000	invalid table name: %s	The table name contains invalid characters
1105	HY000	is not a boolean	This error will be returned if you try to set a variable to a value that can’t be converted to a boolean value.
1105	HY000	negative number cannot be converted to unsigned: %d	The column or variable is expecting an unsigned int, and negative numbers invalid here.
1105	HY000	query arguments missing for %s	Argument expected but was missing.
1105	HY000	require autocommit to be 1: got %s	Connection needs autocommit to be enabled, but it was not.
1105	HY000	require sql_auto_is_null to be 0: got %s	Vitess requires the connection not use the auto_col functionality
1105	HY000	require sql_mode to be STRICT_TRANS_TABLES or STRICT_ALL_TABLES: got ‘%s’	Vitess requires the connection to be in STRICT mode; either or both of these settings need to be enabled.
1105	HY000	unexpected rows from reading sequence %s (possible mis-route): %d	The sequence table used returned invalid results.

Error Number	Error State	Message	Meaning
1105	HY000	unsigned number overflows int64 value: %d	Tried to convert an unsigned integer into a signed integer, and the value overflows

Features

description: Reference documents for Vitess features

Messaging

Vitess messaging gives the application an easy way to schedule and manage work that needs to be performed asynchronously. Under the covers, messages are stored in a traditional MySQL table and therefore enjoy the following properties:

- **Scalable**: Because of vitess's sharding abilities, messages can scale to very large QPS or sizes.
- **Guaranteed delivery**: A message will be indefinitely retried until a successful ack is received.
- **Non-blocking**: If the sending is backlogged, new messages continue to be accepted for eventual delivery.
- **Adaptive**: Messages that fail delivery are backed off exponentially.
- **Analytics**: The retention period for messages is dictated by the application. One could potentially choose to never delete any messages and use the data for performing analytics.
- **Transactional**: Messages can be created or acked as part of an existing transaction. The action will complete only if the commit succeeds.

The properties of a message are chosen by the application. However, every message needs a uniquely identifiable key. If the messages are stored in a sharded table, the key must also be the primary vindex of the table.

Although messages will generally be delivered in the order they're created, this is not an explicit guarantee of the system. The focus is more on keeping track of the work that needs to be done and ensuring that it was performed. Messages are good for:

- Handing off work to another system.
- Recording potentially time-consuming work that needs to be done asynchronously.
- Scheduling for future delivery.
- Accumulating work that could be done during off-peak hours.

Messages are not a good fit for the following use cases:

- Broadcasting of events to multiple subscribers.
- Ordered delivery.
- Real-time delivery.

Creating a message table

The current implementation requires a fixed schema. This will be made more flexible in the future. There will also be a custom DDL syntax. For now, a message table must be created like this:

```
create table my_message(
  time_scheduled bigint,
  id bigint,
  time_next bigint,
  epoch bigint,
```

```
  time_created bigint,
  time_acked bigint,
  message varchar(128),
  primary key(time_scheduled, id),
  unique index id_idx(id),
  index next_idx(time_next, epoch)
) comment
    'vitess_message,vt_ack_wait=30,vt_purge_after=86400,vt_batch_size=10,vt_cache_size=10000,vt_pol
```

The application-related columns are as follows:

- `id`: can be any type. Must be unique.
- `message`: can be any type.
- `time_scheduled`: must be a bigint. It will be used to store unix time in nanoseconds. If unspecified, the `Now` value is inserted.

The above indexes are recommended for optimum performance. However, some variation can be allowed to achieve different performance trade-offs.

The comment section specifies additional configuration parameters. The fields are as follows:

- `vitess_message`: Indicates that this is a message table.
- `vt_ack_wait=30`: Wait for 30s for the first message ack. If one is not received, resend.
- `vt_purge_after=86400`: Purge acked messages that are older than 86400 seconds (1 day).
- `vt_batch_size=10`: Send up to 10 messages per RPC packet.
- `vt_cache_size=10000`: Store up to 10000 messages in the cache. If the demand is higher, the rest of the items will have to wait for the next poller cycle.
- `vt_poller_interval=30`: Poll every 30s for messages that are due to be sent.

If any of the above fields are missing, vitess will fail to load the table. No operation will be allowed on a table that has failed to load.

Enqueuing messages

The application can enqueue messages using an insert statement:

```
insert into my_message(id, message) values(1, 'hello world')
```

These inserts can be part of a regular transaction. Multiple messages can be inserted to different tables. Avoid accumulating too many big messages within a transaction as it consumes memory on the VTTablet side. At the time of commit, memory permitting, all messages are instantly enqueued to be sent.

Messages can also be created to be sent in the future:

```
insert into my_message(id, message, time_scheduled) values(1, 'hello world', :future_time)
```

`future_time` must be the unix time expressed in nanoseconds.

Receiving messages

Processes can subscribe to receive messages by sending a `MessageStream` request to VTGate. If there are multiple subscribers, the messages will be delivered in a round-robin fashion. Note that this is not a broadcast; Each message will be sent to at most one subscriber.

The format for messages is the same as a vitess `Result`. This means that standard database tools that understand query results can also be message recipients. Currently, there is no SQL format for subscribing to messages, but one will be provided soon.

Subsetting It's possible that you may want to subscribe to specific shards or groups of shards while requesting messages. This is useful for partitioning or load balancing. The `MessageStream` API allows you to specify these constraints. The request parameters are as follows:

- `Name`: Name of the message table.
- `Keyspace`: Keyspace where the message table is present.
- `Shard`: For unsharded keyspaces, this is usually "0". However, an empty shard will also work. For sharded keyspaces, a specific shard name can be specified.
- `KeyRange`: If the keyspace is sharded, streaming will be performed only from the shards that match the range. This must be an exact match.

Acknowledging messages

A received (or processed) message can be acknowledged using the `MessageAck` API call. This call accepts the following parameters:

- `Name`: Name of the message table.
- `Keyspace`: Keyspace where the message table is present. This field can be empty if the table name is unique across all keyspaces.
- `Ids`: The list of ids that need to be acked.

Once a message is successfully acked, it will never be resent.

Exponential backoff

A message that was successfully sent will wait for the specified ack wait time. If no ack is received by then, it will be resent. The next attempt will be 2x the previous wait, and this delay is doubled for every attempt.

Purging

Messages that have been successfully acked will be deleted after their age exceeds the time period specified by `vt_purge_after`.

Advanced usage

The `MessageAck` functionality is currently an API call and cannot be used inside a transaction. However, you can ack messages using a regular DML. It should look like this:

```
update my_message set time_acked = :time_acked, time_next = null where id in ::ids and
   time_acked is null
```

You can manually change the schedule of existing messages with a statement like this:

```
update my_message set time_next = :time_next, epoch = :epoch where id in ::ids and
   time_acked is null
```

This comes in handy if a bunch of messages had chronic failures and got postponed to the distant future. If the root cause of the problem was fixed, the application could reschedule them to be delivered immediately. You can also optionally change the epoch. Lower epoch values increase the priority of the message and the back-off is less aggressive.

You can also view messages using regular `select` queries.

Undocumented features

These are features that were previously known limitations, but have since been supported and are awaiting further documentation.

- Flexible columns: Allow any number of application defined columns to be in the message table.
- No ACL check for receivers: To be added.
- Monitoring support: To be added.
- Dropped tables: The message engine does not currently detect dropped tables.

Known limitations

The message feature is currently in alpha, and can be improved. Here is the list of possible limitations/improvements:

- Proactive scheduling: Upcoming messages can be proactively scheduled for timely delivery instead of waiting for the next polling cycle.
- Changed properties: Although the engine detects new message tables, it does not refresh properties of an existing table.
- A `SELECT` style syntax for subscribing to messages.
- No rate limiting.
- Usage of partitions for efficient purging.

Monitoring

Current state of monitoring

There are currently three main ways that a Vitess cluster can be monitored. Depending on your needs, you can use any of the following methods:

1. Vitess status pages

The status HTML pages of various Vitess components can be accessed by pointing your browser to `http://<host>:<port>/debug/statu`
The status pages will often display some basic, but useful, information for monitoring. For example, the status page of a vttablet will show the QPS graph for the past few minutes.

Viewing a status page can be useful since it works out of the box, but it only provides very basic monitoring capabilities.

2. Pull-based metrics system

Vitess uses Go's expvar package to expose various metrics, with the expectation that a user can configure a pull-based metrics system to ingest those metrics. Metrics are published to `http://<host>:<port>/debug/vars` as JSON key-value pairs, which should be easy for any metrics system to parse.

Scraping Vitess variables is a good way to integrate Vitess into an existing monitoring system, and is useful for building up detailed monitoring dashboards. It is also the officially supported way for monitoring Vitess.

3. Push-based metrics system

Vitess also includes support for push-based metrics systems via plug-ins. Each Vitess component would need to be run with the `--emit_stats` flag.

By default, the stats_emit_period is 60s, so each component will push stats to the selected backend every minute. This is configurable via the `--stats_emit_period` flag.

Vitess has preliminary plug-ins to support OpenTSDB as a push-based metrics backend.

It should be fairly straightforward to write your own plug-in, if you want to support a different backend. The plug-in package simply needs to implement the `PushBackend` interface of the `stats` package. For an example, you can see the OpenTSDB plugin.

Once you've written the backend plug-in, you also need to register the plug-in from within all the relevant Vitess binaries. An example of how to do this can be seen in this pull request.

You can then specify that Vitess should publish stats to the backend that you're targeting by using the `--stats_backend` flag.

Connecting Vitess to a push-based metrics system can be useful if you're already running a push-based system that you would like to integrate into.

Monitoring with Kubernetes

The existing methods for integrating metrics are not supported in a Kubernetes environment by the Vitess team yet, but are on the roadmap for the future. However, it should be possible to get the Prometheus backend working with Kubernetes.

In the meantime, if you run into issues or have questions, please post on our Slack.

Replication

{{< warning >}} Vitess requires the use of Row-Based Replication with GTIDs enabled. In addition, Vitess only supports the default `binlog_row_image` of `FULL`. {{< /warning >}}

Vitess makes use of MySQL Replication for both high availability and to receive a feed of changes to database tables. This feed is then used in features such as VReplication, and to identify schema changes so that caches can be updated.

Semi-Sync

Vitess strongly recommends the use of Semi-synchronous replication for High Availability. When enabled in Vitess, *semi-sync* has the following characteristics:

- The master will only accept writes if it has at least one replica connected, and configured correctly to send semi-sync ACKs. Vitess configures the semi-sync timeout to essentially an unlimited number so that it will never fallback to asyncronous replication. This is important to prevent split brain (or alternate futures) in case of a network partition. If we can verify all replicas have stopped replicating, we know the old master is not accepting writes, even if we are unable to contact the old master itself.
- Tablets of type rdonly will not send semi-sync ACKs. This is intentional because rdonly tablets are not eligible to be promoted to master, so Vitess avoids the case where a rdonly tablet is the single best candidate for election at the time of master failure.

These behaviors combine to give you the property that, in case of master failure, there is at least one other replica that has every transaction that was ever reported to clients as having completed. You can then (manually, or using Orchestrator to pick the replica that is farthest ahead in GTID position and promote that to be the new master.

Thus, you can survive sudden master failure without losing any transactions that were reported to clients as completed. In MySQL 5.7+, this guarantee is strengthened slightly to preventing loss of any transactions that were ever **committed** on the original master, eliminating so-called phantom reads.

On the other hand these behaviors also give a requirement that each shard must have at least 2 tablets with type *replica* (with addition of the master that can be demoted to type *replica* this gives a minimum of 3 tablets with initial type *replica*). This will allow for the master to have a semi-sync acker when one of the replica tablets is down for any reason (for a version update, machine reboot, schema swap or anything else).

With regard to replication lag, note that this does **not** guarantee there is always at least one replica from which queries will always return up-to-date results. Semi-sync guarantees that at least one replica has the transaction in its relay log, but it has not necessarily been applied yet. The only way Vitess guarantees a fully up-to-date read is to send the request to the master.

Database Schema Considerations

- Row-based replication requires that replicas have the same schema as the master, and corruption will likely occur if the column order does not match. Earlier versions of Vitess which used Statement-Based replication recommended applying schema changes on replicas first, and then swapping their role to master. This method is no longer recommended and a tool such as `gh-ost` or `pt-online-schema-change` should be used instead.
- Using a column of type `FLOAT` or `DOUBLE` as part of a Primary Key is not supported. This limitation is because Vitess may try to execute a query for a value (for example 2.2) which MySQL will return zero results, even when the approximate value is present.
- It is not recommended to change the schema at the same time a resharding operation is being performed. This limitation exists because interpreting RBR events requires accurate knowledge of the table's schema, and Vitess does not always correctly handle the case that the schema has changed.

Point In Time Recovery

Point in Time Recovery

Supported Databases

- MySQL 5.7

Believed to work, but untested

- MySQL 8.0

Introduction The Point in Time Recovery feature in Vitess enables recovery of data to a specific point time (timestamp). There can be multiple recovery requests active at the same time. It is possible to recover across sharding actions, i.e. you can recover to a time when there were two shards even though at present there are four.

Point in Time Recovery leverages two Vitess features:

1. The use of `SNAPSHOT` keyspaces for recovery of the last backup before a requested specific timestamp to restore to.
2. Integration with a binlog server to allow vttablet to apply binary logs from the recovered backup up to the specified timestamp.

Use cases

- Accidental deletion of data, e.g. dropping a table by mistake, running an UPDATE or DELETE with an incorrect WHERE clause, etc.
- Corruption of data due to application bugs.
- Corruption of data due to MySQL bugs or underlying hardware (e.g. storage) problems.

Preconditions

- There should be a Vitess backup taken before the desired point in time.
- There should be continuous binlogs available from the backup time to the desired point in time.
- This feature is tested using Ripple as the binlog server. However, it should be possible to use a MySQL instance as source for the binlogs as well.

Example usage To use this feature, you need a usable backup of Vitess data and continuous binlogs.

Here is how you can create a backup.

```
$ vtctlclient -server <vtctld_host>:<vtctld_port> Backup zone1-101
```

Here `zone1-101` is the tablet alias of a replica tablet in the shard that you want to back up. Note that you can also use `vtctlclient BackupShard` to just specify a keyspace and shard, and have Vitess choose the tablet to run the backup for you, instead of having to specify the tablet alias explicitly.

To maintain continuous binlogs, you need to have a binlog server pointing to the master (or a replica, assuming that the replica is also maintaining its own binlogs, which is the default Vitess configuration). You can use Ripple as a binlog server, although there are other options; and you could use an existing MySQL server as well.

If you use Ripple, you will need to configure it yourself, and ensure you take care of the following:

- You should have a highly available binlog server setup. If the binlog server goes down, you need to ensure that it is back up and able to synchronize the MySQL binary logs from its upstream MySQL server before the upstream server deletes the current binlog. If you do not do this, you will end up with gaps in your binlogs, which could make restoring to a specific point in time impossible. Make sure that you setup your operational and monitoring procedures accordingly.
- The binlog files should be safely kept at some reliable and recoverable location (e.g. AWS S3, remote file storage).

Once the above is done, you can proceed with doing a recovery.

Recovery Procedure First, you need to create a `SNAPSHOT` keyspace with a `base_keyspace` pointing to the original keyspace you are recovering the backup of. This can be done by using following:

```
$ vtctlclient -server <vtctld_host>:<vtctld_port> CreateKeyspace -keyspace_type=SNAPSHOT
   -base_keyspace=originalks -snapshot_time=2020-07-17T18:25:20Z restoreks
```

Here:

- `originalks` is the base keyspace, i.e. the keyspace we took a backup of, and are trying to recover.
- `snapshot_time` is the timestamp of the point in time to we want to recover to. Note the use of the `Z` in the timestamp, indicating it is expressed in UTC.
- `restoreks` is the name of recovery keyspace, i.e. the keyspace to which we are restoring our backup.

Next, you can launch the vttablet, which as part of vttablet's normal initialization procedure will look for a backup to restore. It will detect the meta-information you added on the keyspace topology node when creating the keyspace above. It will then use that information to restore the last backup earlier than the timestamp provided for the specific shard the vttablet is in.

Here are the command line arguments vttablet uses in this process. You may already be using some of these as part of your normal vttablet initialization parameters (e.g. if you are using the Vitess K8s operator):

- `-init_keyspace restoreks` - here `restoreks` is the recovery keyspace name which we created earlier
- `-init_db_name_override vt_originalks` - here `vt_originalks` is the name of the original underlying database for the keyspace that you backed up and want to restore. Usually, this takes the form of `vt_` prepended to the keyspace name. However, the original underlying database could also have been using an `-init_db_name_override` directive of its own, and this value should then be set to match that.
- `-init_shard 0` - here `0` is the shard name (or range) which we want to recover.
- `-binlog_host x.x.x.x` - hostname or IP address of binlog server.
- `-binlog_port XXXX` - TCP port of binlog server.
- `-binlog_user XXXX` - username to access binlog server.
- `-binlog_password YYYY` - password to access binlog server.
- `-pitr_gtid_lookup_timeout duration` - See below for details.

And then, depending on your backup storage implementation, you can use a variety of flags:

- `-backup_storage_implementation file` - for plain file backup type. If you use this option, you will also need to specify:
  - `-file_backup_storage_root` - with a path pointing to your backup storage location.
- `-backup_storage_implementation s3` - for backing up to S3. If you use this option, you may need additional flags like:
  - `-s3_backup_aws_region`
  - `-s3_backup_storage_bucket`
  - `-s3_backup_storage_root`
- There are more `-backup_storage_implementation` options like `gcs` and others.

You will also probably want to use other flags for backup and restore like:

- `-backup_engine_implementation xtrabackup` - Use Percona Xtrabackup to take online backups. Without this flag, the mysql instance on the replica being backed up will be shut down during the backup.
- `-backup_storage_compress true` - gzip compress the backup (default is true).

You need to be consistent in your use of these flags for backup and restore.

Once the restore of the last backup earlier than the `snapshot_time` timestamp is completed, the vttablet proceeds to use the `binlog_*` parameters to connect to the binlog server and then apply all binlog events from the time of the backup until the timestamp provided.

Since the last backup for each shard making up the keyspace could be taken at different points in time, the amount of time that it takes to apply these events may differ between restores of different shards in the keyspace.

Note that to restore to the specified `snapshot_time` timestamp, vttablet needs to find the GTID corresponding to the last event before this timestamp from the binlog server. This is an expensive operation and may take some time. By default the timeout for this operation is one minute (1m). This can be changed by setting the vttablet `-pitr_gtid_lookup_timeout` flag.

VTGate will automatically exclude tablets belonging to snapshot keyspaces from query routing unless they are specifically addressed using `USE restoreks` or by using queries of the form `SELECT ... FROM restoreks.table`.

The base keyspace's vschema will be copied over to the new snapshot keyspace as a default. If desired this can be overwritten by the user. Care needs to be taken to set `require_explicit_routing` to true when modifying a snapshot keyspace's vschema, or you will bypass the VTGate routing safety feature described above.

Schema Management

Using Vitess requires you to work with two different types of schemas:

1. The MySQL database schema. This is the schema of the individual MySQL instances.
2. The VSchema, which describes all the keyspaces and how they're sharded.

The workflow for the `VSchema` is as follows:

1. Apply the `VSchema` for each keyspace using the `ApplyVschema` command. This saves the VSchemas in the global topology service.
2. Execute `RebuildVSchemaGraph` for each cell (or all cells). This command propagates a denormalized version of the combined VSchema to all the specified cells. The main purpose for this propagation is to minimize the dependency of each cell from the global topology. The ability to push a change to only specific cells allows you to canary the change to make sure that it's good before deploying it everywhere.

This document describes the `vtctl` commands that you can use to review or update your schema in Vitess.

Note that this functionality is not recommended for long-running schema changes. It is recommended to use a tool such as `pt-online-schema-change` or `gh-ost` instead.

Reviewing your schema

This section describes the following vtctl commands, which let you look at the schema and validate its consistency across tablets or shards:

- GetSchema
- ValidateSchemaShard
- ValidateSchemaKeyspace
- GetVSchema
- GetSrvVSchema

GetSchema The GetSchema command displays the full schema for a tablet or a subset of the tablet's tables. When you call `GetSchema`, you specify the tablet alias that uniquely identifies the tablet. The `<tablet alias>` argument value has the format `<cell name>-<uid>`.

Note: You can use the `vtctl ListAllTablets` command to retrieve a list of tablets in a cell and their unique IDs.

The following example retrieves the schema for the tablet with the unique ID test-000000100:

```
GetSchema test-000000100
```

ValidateSchemaShard The `ValidateSchemaShard` command confirms that for a given keyspace, all of the replica tablets in a specified shard have the same schema as the master tablet in that shard. When you call `ValidateSchemaShard`, you specify both the keyspace and the shard that you are validating.

The following command confirms that the master and replica tablets in shard `0` all have the same schema for the `user` keyspace:

```
ValidateSchemaShard user/0
```

ValidateSchemaKeyspace The `ValidateSchemaKeyspace` command confirms that all of the tablets in a given keyspace have the the same schema as the master tablet on shard `0` in that keyspace. Thus, whereas the `ValidateSchemaShard` command confirms the consistency of the schema on tablets within a shard for a given keyspace, `ValidateSchemaKeyspace` confirms the consistency across all tablets in all shards for that keyspace.

The following command confirms that all tablets in all shards have the same schema as the master tablet in shard 0 for the user keyspace:

```
ValidateSchemaKeyspace user
```

GetVSchema The `GetVSchema` command displays the global VSchema for the specified keyspace.

GetSrvVSchema The `GetSrvVSchema` command displays the combined VSchema for a given cell.

Changing your schema

This section describes the following commands:

- ApplySchema
- ApplyVSchema
- RebuildVSchemaGraph

ApplySchema Vitess' schema modification functionality is designed the following goals in mind:

- Enable simple updates that propagate to your entire fleet of servers.
- Require minimal human interaction.
- Minimize errors by testing changes against a temporary database.
- Guarantee very little downtime (or no downtime) for most schema updates.
- Do not store permanent schema data in the topology service.

Note that, at this time, Vitess only supports data definition statements that create, modify, or delete database tables. For instance, `ApplySchema` does not affect stored procedures or grants.

The ApplySchema command applies a schema change to the specified keyspace on every master tablet, running in parallel on all shards. Changes are then propagated to replicas. The command format is: `ApplySchema {-sql=<sql> || -sql_file=<filename>} <keyspace>`

When the `ApplySchema` action actually applies a schema change to the specified keyspace, it performs the following steps:

1. It finds shards that belong to the keyspace, including newly added shards if a resharding event has taken place.
2. It validates the SQL syntax and determines the impact of the schema change. If the scope of the change is too large, Vitess rejects it. See the permitted schema changes section for more detail.
3. It employs a pre-flight check to ensure that a schema update will succeed before the change is actually applied to the live database. In this stage, Vitess copies the current schema into a temporary database, applies the change there to validate it, and retrieves the resulting schema. By doing so, Vitess verifies that the change succeeds without actually touching live database tables.
4. It applies the SQL command on the master tablet in each shard.

The following sample command applies the SQL in the **user_table.sql** file to the **user** keyspace:

```
ApplySchema -sql_file=user_table.sql user
```

Permitted schema changes The `ApplySchema` command supports a limited set of DDL statements. In addition, Vitess rejects some schema changes because large changes can slow replication and may reduce the availability of your overall system.

The following list identifies types of DDL statements that Vitess supports:

- `CREATE TABLE`
- `CREATE INDEX`
- `CREATE VIEW`
- `ALTER TABLE`
- `ALTER VIEW`
- `RENAME TABLE`
- `DROP TABLE`
- `DROP INDEX`
- `DROP VIEW`

In addition, Vitess applies the following rules when assessing the impact of a potential change:

- `DROP` statements are always allowed, regardless of the table's size.
- `ALTER` statements are only allowed if the table on the shard's master tablet has 100,000 rows or less.
- For all other statements, the table on the shard's master tablet must have 2 million rows or less.

If a schema change gets rejected because it affects too many rows, you can specify the flag `-allow_long_unavailability` to tell `ApplySchema` to skip this check. However, we do not recommend this. Instead, you should apply large schema changes by using an external tool such as `gh-ost` or `pt-online-schema-change`.

ApplyVSchema The `ApplyVSchema` command applies the specified VSchema to the keyspace. The VSchema can be specified as a string or in a file.

RebuildVSchemaGraph The `RebuildVSchemaGraph` command propagates the global VSchema to a specific cell or the list of specified cells.

Schema Routing Rules

The Vitess routing rules feature is a powerful mechanism for directing traffic to the right keyspaces, shards or tablet types. It fulfils the following use cases:

- **Routing traffic during resharding**: During resharding, you can specify rules that decide where to send reads and writes. For example, you can move traffic from the source shard to the destination shards, but only for the `rdonly` or `replica` types. This gives you the option to try out the new shards and make sure they will work as intended before committing to move the rest of the traffic.
- **Table equivalence**: The new VReplication feature allows you to materialize tables in different keyspaces. In this situation, you can specify that two tables are 'equivalent'. This will allow VTGate to use the best possible plan depending on the input query.

ApplyRoutingRules

You can use the vtctlclient command to apply routing rules:

```
ApplyRoutingRules {-rules=<rules> || -rules_file=<rules_json_file>} [-cells=c1,c2,...]
   [-skip_rebuild] [-dry-run]
```

Syntax

Resharding Routing rules can be specified using JSON format. Here's an example:

```
{"rules": [
  {
    "from_table": "t@rdonly",
    "to_tables": ["target.t"]
  }, {
    "from_table": "target.t",
    "to_tables": ["source.t"]
  }, {
    "from_table": "t",
    "to_tables": ["source.t"]
  }
]}
```

The above JSON specifies the following rules:

- If you sent a query accessing `t` for an `rdonly` instance, then it would be sent to table `t` in the `target` keyspace.
- If you sent a query accessing `target.t` for anything other than `rdonly`, it would be sent `t` in the `source` keyspace.
- If you sent a query accessing `t` without any qualification, it would be sent to `t` in the `source` keyspace.

These rules are an example of how they can be used to shift traffic for a table during a vertical resharding process. In this case, the assumption is that we are moving `t` from `source` to `target`, and so far, we've shifted traffic for just the `rdonly` tablet types.

By updating these rules, you can eventually move all traffic to `target.t`

The rules are applied only once. The resulting targets need to specify fully qualified table names.

Table equivalence The routing rules allow you to specify table equivalence. Here's an example:

```
{"rules": [
  {
    "from_table": "product",
    "to_tables": ["lookup.product", "user.uproduct"]
  }
]}
```

In the above case, we are declaring that the `product` table is present in both `lookup` and `user`. If a query is issued using the unqualified `product` table, then VTGate will consider sending the query to both `lookup.product` as well as `user.uproduct` (note the name change).

For example, if `user` was a sharded keyspace, and the query joined a `user` table with `product`, then vtgate will know that it's better to send the query to the `user` keyspace instead of `lookup`.

Typically, table equivalence makes sense when a view table is materialized from a source table using VReplication.

Orthogonality The tablet type targeting and table equivalence features are orthogonal to each other and can be combined. Although there's no immediate use case for this, it's a possibility we can consider if the use case arises.

Schema Tracking

{{< warning >}} This feature is experimental. {{< /warning >}}

VTGate does not natively track table schema. Users are allowed to provide an authoritative list of columns through a VSchema which is then used to enhance query planning. If no such list is provided, a set of queries will not be supported by VTGate due to a lack of information on the underlying tables/columns.

The schema tracking functionality alleviates this issue and enable VTGate to plan more queries. When using schema tracking, VTGate keeps an authoritative list of columns on all tables. The following query set can be planned:

- `SELECT *` cross-shard queries that need evaluation at the VTGate level.
- Queries that are not able to resolve columns dependencies. For instance: queries with no table qualifier in the projection/-filter list.

VTGate

Schema tracking is enabled in VTGate with the flag `-schema_change_signal`. When enabled, VTGate listens for schema changes from VTTablet.

VTTablet

Schema tracking is enabled in VTTablet with the flag `-queryserver-config-schema-change-signal`. When enabled, VTTablet sends schema changes to VTGate based on an interval that can be modified with the flag `-queryserver-config-schema-change-signal` (defaults to 5 seconds).

Sharding

description: Shard widely, shard often.

Sharding is a method of horizontally partitioning a database to store data across two or more database servers. This document explains how sharding works in Vitess and the types of sharding that Vitess supports.

Overview

A keyspace in Vitess can be sharded or unsharded. An unsharded keyspace maps directly to a MySQL database. If sharded, the rows of the keyspace are partitioned into different databases of identical schema.

For example, if an application's "user" keyspace is split into two shards, each shard contains records for approximately half of the application's users. Similarly, each user's information is stored in only one shard.

Note that sharding is orthogonal to (MySQL) replication. A Vitess shard typically contains one MySQL master and many MySQL replicas. The master handles write operations, while replicas handle read-only traffic, batch processing operations, and other tasks. Each MySQL instance within the shard should have the same data, excepting some replication lag.

Supported Operations Vitess supports the following types of sharding operations:

- **Horizontal sharding:** Splitting or merging shards in a sharded keyspace
- **Vertical sharding:** Moving tables from an unsharded keyspace to a different keyspace.

With these features, you can start with a single keyspace that contains all of your data (in multiple tables). As your database grows, you can move tables to different keyspaces (vertical split) and shard some or all of those keyspaces (horizontal split) without any real downtime for your application.

Sharding scheme

Vitess allows you to choose the type of sharding scheme by the choice of your Primary Vindex for the tables of a shard. Once you have chosen the Primary Vindex, you can choose the partitions depending on how the resulting keyspace IDs are distributed.

Vitess calculates the sharding key or keys for each query and then routes that query to the appropriate shards. For example, a query that updates information about a particular user might be directed to a single shard in the application's "user" keyspace. On the other hand, a query that retrieves information about several products might be directed to one or more shards in the application's "product" keyspace.

Key Ranges and Partitions Vitess uses key ranges to determine which shards should handle any particular query.

- A **key range** is a series of consecutive keyspace ID values. It has starting and ending values. A key falls inside the range if it is equal to or greater than the start value and strictly less than the end value.
- A **partition** represents a set of key ranges that covers the entire space.

When building the serving graph for a sharded keyspace, Vitess ensures that each shard is valid and that the shards collectively constitute a full partition. In each keyspace, one shard must have a key range with an empty start value and one shard, which could be the same shard, must have a key range with an empty end value.

- An empty start value represents the lowest value, and all values are greater than it.
- An empty end value represents a value larger than the highest possible value, and all values are strictly lower than it.

Vitess always converts sharding keys to a left-justified binary string for computing a shard. This left-justification makes the right-most zeroes insignificant and optional. Therefore, the value `0x80` is always the middle value for sharding keys. So, in a keyspace with two shards, sharding keys that have a binary value lower than 0x80 are assigned to one shard. Keys with a binary value equal to or higher than 0x80 are assigned to the other shard.

Several sample key ranges are shown below:

```
Start=[], End=[]: Full Key Range
Start=[], End=[0x80]: Lower half of the Key Range.
Start=[0x80], End=[]: Upper half of the Key Range.
Start=[0x40], End=[0x80]: Second quarter of the Key Range.
Start=[0xFF00], End=[0xFF80]: Second to last 1/512th of the Key Range.
```

Two key ranges are consecutive if the end value of one range equals the start value of the other range.

Shard Names A shard's name identifies the start and end of the shard's key range, printed in hexadecimal and separated by a hyphen. For instance, if a shard's key range is the array of bytes beginning with [0x80] and ending, noninclusively, with [0xc0], then the shard's name is `80-c0`.

Using this naming convention, the following four shards would be a valid full partition:

- -40
- 40-80
- 80-c0
- c0-

Shards do not need to handle the same size portion of the key space. For example, the following five shards would also be a valid full partition, possibly with a highly uneven distribution of keys.

- -80
- 80-c0
- c0-dc00
- dc00-dc80
- dc80-

Resharding

Resharding describes the process of updating the sharding scheme for a keyspace and dynamically reorganizing data to match the new scheme. During resharding, Vitess copies, verifies, and keeps data up-to-date on new shards while the existing shards continue to serve live read and write traffic. When you're ready to switch over, the migration occurs with only a few seconds of read-only downtime. During that time, existing data can be read, but new data cannot be written.

The table below lists the sharding (or resharding) processes that you would typically perform for different types of requirements:

Requirement	Action
Uniformly increase read capacity	Add replicas or split shards
Uniformly increase write capacity	Split shards
Reclaim overprovisioned resources	Merge shards and/or keyspaces
Increase geo-diversity	Add new cells and replicas
Cool a hot tablet	For read access, add replicas or split shards. For write access, split shards.

Additional Tools and Processes Vitess provides the following tools to help manage range-based shards:

- The vtctl command-line tool supports functions for managing keyspaces, shards, tablets, and more.
- Client APIs account for sharding operations.—

Table lifecycle

Vitess manages a table lifecycle flow, an abstraction and automation for a `DROP TABLE` operation.

Problems with DROP TABLE

Vitess inherits the same issues that MySQL has with `DROP TABLE`. Doing a direct `DROP TABLE my_table` in production can be a risky operation. In busy environments this can lead to a complete lockdown of the database for the duration of seconds, to minutes and more. This is typically less of a problem in Vitess than it might be in normal MySQL, if you are keeping your shard instances (and thus shard table instances) small, but could still be a problem.

There are two locking aspects to dropping tables:

- Purging dropped table's pages from the InnoDB buffer pool(s)
- Removing table's data file (`.ibd`) from the filesystem.

The exact locking behavior and duration can vary depending on various factors:

- Which filesystem is used
- Whether the MySQL adaptive hash index is used
- Whether you are attempting to hack around some of the MySQL `DROP TABLE` performance problems using hard links

It is common practice to avoid direct `DROP TABLE` statements and to follow a more elaborate table lifecycle.

Vitess table lifecycle

The lifecycle offered by Vitess consists of the following stages or some subset:

> *in use* -> hold -> purge -> evac -> drop -> *removed*

To understand the flow better, consider the following breakdown:

- *In use*: the table is serving traffic, like a normal table.
- `hold`: the table is renamed to some arbitrary new name. The application cannot see it, and considers it as gone. However, the table still exists, with all of its data intact. It is possible to reinstate it (e.g. in case we realize some application still requires it) by renaming it back to its original name.
- `purge`: the table is in the process of being purged (i.e. rows are being deleted). The purge process completes when the table is completely empty. At the end of the purge process the table no longer has any pages in the buffer pool(s). However, the purge process itself loads the table pages to cache in order to delete rows. Vitess purges the table a few rows at a time, and uses a throttling mechanism to reduce load. Vitess disables binary logging for the purge. The deletes are not written to the binary logs and are not replicated. This reduces load from disk IO, network, and replication lag. Data is not purged on the replicas. Experience shows that dropping a table populated with data on a replica has lower performance impact than on the primary, and the tradeoff is worthwhile.
- `evac`: a waiting period during which we expect normal production traffic to slowly evacuate the (now inactive) table's pages from the buffer pool. Vitess hard codes this period for `72` hours. The time is heuristic, there is no tracking of table pages in the buffer pool.
- `drop`: an actual `DROP TABLE` is imminent
- *removed*: table is dropped. When using InnoDB and `innodb_file_per_table` this means the `.ibd` data file backing the table is removed, and disk space is reclaimed.

Lifecycle subsets and configuration

Different environments and users have different requirements and workflows. For example:

- Some wish to immediately start purging the table, wait for pages to evacuate, then drop it.
- Some want to keep the table's data for a few days, then directly drop it.
- Some just wish to directly drop the table, they see no locking issues (e.g. smaller table).

Vitess supports all subsets via `-table_gc_lifecycle` flag to `vttablet`. The default is `"hold,purge,evac,drop"` (the complete cycle). Users may configure any subset, e.g. `"purge,drop"`, `"hold,drop"`, `"hold,evac,drop"` or even just `"drop"`.

Vitess will always work the steps in this order: `hold -> purge -> evac -> drop`. For example, setting `-table_gc_lifecycle "drop,hold"` still first *holds*, then *drops*

All subsets end with a `drop`, even if not explicitly mentioned. Thus, `"purge"` is interpreted as `"purge,drop"`.

Stateless flow by table name hints

Vitess does not track the state of the table lifecycle. The process is stateless thanks to an encoding scheme in the table names. Examples:

- The table `_vt_HOLD_6ace8bcef73211ea87e9f875a4d24e90_20210915120000` is held until `2021-09-15 12:00:00`. The data remains intact.
- The table `_vt_PURGE_6ace8bcef73211ea87e9f875a4d24e90_20210915123000` is at the state where it is being purged, or queued to be purged. Once it's fully purged (zero rows remain), it transitions to the next stage.
- The table `_vt_EVAC_6ace8bcef73211ea87e9f875a4d24e90_20210918093000` is held until `2021-09-18 09:30:00`
- The table `_vt_DROP_6ace8bcef73211ea87e9f875a4d24e90_20210921170000` is eligible to be dropped on `2021-09-21 17:00:00`

Automated lifecycle

Vitess internally uses the above table lifecycle for online, managed schema migrations. Online schema migration tools `gh-ost` and `pt-online-schema-change` create artifact tables or end with leftover tables: Vitess automatically collects those tables. The artifact or leftover tables are immediate moved to `purge` state. Depending on `-table_gc_lifecycle`, they may spend time in this state, getting purged, or immediately transitioned to the next state.

User-facing DROP TABLE lifecycle

When using an online `ddl_strategy`, a `DROP TABLE` is a managed schema migration. It is internally replaced by a `RENAME TABLE` statement, renaming it into a `HOLD` state (e.g. `_vt_HOLD_6ace8bcef73211ea87e9f875a4d24e90_20210915120000`). It will then participate in the table lifecycle mechanism. If `table_gc_lifecycle` does not include the `hold` state, the table proceeds to transition to next included state.

A multi-table `DROP TABLE` statement is converted to multiple single-table `DROP TABLE` statements, each to then convert to a `RENAME TABLE` statement.

Tablet throttler

VTTablet runs a cooperative throttling service. This service probes the shard's MySQL topology and observes health, measure by replication lag, or by another metric delivered by custom query, on servers. This throttler is derived from GitHub's freno.

Note: the Vitess documentation is transitioning from the term "Master" (with regard to MySQL replication) to "Primary". this document reflects this transition.

Why throttler: maintaining shard health via low replication lag

Vitess uses MySQL with asynchronous or semi-synchronous replication. In these modes, each shard has a primary instance that applies changes and logs them to the binary log. The replicas for that shard will get binary log entries from the primary, potentially acknowledge them (if semi-synchronous replication is enabled), and apply them. A running replica normally applies the entries as soon as possible, unless it is stopped or configured to delay. However, if the replica is busy, then it may not have the resources to apply events in a timely fashion, and can therefore start lagging. For example, if the replica is serving traffic, it may lack the necessary disk I/O or CPU to avoid lagging behind the primary.

Maintaining low replication lag is important in production for two reasons:

- A lagging replica may not be representative of the data on the primary. Reads from the replica reflect data that is not consistent with the data on the primary. This is noticeable on web services following read-after-write from the replica, and this can produce results not reflecting the write.
- An up-to-date replica makes for a good failover experience. If all replicas are lagging, then a failover process must choose between waiting for a replica to catch up or losing data.

Some common database operations include mass writes to the database, including the following:

- Online schema migrations duplicating entire tables
- Mass population of columns, such as populating the new column with derived values following an `ADD COLUMN` migration
- Purging of old data
- Purging of tables as part of safe table `DROP` operation

Other operations include mass reads from the database:

- An ETL reading content of entire tables
- VReplication scanning an entire keyspace data and binary logs

These operations can easily incur replication lag. However, these operations are typically not time-limited. It is possible to rate-limit them to reduce database load.

This is where a throttler becomes useful. A throttler can detect when replication lag is low, a cluster is healthy, and operations can proceed. It can also detect when replication lag is high and advise applications to hold the next operation.

Applications are expected to break down their tasks into small sub-tasks. For example, instead of deleting `1,000,000` rows, an application should only delete `50` at a time. Between these sub-tasks, the application should check in with the throttler.

The throttler is only intended for use with operations such as the above mass write/read cases. It should not be used for ongoing, normal OLTP queries.

Throttler overview

Each `vttablet` runs an internal throttler service, and provides API endpoints to the throttler. Each tablet, including the primary, measures its own "self" health, discussed later.

Cluster health: In addition, the primary tablet is responsible for the overall health of the cluster/shard:

- The throttler confirms it is still the primary tablet for its shard.
- Every `10sec`, the throttler uses the topology server to refresh the shard's tablets list.
- The throttler probes all `REPLICA` tablets for their replication lag. This is done by querying the `_vt.heartbeat` table.
  - The throttler begins in dormant probe mode. As long as no application or client is actually looking for metrics, it probes the servers at multi-second intervals.
  - When applications check for throttle advice, the throttler begins probing servers in subsecond intervals. It reverts to dormant probe mode if no requests are made in the duration of `1min`.
- The throttler aggregates the last probed values from all relevant tablets. This is *the cluster's metric* .

The cluster's metric is only as accurate as the following metrics:

- The probe interval
- The heartbeat injection interval
- The aggregation interval

The error margin equals approximately the sum of the above values, plus additional overhead. The defaults for these intervals are as follows:

- Probe interval: `100ms`
- Aggregation interval: `100ms`
- Heartbeat interval: `250ms`

The user may override the heartbeat interval by sending `-heartbeat_interval` flag to `vttablet`.

Thus, the aggregated interval can be off, by default, by some `500ms`. This makes it inaccurate for evaluations that require high resolution lag evaluation. This resolution is sufficient for throttling purposes.

Self health Each tablet runs a local health check against its backend database, again in the form of evaluating replication lag from `_vt.heartbeat`. Intervals are identical to the cluster health interval illustrated above.

Response codes The throttler allows clients and applications to `check` for throttle advice. The check is an `HTTP` request, `HEAD` method, or `GET` method. Throttler returns one of the following HTTP response codes as an answer:

- `200` (OK): The application may write to the data store. This is the desired response.
- `404` (Not Found): The check contains an unknown metric name. This can take place immediately upon startup or immediately after failover, and should resolve within 10 seconds.
- `417` (Expectation Failed): The requesting application is explicitly forbidden to write. The throttler does not implement this at this time.
- `429` (Too Many Requests): Do not write. A normal, expected state indicating there is replication lag. This is the hint for applications or clients to withhold writes.
- `500` (Internal Server Error): An internal error has occurred. Do not write.

Normally, apps will see either `200` or `429`. An app should only ever proceed to write to the database when it receives a `200` response code.

The throttler chooses the response by comparing the replication lag with a pre-defined *threshold*. If the lag is lower than the threshold, response can be `200` (OK). If the lag is higher than the threshold, the response would be `429` (Too Many Requests).

The throttler only collects and evaluates lag on a set of predefined tablet types. By default, this tablet type set is `REPLICA`. See Configuration.

When the throttler sees no relevant replicas in the shard, it allows writes by responding with `HTTP 200 OK`.

Custom metrics & queries

The default behavior is to measure replication lag and throttle based on that lag. Vitess allows the user to use custom metrics and thresholds for throttling.

Vitess only supports gauges for custom metrics: the user may define a query which returns a gauge value, an absolute metric by which Vitess can throttle. See #Configuration, below.

Configuration

- The throttler is currently **disabled** by default. Use the `vttablet` option `-enable-lag-throttler` to enable the throttler. When the throttler is disabled, it still serves `/throttler/check` and `/throttler/check-self` API endpoints, and responds with `HTTP 200 OK` to all requests. When the throttler is enabled, it implicitly also runs heartbeat injections.
- Use the `vttablet` flag `-throttle_threshold` to set a lag threshold value. The default threshold is `1sec` and is set upon tablet startup. For example, to set a half-second lag threshold, use the flag `-throttle_threshold=0.5s`.
- To set the tablet types that the throttler queries for lag, use the `vttablet` flag `-throttle_tablet_types="replica,rdonly"`. The default tablet type is `replica`; this type is always implicitly included in the tablet types list. You may add any other tablet type. Any type not specified is ignored by the throttler.
- To override the default lag evaluation, and measure a different metric, use `-throttle_metrics_query`. The query must be either of these forms:
  - `SHOW GLOBAL STATUS LIKE '<metric>'`
  - `SHOW GLOBAL VARIABLES LIKE '<metric>'`
  - `SELECT <single-column> FROM ...`, expecting single column, single row result
- To override the throttle threshold, use `-throttle_metrics_threshold`. Floating point values are accepted.
- Use `-throttle_check_as_check_self` to implicitly reroute any `/throttler/check` call into `/throttler/check-self`. This makes sense when the user supplies a custom query, and where the user wishes to throttle writes to the cluster based on the primary tablet's health, rather than the overall health of the cluster.

An example for custom query & threshold setup, using the MySQL metrics `Threads_running` (number of threads actively executing a query at a given time) on the primary, might look like:

```
$ vttablet
  -throttle_metrics_query "show global status like 'threads_running'"
  -throttle_metrics_threshold 150
  -throttle_check_as_check_self
```

API & usage

Applications use these API endpoints:

- `/throttler/check`, for apps that wish to write mass amounts of data to a shard, and wish to maintain the overall health of the shard.
- `/throttler/check-self`, for apps that wish to perform some operation (e.g. a massive *read*) on a specific tablet and only wish to maintain the health of that tablet.
- Applications may indicate their identity via `?app=<name>` parameter.
- Applications may also declare themselves to be *low priority* via `?p=low` param. Managed online schema migrations (`gh-ost`, `pt-online-schema-change`) do so, as does the table purge process.

Examples:

- `gh-ost` uses this throttler endpoint: `/throttler/check?app=gh-ost&p=low`
- A data backfill application will identify as such, and use *normal* priority: `/throttler/check?app=my_backfill` (priority not indicated in URL therefore assumed to be *normal*)
- An app reading a massive amount of data directly from a replica tablet will use `/throttler/check-self?app=my_data_reader`

A `HEAD` request is sufficient. A `GET` request also provides a `JSON` output. For example:

- `{"StatusCode":200,"Value":0.207709,"Threshold":1,"Message":""}`
- `{"StatusCode":429,"Value":3.494452,"Threshold":1,"Message":"Threshold exceeded"}`
- `{"StatusCode":404,"Value":0,"Threshold":0,"Message":"No such metric"}`

In the first two above examples we can see that the tablet is configured to throttle at `1sec`

Tablet also provides `/throttler/status` endpoint. This is useful for monitoring and management purposes.

Example: Healthy primary tablet

The following command gets throttler status on a primary tablet hosted on `tablet1`, serving on port `15100`.

```
$ curl -s http://tablet1:15100/throttler/status | jq .
```

This API call returns the following JSON object:

```
{
  "Keyspace": "commerce",
  "Shard": "80-c0",
  "IsLeader": true,
  "IsOpen": true,
  "IsDormant": false,
  "AggregatedMetrics": {
    "mysql/self": {
      "Value": 0.749837
    },
    "mysql/shard": {
      "Value": 0.749887
    }
```

```
  },
  "MetricsHealth": {
    "mysql/self": {
      "LastHealthyAt": "2021-01-24T19:03:19.141933727+02:00",
      "SecondsSinceLastHealthy": 0
    },
    "mysql/shard": {
      "LastHealthyAt": "2021-01-24T19:03:19.141974429+02:00",
      "SecondsSinceLastHealthy": 0
    }
  }
}
```

The primary tablet serves two types of metrics:

- `mysql/shard`: an aggregated lag on relevant replicas in this shard. This is the metric to check when writing massive amounts of data to this server.
- `mysql/self`: the health of the specific primary MySQL server backed by this tablet.

`"IsLeader": true` indicates this tablet is active, is the `primary`, and is running probes. `"IsDormant": false,` means that an application has recently issued a `check`, and the throttler is probing for lag at high frequency.

Example: replica tablet

The following command gets throttler status on a replica tablet hosted on `tablet2`, serving on port `15100`.

```
$ curl -s http://tablet2:15100/throttler/status | jq .
```

This API call returns the following JSON object:

```
{
  "Keyspace": "commerce",
  "Shard": "80-c0",
  "IsLeader": false,
  "IsOpen": true,
  "IsDormant": false,
  "AggregatedMetrics": {
    "mysql/self": {
      "Value": 0.346409
    }
  },
  "MetricsHealth": {
    "mysql/self": {
      "LastHealthyAt": "2021-01-24T19:04:25.038290475+02:00",
      "SecondsSinceLastHealthy": 0
    }
  }
}
```

The replica tablet only presents `mysql/self` metric (measurement of its own backend MySQL's lag). It does not serve checks for the shard in general.

Resources

- freno project page
- Mitigating replication lag and reducing read load with freno, a GitHub Engineering blog post

Topology Service

This document describes the Topology Service, a key part of the Vitess architecture. This service is exposed to all Vitess processes, and is used to store small pieces of configuration data about the Vitess cluster, and provide cluster-wide locks. It also supports watches, and master election.

Vitess uses a plugin implementation to support multiple backend technologies for the Topology Service (etcd, ZooKeeper, Consul). Concretely, the Topology Service handles two functions: it is both a distributed lock manager and a repository for topology metadata. In earlier versions of Vitess, the Topology Serice was also referred to as the Lock Service.

Requirements and usage

The Topology Service is used to store information about the Keyspaces, the Shards, the Tablets, the Replication Graph, and the Serving Graph. We store small data structures (a few hundred bytes) per object.

The main contract for the Topology Service is to be very highly available and consistent. It is understood it will come at a higher latency cost and very low throughput.

We never use the Topology Service as an RPC or queuing mechanism or as a storage system for logs. We never depend on the Topology Service being responsive and fast to serve every query.

The Topology Service must also support a Watch interface, to signal when certain conditions occur on a node. This is used, for instance, to know when the Keyspace topology changes (e.g. for resharding).

Global vs Local We differentiate two instances of the Topology Service: the Global instance, and the per-cell Local instance:

- The Global instance is used to store global data about the topology that doesn't change very often, e.g. information about Keyspaces and Shards. The data is independent of individual instances and cells, and needs to survive a cell going down entirely.
- There is one Local instance per cell, that contains cell-specific information, and also rolled-up data from the Global + Local cell to make it easier for clients to find the data. The Vitess local processes should not use the Global topology instance, but instead the rolled-up data in the Local topology server as much as possible.

The Global instance can go down for a while and not impact the local cells (an exception to that is if a reparent needs to be processed, it might not work). If a Local instance goes down, it only affects the local tablets in that instance (and then the cell is usually in bad shape, and should not be used).

Vitess will not use the global or local topology service as part of serving individual queries. The Topology Service is only used to get the topology information at startup and in the background.

Recovery If a Local Topology Service dies and is not recoverable, it can be wiped out. All the tablets in that cell then need to be restarted so they re-initialize their topology records (but they won't lose any MySQL data).

If the Global Topology Service dies and is not recoverable, this is more of a problem. All the Keyspace / Shard objects have to be recreated or be restored. Then the cells should recover.

Global data

This section describes the data structures stored in the Global instance of the topology service.

Keyspace The Keyspace object contains various information, mostly about sharding: how is this Keyspace sharded, what is the name of the sharding key column, is this Keyspace serving data yet, how to split incoming queries, ...

An entire Keyspace can be locked. We use this during resharding for instance, when we change which Shard is serving what inside a Keyspace. That way we guarantee only one operation changes the Keyspace data concurrently.

Shard A Shard contains a subset of the data for a Keyspace. The Shard record in the Global topology service contains:

- the Master tablet alias for this shard (that has the MySQL master).
- the sharding key range covered by this Shard inside the Keyspace.
- the tablet types this Shard is serving (master, replica, batch, …), per cell if necessary.
- if using filtered replication, the source shards this shard is replicating from.
- the list of cells that have tablets in this shard.
- shard-global tablet controls, like blacklisted tables no tablet should serve in this shard.

A Shard can be locked. We use this during operations that affect either the Shard record, or multiple tablets within a Shard (like reparenting), so multiple tasks cannot concurrently alter the data.

VSchema data The VSchema data contains sharding and routing information for the VTGate V3 API.

Local data

This section describes the data structures stored in the Local instance (per cell) of the topology service.

Tablets The Tablet record has a lot of information about each vttablet process making up each tablet (along with the MySQL process):

- the Tablet Alias (cell+unique id) that uniquely identifies the Tablet.
- the Hostname, IP address and port map of the Tablet.
- the current Tablet type (master, replica, batch, spare, …).
- which Keyspace / Shard the tablet is part of.
- the sharding Key Range served by this Tablet.
- user-specified tag map (e.g. to store per-installation data).

A Tablet record is created before a tablet can be running (either by `vtctl InitTablet` or by passing the `init_*` parameters to the vttablet process). The only way a Tablet record will be updated is one of:

- The vttablet process itself owns the record while it is running, and can change it.
- At init time, before the tablet starts.
- After shutdown, when the tablet gets deleted.
- If a tablet becomes unresponsive, it may be forced to spare to make it unhealthy when it restarts.

Replication graph The Replication Graph allows us to find Tablets in a given Cell / Keyspace / Shard. It used to contain information about which Tablet is replicating from which other Tablet, but that was too complicated to maintain. Now it is just a list of Tablets.

Serving graph The Serving Graph is what the clients use to find the per-cell topology of a Keyspace. It is a roll-up of global data (Keyspace + Shard). vtgates only open a small number of these objects and get all the information they need quickly.

SrvKeyspace It is the local representation of a Keyspace. It contains information on what shard to use for getting to the data (but not information about each individual shard):

- the partitions map is keyed by the tablet type (master, replica, batch, …) and the value is a list of shards to use for serving.
- it also contains the global Keyspace fields, copied for fast access.

It can be rebuilt by running `vtctl RebuildKeyspaceGraph <keyspace>`. It is automatically rebuilt when a tablet starts up in a cell and the SrvKeyspace for that cell / keyspace does not exist yet. It will also be changed during horizontal and vertical splits.

SrvVSchema It is the local roll-up for the VSchema. It contains the VSchema for all keyspaces in a single object.

It can be rebuilt by running `vtctl RebuildVSchemaGraph`. It is automatically rebuilt when using `vtctl ApplyVSchema` (unless prevented by flags).

Workflows involving the Topology Service

The Topology Service is involved in many Vitess workflows.

When a Tablet is initialized, we create the Tablet record, and add the Tablet to the Replication Graph. If it is the master for a Shard, we update the global Shard record as well.

Administration tools need to find the tablets for a given Keyspace / Shard. To retrieve this:

- first we get the list of Cells that have Tablets for the Shard (global topology Shard record has these)
- then we use the Replication Graph for that Cell / Keyspace / Shard to find all the tablets then we can read each tablet record.

When a Shard is reparented, we need to update the global Shard record with the new master alias.

Finding a tablet to serve the data is done in two stages:

- vtgate maintains a health check connection to all possible tablets, and they report which Keyspace / Shard / Tablet type they serve.
- vtgate also reads the SrvKeyspace object, to find out the shard map.

With these two pieces of information, vtgate can route the query to the right vttablet.

During resharding events, we also change the topology significantly. A horizontal split will change the global Shard records, and the local SrvKeyspace records. A vertical split will change the global Keyspace records, and the local SrvKeyspace records.

Exploring the data in a Topology Service

We store the proto3 serialized binary data for each object.

We use the following paths for the data, in all implementations:

Global Cell:

- CellInfo path: `cells/<cell name>/CellInfo`
- Keyspace: `keyspaces/<keyspace>/Keyspace`
- Shard: `keyspaces/<keyspace>/shards/<shard>/Shard`
- VSchema: `keyspaces/<keyspace>/VSchema`

Local Cell:

- Tablet: `tablets/<cell>-<uid>/Tablet`
- Replication Graph: `keyspaces/<keyspace>/shards/<shard>/ShardReplication`
- SrvKeyspace: `keyspaces/<keyspace>/SrvKeyspace`
- SrvVSchema: `SvrVSchema`

The `vtctl TopoCat` utility can decode these files when using the `-decode_proto` option:

```
TOPOLOGY="-topo_implementation zk2 -topo_global_server_address
   global_server1,global_server2 -topo_global_root /vitess/global"

$ vtctl $TOPOLOGY TopoCat -decode_proto -long /keyspaces/*/Keyspace
path=/keyspaces/ks1/Keyspace version=53
sharding_column_name: "col1"
path=/keyspaces/ks2/Keyspace version=55
sharding_column_name: "col2"
```

The `vtctld` web tool also contains a topology browser (use the `Topology` tab on the left side). It will display the various proto files, decoded.

Implementations

The Topology Service interfaces are defined in our code in `go/vt/topo/`, specific implementations are in `go/vt/topo/<name>`, and we also have a set of unit tests for it in `go/vt/topo/test`.

This part describes the implementations we have, and their specific behavior.

If starting from scratch, please use the `zk2`, `etcd2` or `consul` implementations. We deprecated the old `zookeeper` and `etcd` implementations. See the migration section below if you want to migrate.

Zookeeper `zk2` implementation This is the current implementation when using Zookeeper. (The old `zookeeper` implementation is deprecated).

The global cell typically has around 5 servers, distributed one in each cell. The local cells typically have 3 or 5 servers, in different server racks / sub-networks for higher resilience. For our integration tests, we use a single ZK server that serves both global and local cells.

We provide the `zk` utility for easy access to the topology data in Zookeeper. It can list, read and write files inside any Zoopeeker server. Just specify the `-server` parameter to point to the Zookeeper servers. Note the vtctld UI can also be used to see the contents of the topology data.

To configure a Zookeeper installation, let's start with the global cell service. It is described by the addresses of the servers (comma separated list), and by the root directory to put the Vitess data in. For instance, assuming we want to use servers `global_server1,global_server2` in path `/vitess/global`:

```
# The root directory in the global server will be created
# automatically, same as when running this command:
# zk -server global_server1,global_server2 touch -p /vitess/global

# Set the following flags to let Vitess use this global server:
# -topo_implementation zk2
# -topo_global_server_address global_server1,global_server2
# -topo_global_root /vitess/global
```

Then to add a cell whose local topology service `cell1_server1,cell1_server2` will store their data under the directory `/vitess/cell1`:

```
TOPOLOGY="-topo_implementation zk2 -topo_global_server_address
   global_server1,global_server2 -topo_global_root /vitess/global"

# Reference cell1 in the global topology service:
vtctl $TOPOLOGY AddCellInfo \
  -server_address cell1_server1,cell1_server2 \
  -root /vitess/cell1 \
  cell1
```

If only one cell is used, the same Zookeeper instance can be used for both global and local data. A local cell record still needs to be created, just use the same server address, and very importantly a *different* root directory.

Zookeeper Observers can also be used to limit the load on the global Zookeeper. They are configured by specifying the addresses of the observers in the server address, after a `|`, for instance: `global_server1:p1,global_server2:p2|observer1:po1,observer2:po2`.

Implementation details We use the following paths for Zookeeper specific data, in addition to the regular files:

- Locks sub-directory: `locks/` (for instance: `keyspaces/<keyspace>/Keyspace/locks/` for a keyspace)
- Master election path: `elections/<name>`

Both locks and master election are implemented using ephemeral, sequential files which are stored in their respective directory.

etcd `etcd2` implementation (new version of `etcd`) This topology service plugin is meant to use etcd clusters as storage backend for the topology data. This topology service supports version 3 and up of the etcd server.

This implementation is named `etcd2` because it supersedes our previous implementation `etcd`. Note that the storage format has been changed with the `etcd2` implementation, i.e. existing data created by the previous `etcd` implementation must be migrated manually (See migration section below).

To configure an `etcd2` installation, let's start with the global cell service. It is described by the addresses of the servers (comma separated list), and by the root directory to put the Vitess data in. For instance, assuming we want to use servers `http://global_server1,http://global_server2` in path `/vitess/global`:

```
# Set the following flags to let Vitess use this global server,
# and simplify the example below:
# -topo_implementation etcd2
# -topo_global_server_address http://global_server1,http://global_server2
# -topo_global_root /vitess/global
TOPOLOGY="-topo_implementation etcd2 -topo_global_server_address
   http://global_server1,http://global_server2 -topo_global_root /vitess/global
```

Then to add a cell whose local topology service `http://cell1_server1,http://cell1_server2` will store their data under the directory `/vitess/cell1`:

```
# Reference cell1 in the global topology service:
# (the TOPOLOGY variable is defined in the previous section)
vtctl $TOPOLOGY AddCellInfo \
  -server_address http://cell1_server1,http://cell1_server2 \
  -root /vitess/cell1 \
  cell1
```

If only one cell is used, the same etcd instances can be used for both global and local data. A local cell record still needs to be created, just use the same server address and, very importantly, a *different* root directory.

Implementation details For locks, we use a subdirectory named `locks` in the directory to lock, and an ephemeral file in that subdirectory (it is associated with a lease, whose TTL can be set with the `-topo_etcd_lease_duration` flag, defaults to 30 seconds). The ephemeral file with the lowest ModRevision has the lock, the others wait for files with older ModRevisions to disappear.

Master elections also use a subdirectory, named after the election Name, and use a similar method as the locks, with ephemeral files.

We store the proto3 binary data for each object (as the v3 API allows us to store binary data). Note that this means that if you want to interact with etcd using the `etcdctl` tool, you will have to tell it to use the v3 API, e.g.:

```
ETCDCTL_API=3 etcdctl get / --prefix --keys-only
```

Consul `consul` implementation This topology service plugin is meant to use Consul clusters as storage backend for the topology data.

To configure a `consul` installation, let's start with the global cell service. It is described by the address of a server, and by the root node path to put the Vitess data in (it cannot start with `/`). For instance, assuming we want to use servers `global_server:global_port` with node path `vitess/global`:

```
# Set the following flags to let Vitess use this global server,
# and simplify the example below:
# -topo_implementation consul
# -topo_global_server_address global_server:global_port
# -topo_global_root vitess/global
TOPOLOGY="-topo_implementation consul -topo_global_server_address global_server:global_port
   -topo_global_root vitess/global
```

Then to add a cell whose local topology service `cell1_server1:cell1_port` will store their data under the directory `vitess/cell1`:

```
# Reference cell1 in the global topology service:
# (the TOPOLOGY variable is defined in the previous section)
vtctl $TOPOLOGY AddCellInfo \
  -server_address cell1_server1:cell1_port \
  -root vitess/cell1 \
  cell1
```

If only one cell is used, the same consul instances can be used for both global and local data. A local cell record still needs to be created, just use the same server address and, very importantly, a *different* root node path.

Implementation details For locks, we use a file named `Lock` in the directory to lock, and the regular Consul Lock API.

Master elections use a single lock file (the Election path) and the regular Consul Lock API. The contents of the lock file is the ID of the current master.

Watches use the Consul long polling Get call. They cannot be interrupted, so we use a long poll whose duration is set by the `-topo_consul_watch_poll_duration` flag. Canceling a watch may have to wait until the end of a polling cycle with that duration before returning.

Running in only one cell

The topology service is meant to be distributed across multiple cells, and survive single cell outages. However, one common usage is to run a Vitess cluster in only one cell / region. This part explains how to do this, and later on upgrade to multiple cells / regions.

If running in a single cell, the same topology service can be used for both global and local data. A local cell record still needs to be created, just use the same server address and, very importantly, a *different* root node path.

In that case, just running 3 servers for topology service quorum is probably sufficient. For instance, 3 etcd servers. And use their address for the local cell as well. Let's use a short cell name, like `local`, as the local data in that topology service will later on be moved to a different topology service, which will have the real cell name.

Extending to more cells To then run in multiple cells, the current topology service needs to be split into a global instance and one local instance per cell. Whereas, the initial setup had 3 topology servers (used for global and local data), we recommend to run 5 global servers across all cells (for global topology data) and 3 local servers per cell (for per-cell topology data).

To migrate to such a setup, start by adding the 3 local servers in the second cell and run `vtctl AddCellinfo` as was done for the first cell. Tablets and vtgates can now be started in the second cell, and used normally.

vtgate can then be configured with a list of cells to watch for tablets using the `-cells_to_watch` command line parameter. It can then use all tablets in all cells to route traffic. Note this is necessary to access the master in another cell.

After the extension to two cells, the original topo service contains both the global topology data, and the first cell topology data. The more symmetrical configuration we are after would be to split that original service into two: a global one that only contains the global data (spread across both cells), and a local one to the original cells. To achieve that split:

- Start up a new local topology service in that original cell (3 more local servers in that cell).
- Pick a name for that cell, different from `local`.
- Use `vtctl AddCellInfo` to configure it.
- Make sure all vtgates can see that new local cell (again, using `-cells_to_watch`).
- Restart all vttablets to be in that new cell, instead of the `local` cell name used before.
- Use `vtctl RemoveKeyspaceCell` to remove all mentions of the `local` cell in all keyspaces.
- Use `vtctl RemoveCellInfo` to remove the global configurations for that `local` cell.
- Remove all remaining data in the global topology service that are in the old local server root.

After this split, the configuration is completely symmetrical:

- a global topology service, with servers in all cells. Only contains global topology data about Keyspaces, Shards and VSchema. Typically it has 5 servers across all cells.
- a local topology service to each cell, with servers only in that cell. Only contains local topology data about Tablets, and roll-ups of global data for efficient access. Typically, it has 3 servers in each cell.

Migration between implementations

We provide the `topo2topo` utility to migrate between one implementation and another of the topology service.

The process to follow in that case is:

- Start from a stable topology, where no resharding or reparenting is ongoing.
- Configure the new topology service so it has at least all the cells of the source topology service. Make sure it is running.
- Run the `topo2topo` program with the right flags. `-from_implementation`, `-from_root`, `-from_server` describe the source (old) topology service. `-to_implementation`, `-to_root`, `-to_server` describe the destination (new) topology service.
- Run `vtctl RebuildKeyspaceGraph` for each keyspace using the new topology service flags.
- Run `vtctl RebuildVSchemaGraph` using the new topology service flags.
- Restart all `vtgate` processes using the new topology service flags. They will see the same Keyspaces / Shards / Tablets / VSchema as before, as the topology was copied over.
- Restart all `vttablet` processes using the new topology service flags. They may use the same ports or not, but they will update the new topology when they start up, and be visible from `vtgate`.
- Restart all `vtctld` processes using the new topology service flags. So that the UI also shows the new data.

Sample commands to migrate from deprecated `zookeeper` to `zk2` topology would be:

```
# Let's assume the zookeeper client config file is already
# exported in $ZK_CLIENT_CONFIG, and it contains a global record
# pointing to: global_server1,global_server2
# an a local cell cell1 pointing to cell1_server1,cell1_server2
#
# The existing directories created by Vitess are:
# /zk/global/vt/...
# /zk/cell1/vt/...
#
# The new zk2 implementation can use any root, so we will use:
# /vitess/global in the global topology service, and:
# /vitess/cell1 in the local topology service.

# Create the new topology service roots in global and local cell.
zk -server global_server1,global_server2 touch -p /vitess/global
zk -server cell1_server1,cell1_server2 touch -p /vitess/cell1
```

```
# Store the flags in a shell variable to simplify the example below.
TOPOLOGY="-topo_implementation zk2 -topo_global_server_address
   global_server1,global_server2 -topo_global_root /vitess/global"

# Reference cell1 in the global topology service:
vtctl $TOPOLOGY AddCellInfo \
  -server_address cell1_server1,cell1_server2 \
  -root /vitess/cell1 \
  cell1

# Now copy the topology. Note the old zookeeper implementation does not need
# any server or root parameter, as it reads ZK_CLIENT_CONFIG.
topo2topo \
  -from_implementation zookeeper \
  -to_implementation zk2 \
  -to_server global_server1,global_server2 \
  -to_root /vitess/global \

# Rebuild SvrKeyspace objects in new service, for each keyspace.
vtctl $TOPOLOGY RebuildKeyspaceGraph keyspace1
vtctl $TOPOLOGY RebuildKeyspaceGraph keyspace2

# Rebuild SrvVSchema objects in new service.
vtctl $TOPOLOGY RebuildVSchemaGraph

# Now restart all vtgate, vttablet, vtctld processes replacing:
# -topo_implementation zookeeper
# With:
# -topo_implementation zk2
# -topo_global_server_address global_server1,global_server2
# -topo_global_root /vitess/global
#
# After this, the ZK_CLIENT_CONF file and environment variables are not needed
# any more.
```

Migration using the `Tee` implementation If your migration is more complex, or has special requirements, we also support a 'tee' implementation of the topo service interface. It is defined in `go/vt/topo/helpers/tee.go`. It allows communicating to two topo services, and the migration uses multiple phases:

- Start with the old topo service implementation we want to replace.
- Bring up the new topo service, with the same cells.
- Use `topo2topo` to copy the current data from the old to the new topo.
- Configure a `Tee` topo implementation to maintain both services.
  - Note we do not expose a plugin for this, so a small code change is necessary.
  - all updates will go to both services.
  - the `primary` topo service is the one we will get errors from, if any.
  - the `secondary` topo service is just kept in sync.
  - at first, use the old topo service as `primary`, and the new one as `secondary`.
  - then, change the configuration to use the new one as `primary`, and the old one as `secondary`. Reverse the lock order here.
  - then rollout a configuration to just use the new service.

Transport Security Model

Vitess exposes a few RPC services and internally uses RPCs. These RPCs can optionally utilize secure transport options to use TLS over the gRPC HTTP/2 transport protocol. This document explains how to use these features. Finally, we briefly cover how to secure the MySQL protocol transport to VTGate.

Overview

The following diagram represents all the RPCs we use in a Vitess cluster via gRPC:

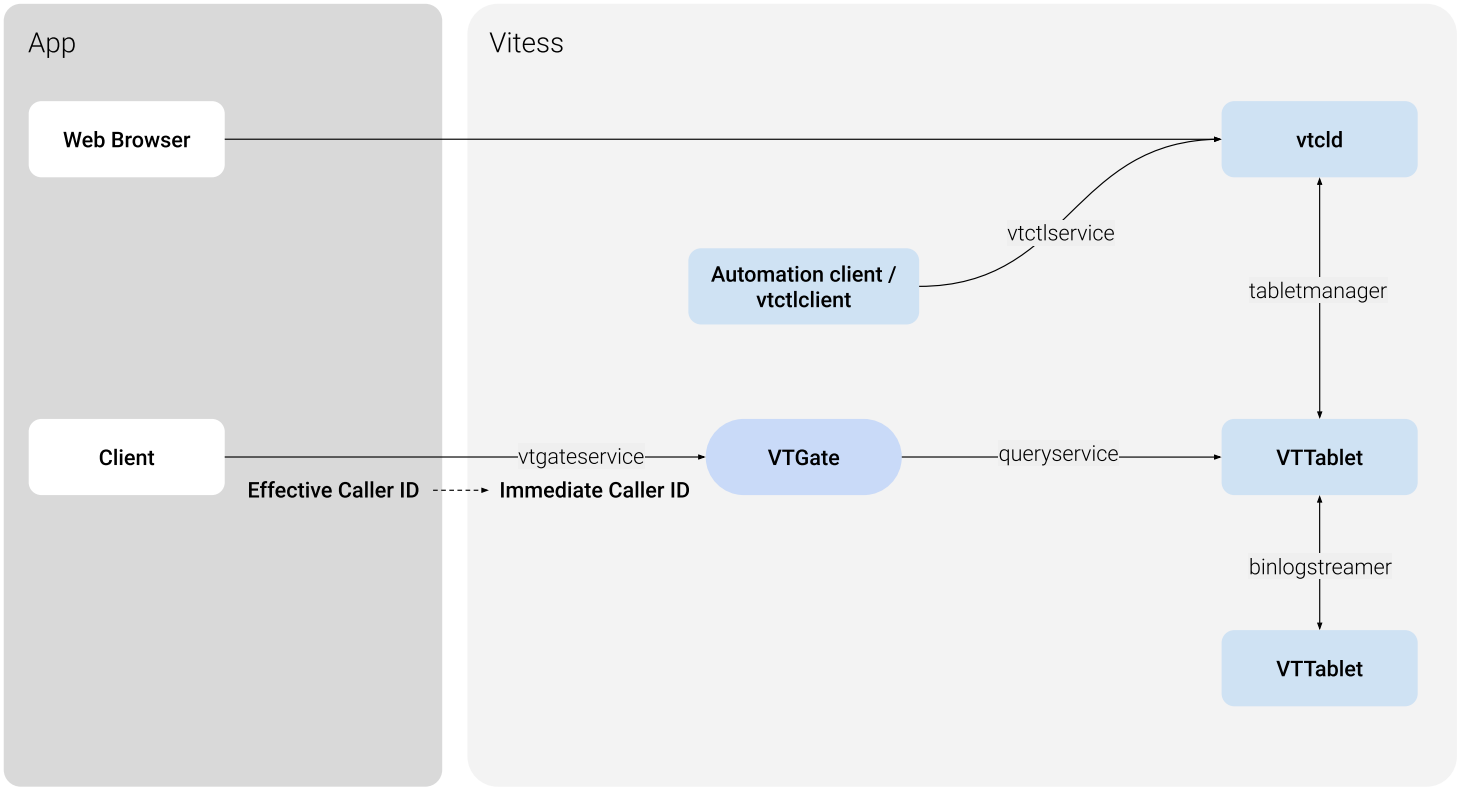


Figure 3: Vitess Transport Security Model Diagram

There are two main categories:

- Internal RPCs: They are used to connect Vitess components.
- Externally visible RPCs: They are used by the app to talk to Vitess. Note that it is not necessary to use this gRPC interface. It is still possible to instead use the MySQL protocol to VTGate, which is not covered in this document.

A few features in the Vitess ecosystem depend on authentication including Caller ID and table ACLs.

Caller ID

Caller ID is a feature provided by the Vitess stack to identify the source of queries. There are two different Caller IDs:

- Immediate Caller ID: It represents the secure client identity when it enters the Vitess side:
  - It is a single string representing the user connecting to Vitess (VTGate).
  - It is authenticated by the transport layer used.
  - It can be used by the Vitess TableACL feature.

- Effective Caller ID: It provides detailed information on the individual caller process:
  - It contains more information about the caller: principal, component, and sub-component.
  - It is provided by the application layer.
  - It is not authenticated.
  - It is exposed in query logs. Enabling it can be useful for debugging issues like the source of a slow query.

gRPC Transport

gRPC Encrypted Transport When using gRPC transport, Vitess can use the usual TLS security features. Please note that familiarity with TLS is necessary here:

- Any Vitess server can be configured to use TLS with the following command line parameters:
  - `grpc_cert`, `grpc_key`: server cert and key to use.
  - `grpc_ca` (optional): client cert chains to trust. If specified, the client must then use a certificate signed by one of the CA certs in the provided file.
- A Vitess go client can be configured with symmetrical parameters to enable TLS:
  - `xxxx_grpc_ca`: list of server cert signers to trust. I.E. the client will only connect to servers presenting a cert signed by one of the CAs in this file.
  - `xxxx_grpc_server_name`: common name of the server cert to trust. Instead of the hostname used to connect or IP SAN if using an IP to connect.
  - `xxxx_grpc_cert`, `xxxx_grpc_key`: client side cert and key to use in cases when the server requires client authentication.
  - Other clients can take similar parameters, in various ways. Please view each client's parameters for more information.

With these options, it is possible to use TLS-secured connections for all parts of the gRPC system. This enables the server side to authenticate the client, and/or the client to authenticate the server.

This is not enabled by default, as usually the different Vitess servers will run on a private network. It is also important to note, that in a Cloud environment, for example, usually all local traffic is already secured between VMs.

Options for vtctld

Name	Type	Definition
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_manager_grpc_cert	string	the cert to use to connect
-tablet_manager_grpc_key	string	the key to use to connect
-tablet_manager_grpc_server_name	string	the server name to use to validate server certificate
-throttler_client_grpc_ca	string	the server ca to use to validate servers when connecting
-throttler_client_grpc_cert	string	the cert to use to connect
-throttler_client_grpc_key	string	the key to use to connect
-throttler_client_grpc_server_name	string	the server name to use to validate server certificate
-vtgate_grpc_ca	string	the server ca to use to validate servers when connecting

Name	Type	Definition
-vtgate_grpc_cert	string	the cert to use to connect
-vtgate_grpc_key	string	the key to use to connect
-vtgate_grpc_server_name	string	the server name to use to validate server certificate
-vtworker_client_grpc_ca	string	the server ca to use to validate servers when connecting
-vtworker_client_grpc_cert	string	the cert to use to connect
-vtworker_client_grpc_key	string	the key to use to connect
-vtworker_client_grpc_server_name	string	the server name to use to validate server certificate

Options for vtgate

Name	Type	Definition
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate

Options for vttablet

Name	Type	Definition
-binlog_player_grpc_ca	string	the server ca to use to validate servers when connecting
-binlog_player_grpc_cert	string	the cert to use to connect
-binlog_player_grpc_key	string	the key to use to connect
-binlog_player_grpc_server_name	string	the server name to use to validate server certificate
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_manager_grpc_cert	string	the cert to use to connect
-tablet_manager_grpc_key	string	the key to use to connect
-tablet_manager_grpc_server_name	string	the server name to use to validate server certificate

Certificates and Caller ID Additionally, if a client uses a certificate to connect to Vitess (VTGate) via gRPC, the common name of that certificate is passed to vttablet as the Immediate Caller ID. It can then be used by table ACLs to grant read, write or admin access to individual tables. This should be used if different clients should have different access to Vitess tables.

Caller ID Override In a private network, where TLS security is not required, it might still be desirable to use table ACLs as a safety mechanism to prevent a user from accessing sensitive data. The gRPC connector provides the

`grpc_use_effective_callerid` flag for this purpose: if specified when running vtgate, the Effective Caller ID's principal is copied into the Immediate Caller ID, and then used throughout the Vitess stack.

Important: This is not secure. Any user code can provide any value for the Effective Caller ID's principal, and therefore access any data. This is intended as a safety feature to make sure some applications do not misbehave. Therefore, this flag is not enabled by default.

Example For a concrete example, see encrypted_transport_test.go in the source tree.

It first sets up all the certificates, some table ACLs, and then uses the golang client to connect with TLS. It also exercises the `grpc_use_effective_callerid` flag, by connecting without TLS.

MySQL Transport to VTGate

To get VTGate to support TLS use the `-mysql_server_ssl_cert` and `-mysql_server_ssl_key` VTGate options. To require client certificates, you can set `-mysql_server_ssl_ca`, containing the CA certificate you expect the client TLS certificates to be verified against.

Finally, if you want to require all VTGate clients to only be able to connect using TLS, you can use the `-mysql_server_require_secure_` flag.

Two-Phase Commit

{{< warning >}} Transaction commit is much slower when using 2PC. The authors of Vitess recommend that you design your VSchema so that cross-shard updates (and 2PC) are not required. {{< /warning >}}

Vitess 2PC allows you to perform atomic distributed commits. The feature is implemented using traditional MySQL transactions, and hence inherits the same guarantees. With this addition, Vitess can be configured to support the following three levels of atomicity:

1. **Single database**: At this level, only single database transactions are allowed. Any transaction that tries to go beyond a single database will fail.
2. **Multi database**: A transaction can span multiple databases, but the commit will be best effort. Partial commits are possible.
3. **2PC**: This is the same as Multi-database, but the commit will be atomic.

2PC commits are more expensive than multi-database because the system has to save away the statements before starting the commit process, and also clean them up after a successful commit. This is the reason why it is a separate option instead of being always on.

Isolation

2PC transactions guarantee atomicity: either the whole transaction commits, or it is rolled back entirely. It does not guarantee Isolation (in the ACID sense). This means that a third party that performs cross-database reads can observe partial commits while a 2PC transaction is in progress.

Guaranteeing ACID Isolation is very contentious and has high costs. Providing it by default would have made Vitess impractical for the most common use cases.

Configuring VTGate The atomicity policy is controlled by the `transaction_mode` flag. The default value is multi, and will set it in multi-database mode. This is the same as the previous legacy behavior.

To enforce single-database transactions, the VTGates can be started by specifying `transaction_mode=single`.

To enable 2PC, the VTGates need to be started with `transaction_mode=twopc`. The VTTablets will require additional flags, which will be explained below.

The VTGate `transaction_mode` flag decides what to allow. The application can independently request a specific atomicity for each transaction. The request will be honored by VTGate only if it does not exceed what is allowed by the `transaction_mode`. For example, `transaction_mode=single` will only allow single-db transactions. On the other hand, `transaction_mode=twopc` will allow all three levels of atomicity.

Driver APIs

The way to request atomicity from the application is driver-specific.

MySQL Protocol Clients can set the transaction mode via a session-variable:

```
set transaction_mode='twopc';
```

gRPC Clients

Go driver For the Go driver, you request the atomicity by adding it to the context using the WithAtomicity function. For more details, please refer to the respective GoDocs.

Python driver For Python, the begin function of the cursor has an optional single_db flag. If the flag is True, then the request is for a single-db transaction. If False (or unspecified), then the following commit call's twopc flag decides if the commit is 2PC or Best Effort (multi).

Adding support in a new driver The VTGate RPC API extends the Begin and Commit functions to specify atomicity. The API mimics the Python driver: The BeginRequest message provides a single_db flag and the CommitRequest message provides an atomic flag which is synonymous to twopc.

Configuring VTTablet

The following flags need to be set to enable 2PC support in VTTablet:

- **twopc_enable**: This flag needs to be turned on.
- **twopc_coordinator_address**: This should specify the address (or VIP) of the VTGate that VTTablet will use to resolve abandoned transactions.
- **twopc_abandon_age**: This is the time in seconds that specifies how long to wait before asking a VTGate to resolve an abandoned transaction.

With the above flags specified, every master VTTablet also turns into a watchdog. If any 2PC transaction is left lingering for longer than twopc_abandon_age seconds, then VTTablet invokes VTGate and requests it to resolve it. Typically, the abandon_age needs to be substantially longer than the time it takes for a typical 2PC commit to complete (10s of seconds).

Configuring MySQL

The usual default values of MySQL are sufficient. However, it is important to verify that the `wait_timeout` (28800) has not been changed. If this value was changed to be too short, then MySQL could prematurely kill a prepared transaction causing data loss.

Monitoring

A few additional variables have been added to /debug/vars. Failures described below should be rare. But these variables are present so you can build an alert mechanism if anything were to go wrong.

Critical failures

The following errors are not expected to happen. If they do, it means that 2PC transactions have failed to commit atomically:

- **InternalErrors.TwopcCommit**: This is a counter that shows the number of times a prepared transaction failed to fulfil a commit request.
- **InternalErrors.TwopcResurrection**: This counter is incremented if a new master failed to resurrect a previously prepared (and unresolved) transaction.

Alertable failures

The following failures are not urgent, but require investigation:

- **InternalErrors.WatchdogFail**: This counter is incremented if there are failures in the watchdog thread of VTTablet. This means that the watchdog is not able to alert VTGate of abandoned transactions.
- **Unresolved.Prepares**: This is a gauge that is set based on the number of lingering Prepared transactions that have been alive for longer than 5x the abandon age. This usually means that a distributed transaction has repeatedly failed to resolve. A more serious condition is when the metadata for a distributed transaction has been lost and this Prepare is now permanently orphaned.

Repairs

If any of the alerts fire, it is time to investigate. Once you identify the dtid or the VTTablet that originated the alert, you can navigate to the /twopcz URL. This will display three lists:

- **Failed Transactions**: A transaction reaches this state if it failed to commit. The only action allowed for such transactions is that you can discard it. However, you can record the DMLs that were involved and have someone come up with a plan to repair the partial commit.
- **Prepared Transactions**: Prepared transactions can be rolled back or committed. Prepared transactions must be remedied only if their root Distributed Transaction has been lost or resolved.
- **Distributed Transactions**: Distributed transactions can only be Concluded (marked as resolved).

Vindexes

A Vindex maps column values to keyspace IDs

A Vindex provides a way to map a column value to a `keyspace ID`. Since each shard in Vitess covers a range of `keyspace ID` values, this mapping can be used to identify which shard contains a row. A variety of vindexes are available to choose from with different trade-offs, and you can choose one that best suits your needs.

The Sharding Key is a concept that was introduced by NoSQL datastores. It is based on the fact that, in NoSQL databases, there is only one access path to the data, which is the Key. However, relational databases are more versatile with respect to the data stored and their relationships. So, sharding a database by only designating a single sharding key is often insufficient.

If one were to draw an analogy, the indexes in a database would be the equivalent of the key in a NoSQL datastore, except that databases allow multiple indexes per table, and there are many types of indexes. Extending this analogy to a sharded database results in different types of cross-shard indexes. In Vitess, these are called Vindexes.

Advantages

The advantages of Vindexes stem from their flexibility:

- A table can have multiple Vindexes.
- Vindexes can be NonUnique, which allows a column value to yield multiple keyspace IDs.

- Vindexes can be a simple function or be based on a lookup table.
- Vindexes can be shared across multiple tables.
- Custom Vindexes can be created and used, and Vitess will still know how to reshard using such Vindexes.

The Primary Vindex The Primary Vindex for a table is analogous to a database primary key. Every sharded table must have one defined. A Primary Vindex must be unique: given an input value, it must produce a single keyspace ID. At the time of an insert to the table, the unique mapping produced by the Primary Vindex determines the target shard for the inserted row. Conceptually, this is equivalent to a NoSQL Sharding Key, and we often informally refer to the Primary Vindex as the Sharding Key.

However, there is a subtle difference: NoSQL datastores allow a choice of the Sharding Key, but the Sharding Strategy or Function is generally hardcoded in the engine. In Vitess, the choice of Vindex allows control of how a column value maps to a keyspace ID. In other words, a Primary Vindex in Vitess not only defines the Sharding Key, but also decides the Sharding Strategy.

Uniqueness for a Primary Vindex does not mean that the column has to be a primary key or unique key in the MySQL schema for the underlying shard. You can have multiple rows that map to the same keyspace ID. The Vindex uniqueness constraint only ensures that all rows for a keyspace ID end up in the same shard.

Vindexes come in many varieties. Some of them can be used as Primary Vindex, and others have different purposes. We will describe their properties in the Predefined Vindexes section.

Secondary Vindexes Secondary Vindexes are additional vindexes against other columns of a table offering optimizations for WHERE clauses that do not use the Primary Vindex. Secondary Vindexes return a single or a limited set of `keyspace IDs` which will allow VTGate to only target shards where the relevant data is present. In the absence of a Secondary Vindex, VTGate would have to send the query to all shards (called a scatter query).

It is important to note that Secondary Vindexes are only used for making routing decisions. The underlying database shards will most likely need traditional indexes on those same columns, to allow efficient retrieval from the table on the underlying MySQL instances.

Unique and NonUnique Vindex A Unique Vindex is a vindex that yields at most one keyspace ID for a given input. Knowing that a Vindex is Unique is useful because VTGate can push down certain complex queries into VTTablet if it knows that the scope of that query can be limited to a single shard. Uniqueness is also a prerequisite for a Vindex to be used as Primary Vindex.

A NonUnique Vindex is analogous to a database non-unique index. It is a secondary index for searching by an alternate WHERE clause. An input value could yield multiple keyspace IDs, and rows could be matched from multiple shards. For example, if a table has a `name` column that allows duplicates, you can define a cross-shard NonUnique Vindex for it, and this will allow an efficient search for users that match a certain `name`.

Functional and Lookup Vindex A **Functional Vindex** is a vindex where the column value to keyspace ID mapping is pre-established, typically through an algorithmic function. In contrast, a **Lookup Vindex** is a vindex that provides the ability to create an association between a value and a keyspace ID, and recall it later when needed. Lookup Vindexes are sometimes also informally referred to as cross-shard indexes.

Typically, the Primary Vindex for a table is Functional. In some cases, it is the identity function where the input value yields itself as the keyspace id. However, other algorithms like a hashing function can also be used.

A Lookup Vindex is implemented as a MySQL lookup table that maps a column value to the keyspace id. This is usually needed when database user needs to efficiently find a row using a WHERE clause that does not contain the Primary Vindex. At the time of insert, the computed keyspace ID of the row is stored in the lookup table against the column value.

Lookup Vindex types The lookup table that implements a Lookup Vindex can be sharded or unsharded. Note that the lookup row is most likely not going to be in the same shard as the keyspace id it points to.

Vitess allows for the transparent population of these lookup table rows by assigning an owner table, which is the main table that requires this lookup. When a row is inserted into this owner table, the lookup row for it is created in the lookup table.

The lookup row is also deleted upon a delete of the corresponding row in the owner table. These essentially result in distributed transactions, which traditionally require 2PC to guarantee atomicity.

Consistent lookup vindexes use an alternate approach that makes use of careful locking and transaction sequences to guarantee consistency without using 2PC. This gives the best of both worlds, with the benefit of a consistent cross-shard vindex without paying the price of 2PC.

There are currently two vindex types in Vitess for consistent lookup:

- `consistent_lookup_unique`
- `consistent_lookup`

Shared Vindexes Relational databases encourage normalization, which allows the splitting of data into different tables to avoid duplication in the case of one-to-many relationships. In such cases, a key is shared between the two tables to indicate that the rows are related, a.k.a. `Foreign Key`.

In a sharded environment, it is often beneficial to keep those rows in the same shard. If a Lookup Vindex was created on the foreign key column of each of those tables, the backing tables would actually be identical. In such cases, Vitess allows sharing a single Lookup Vindex for multiple tables. One of these tables is designated as the owner of the Lookup Vindex, and is responsible for creating and deleting these associations. The other tables just reuse these associations.

An existing `lookup_unique` vindex can be trivially switched to a `consistent_lookup_unique` by changing the vindex type in the VSchema. This is because the data is compatible. Caveat: If you delete a row from the owner table, Vitess will not perform cascading deletes. This is mainly for efficiency reasons; the application is likely capable of doing this more efficiently.

As for a `lookup` vindex, it can be changed it to a `consistent_lookup` only if the `from` columns can uniquely identify the owner row. Without this, many potentially valid inserts would fail.

Functional Vindexes can be also be shared. However, there is no concept of ownership because the column to keyspace ID mapping is pre-established.

Lookup Vindex guidance The guidance for implementing lookup vindexes has been to create a two-column table. The first column (`from` column) should match the type of the column of the main table that needs the vindex. The second column (`to` column) should be a `BINARY` or a `VARBINARY` large enough to accommodate the keyspace id.

This guidance remains the same for unique lookup vindexes.

For non-unique lookup Vindexes, the lookup table should consist of multiple columns. The first column continues to be the input for computing the keyspace IDs. Beyond this, additional columns are needed to uniquely identify the owner row. This should typically be the primary key of the owner table. But it can be any other column that can be combined with the `from` column to uniquely identify the owner row. The last column remains the keyspace ID like before.

For example, if a user table had the columns `(user_id, email)`, where `user_id` was the primary key and `email` needed a non-unique lookup vindex, the lookup table would have the columns `(email, user_id, keyspace_id)`.

Independence The previously described properties are mostly independent of each other. Combining them gives rise to the following valid categories:

- **Functional Unique**: The most popular category because it is the one best suited to be a Primary Vindex.
- **Functional NonUnique**: There are currently no use cases that need this category.
- **Lookup Unique Owned**: Used for optimizing high QPS read queries that do not use the Primary Vindex columns in their WHERE clause. There is a price to pay: an extra write to the lookup table for insert and delete operations, and an extra lookup for read operations. However, it may be worth it to avoid high QPS read queries to be sent to all shards. The overheard of maintaining the lookup table is amortized as the number of shards grow.
- **Lookup Unique Unowned**: Can be used as an optimization as described in the Shared Vindexes section.
- **Lookup NonUnique Owned**: Used for high QPS queries on columns that are non-unique.
- **Lookup NonUnique Unowned**: You would rarely have to use this category because it is unlikely that you will be using a column as foreign key that is not unique within a shard. But it is theoretically possible.

Of the above categories, `Functional Unique` and `Lookup Unique Unowned` Vindexes can be a Primary Vindex. This is because those are the only ones that are unique and have the column to keyspace ID mapping pre-established. This is required because the Primary Vindex is responsible for assigning the keyspace ID for a row when it is created.

However, it is generally not recommended to use a Lookup Vindex as a Primary Vindex because it is too slow for resharding. If absolutely unavoidable, it is recommended to add a `keyspace ID` column to the tables that need this level of control of the row-to-shard mapping. While resharding, Vitess can use that column to efficiently compute the target shard. Vitess can also be configured to auto-populate that column on inserts. This is done using the reverse map feature explained below.

Defining Vindexes Vindexes are defined in the VSchema inside the `Vindexes` section of every keyspace. The `column_vindexes` section of each table in that keyspace may refer to the Vindex by name. Here is an example:

```
"name_keyspace_idx": {
  "type": "lookup",
  "params": {
    "table": "name_keyspace_idx",
    "from": "name",
    "to": "keyspace_id"
  },
  "owner": "user"
}
```

In the above case, the name of the vindex is `name_keyspace_idx`. It is of type `lookup`, and it is owned by the `user` table.

Every Vindex has an optional `params` section that contains a map of string key-value pairs. The keys and values differ depending on the vindex type and are explained below.

How Vindexes are used

Cost Vindexes have costs. For routing a query, the applicable Vindex with the lowest cost is chosen. The current general costs for the different Vindex Types are as follows:

Vindex Type	Cost
Identity	0
Functional	1
Lookup Unique	10
Lookup NonUnique	20

Select In the case of a simple select, Vitess scans the WHERE clause to match references to Vindex columns and chooses the best one to use. If there is no match and the query is simple without complex constructs like aggregates, etc., it is sent to all shards.

Vitess can handle more complex queries. For now, refer to the design doc for background information on how it handles them.

Insert

- The Primary Vindex is used to generate a keyspace ID.
- The keyspace ID is validated against the rest of the Vindexes on the table. There must exist a mapping from the column value(s) for these Secondary Vindexes to the keyspace ID.
- If a column value was not provided for a Vindex and the Vindex is capable of reverse mapping a keyspace ID to an input value, that function is used to auto-fill the column. If there is no reverse map, it is an error.

Update The WHERE clause is used to route the update. Updating the value of a Vindex column is supported, but with a restriction: the change in the column value should not result in the row being moved from one shard to another. A workaround

is to perform a delete followed by insert, which works as expected.

Delete If the table owns lookup vindexes, then the rows to be deleted are first read and the associated Vindex entries are deleted. Following this, the query is routed according to the WHERE clause.

Ignore Nulls There are situations where the from columns of a lookup vindex can be `NULL`. Such columns cannot be inserted in the lookup backing table due to the uniqueness constraints of a lookup. There are two ways to deal with a `NULL` value in the from column of a lookup vindex:

- Use a predefined vindex as the primary vindex of the backing table that supports the use of a `NULL` value. The table for predefined vindexes lists what types are and are not nullable.
- Enable the `ignore_nulls` option. If the input value of any of the columns is null, Vitess can skip the creation of the lookup row if `ignore_nulls` is enabled.

{{< info >}} Note: You can have `NULL` values for the primary vindex column, as long as that vindex allows it (e.g. xxhash). However, you cannot have `NULL` values for the lookup input column, unless you have enabled `ignore_nulls`. {{< /info >}}

Predefined Vindexes Vitess provides the following predefined Vindexes:

Name	Type	Description	Primary	Multi-column	Reversible	Nullable	Cost	Data types
binary	Functional Unique	Identity	Yes	No	Yes	Yes	0	Any
binary_md5	Functional Unique	MD5 hash	Yes	No	No	Yes	1	Any
consistent_lookup	Lookup NonUnique	Lookup table non-unique values	No	Identify Row	No	Yes only if	20	Any
consistent_lookup_unique	Lookup Unique	Lookup table unique values	If unowned	Identify Row	No	Yes only if	10	Any
hash	Functional Unique	DES null-key hash	Yes	No	Yes	No	1	64 bit or smaller numeric or equivalent type
lookup	Lookup NonUnique	Lookup table non-unique values	No	Identify Row	No	Yes only if	20	Any
lookup_unique	Lookup Unique	Lookup table unique values	If unowned	Identify Row	No	Yes only if	10	Any
null	Functional Unique	Always map to keyspace ID 0	Yes	No	No	Yes	100	Any
numeric	Functional Unique	Identity	Yes	No	Yes	No	0	64 bit or smaller numeric or equivalent type
numeric_static_map	Functional Unique	JSON file statically mapping input string values to keyspace IDs	Yes	No	No	No	1	Any
region_experimental	Functional Unique	Multi-column prefix-based hash for use in geo-partitioning	Yes	Yes	No	No	1	String and numeric type

Name	Type	Description	Primary	Multi-column	Reversible	Nullable	Cost	Data types
region_json	Functional Unique	Multi-column prefix-based hash combined with a JSON map for key-to-region mapping, for use in geo-partitioning	Yes	Yes	No	No	1	String and numeric type
reverse_bits	Functional Unique	Bit reversal	Yes	No	Yes	No	1	64 bit or smaller numeric or equivalent type
unicode_loose_md5	Functional Unique	Case-insensitive (UCA level 1) MD5 hash	Yes	No	No	Yes	1	String or binary types
unicode_loose_xxhash	Functional Unique	Case-insensitive (UCA level 1) xxHash64 hash	Yes	No	No	Yes	1	String or binary types
xxhash	Functional Unique	xxHash64 hash	Yes	No	No	Yes	1	Any

Consistent lookup vindexes, as described above, are a new category of Vindexes that are meant to replace the existing lookup Vindexes implementation. For the time being, they have a different name to allow for users to switch back and forth.

Under the Multi-column heading, an `Identify Row` comment indicates that the Vindex only uses the first column to map to the keyspace id(s). The rest of the columns are used to identify the owner row.

Lookup Vindexes support the following parameters:

- `table`: The backing table for the lookup vindex. It is recommended that the table name be qualified by its keyspace.
- `from`: The list of "from" columns. The first column is used for routing, and the rest of the columns are used for identifying the owner row.
- `to`: The name of the "to" keyspace_id column.
- `autocommit` (false): if true, vindex entries are updated in their own autocommit transaction. This is useful if values never get remapped to different values. For example, if the input column comes from an auto-increment value.
- `write_only` (false): if true, the vindex is kept updated, but a lookup will return all shards if the key is not found. This mode is used while the vindex is being populated and backfilled.
- `ignore_nulls` (false): if true, null values in input columns do not create entries in the lookup table. Otherwise, a null input results in an error.

The `numeric_static_map` vindex requires a `json_path` parameter. The file must contain a json that maps input values to keyspace ids.

The `region_experimental` vindex is an experimental vindex that uses the first one or two bytes of the input value as prefix for keyspace id. The rest of the bits are hashed. This allows you to group users of the same region within the same group of shards. The vindex requires a `region_bytes` parameter that specifies if the prefix is one or two bytes.

The `region_json` vindex requires an additional `region_map` file name that is used to compute the region from the country. The `region_bytes` is presumed to contain country codes.

Custom Vindexes can also be created as needed. At the moment there is no formal plugin system for custom Vindexes, but the interface is well-defined, and thus custom implementations including code performing arbitary lookups in other systems can be accomodated.

There are also the following legacy (deprecated) Vindexes. **Do not use these**:

Name	Type	Primary	Reversible	Cost
lookup_hash	Lookup NonUnique	No	No	20
lookup_hash_unique	Lookup Unique	If unowned	No	10
lookup_unicodeloosemd5_hash	Lookup NonUnique	No	No	20
lookup_unicodeloosemd5_hash_unique	Lookup Unique	If unowned	No	10

Sequences

This document describes the Vitess Sequences feature, and how to use it.

Motivation

MySQL provides the `auto-increment` feature to assign monotonically incrementing IDs to a column in a table. However, when a table is sharded across multiple instances, maintaining the same feature is a lot more tricky.

Vitess Sequences fill that gap:

- Inspired from the usual SQL sequences (implemented in different ways by Oracle, SQL Server and PostgreSQL).
- Very high throughput for ID creation, using a configurable in-memory block allocation.
- Transparent use, similar to MySQL auto-increment: when the field is omitted in an `insert` statement, the next sequence value is used.

When *not* to Use Auto-Increment

Before we go any further, an auto-increment column has limitations and drawbacks. let's explore this topic a bit here.

Security Considerations Using auto-increment can leak confidential information about a service. Let's take the example of a web site that store user information, and assign user IDs to its users as they sign in. The user ID is then passed in a cookie for all subsequent requests.

The client then knows their own user ID. It is now possible to:

- Try other user IDs and expose potential system vulnerabilities.
- Get an approximate number of users of the system (using the user ID).
- Get an approximate number of sign-ins during a week (creating two accounts a week apart, and diffing the two IDs).

Auto-incrementing IDs should be reserved for either internal applications, or exposed to the clients only when safe.

Alternatives Alternative to auto-incrementing IDs are:

- use a 64 bits random generator number. Try to insert a new row with that ID. If taken (because the statement returns an integrity error), try another ID.
- use a UUID scheme, and generate truely unique IDs.

Now that this is out of the way, let's get to MySQL auto-increment.

MySQL Auto-increment Feature

Let's start by looking at the MySQL auto-increment feature:

- A row that has no value for the auto-increment value will be given the next ID.
- The current value is stored in the table metadata.
- Values may be 'burned' (by rolled back transactions).
- Inserting a row with a given value that is higher than the current value will set the current value.
- The value used by the master in a statement is sent in the replication stream, so replicas will have the same value when re-playing the stream.
- There is no strict guarantee about ordering: two concurrent statements may have their commit time in one order, but their auto-incrementing ID in the opposite order (as the value for the ID is reserved when the statement is issued, not when the transaction is committed).
- MySQL has multiple options for auto-increment, like only using every N number (for multi-master configurations), or performance related features (locking that table's current ID may have concurrency implications).
- When inserting a row in a table with an auto-increment column, if the value for the auto-increment row is not set, the value for the column is returned to the client alongside the statement result.

Vitess Sequences

An early design was to use a single unsharded database and a table with an auto-increment value to generate new values. However, this has serious limitations, in particular throughtput, and storing one entry for each value in that table, for no reason.

So we decided instead to base sequences on a MySQL table, and use a single value in that table to describe which values the sequence should have next. To increase performance, we also support block allocation of IDs: each update to the MySQL table is only done every N IDs (N being configurable), and in between only memory structures in vttablet are updated, making the QPS only limited by RPC latency.

The sequence table then is an unsharded single row table that Vitess can use to generate monotonically increasing ids. The VSchema allows you to associate a column of a table with the sequence table. Once they are associated, an insert on that table will transparently fetch an id from the sequence table, fill in the value, and route the row to the appropriate shard.

Since sequences are unsharded tables, they will be stored in the database (in our tutorial example, this is the commerce database).

The final goal is to have Sequences supported with SQL statements, like:

```
/* DDL support */
CREATE SEQUENCE my_sequence;

SELECT NEXT VALUE FROM my_sequence;

ALTER SEQUENCE my_sequence ...;

DROP SEQUENCE my_sequence;

SHOW CREATE SEQUENCE my_sequence;
```

In the current implementation, we support the query access to Sequences, but not the administration commands yet.

Creating a Sequence *Note*: The names in this section are extracted from the examples/demo sample application.

To create a Sequence, a backing table must first be created and initialized with a single row. The columns for that table have to be respected.

This is an example:

```
create table user_seq(id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';

insert into user_seq(id, next_id, cache) values(0, 1, 100);
```

Then, the Sequence has to be defined in the VSchema for that keyspace:

```
{
  "sharded": false,
  "tables": {
    "user_seq": {
      "type": "sequence"
    },
    ...
  }
}
```

And the table it is going to be using it can also reference the Sequence in its VSchema:

```
{
  ...
  "tables" : {
    "user": {
      "column_vindexes": [
           ...
      ],
      "auto_increment": {
        "column": "user_id",
        "sequence": "user_seq"
      }
    },
```

After this done (and the Schema has been reloaded on master tablet, and the VSchema has been pushed), the sequence can be used.

Accessing a Sequence If a Sequence is used to fill in a column for a table, nothing further needs to be done. Just sending no value for the column will make vtgate insert the next Sequence value in its place.

It is also possible to access the Sequence directly with the following SQL constructs:

```
/* Returns the next value for the sequence */
select next value from my_sequence;

/* Returns the next value for the sequence, and also reserve 4 values after that. */
select next 5 values from my_sequence;
```

VSchema

Overview

VSchema stands for Vitess Schema. It is an abstraction layer that presents a unified view of the underlying keyspaces and shards, and gives the semblance of a single MySQL server.

For example, VSchema will contain the information about the sharding key for a sharded table. When the application issues a query with a WHERE clause that references the key, the VSchema information will be used to route the query to the appropriate shard.

Architecture

The VSchema is specified on a per-keyspace basis. Additionally, a separate set of RoutingRules can be specified. This information is stored in the global topology. A vtctld `RebuildVSchemaGraph` command combines the RoutingRules and the per-keyspace VSchemas into a unified data structure called SrvVSchema, which is deployed into the topo of each cell. The VTGates consume this information, which they use for planning and routing queries from the application.

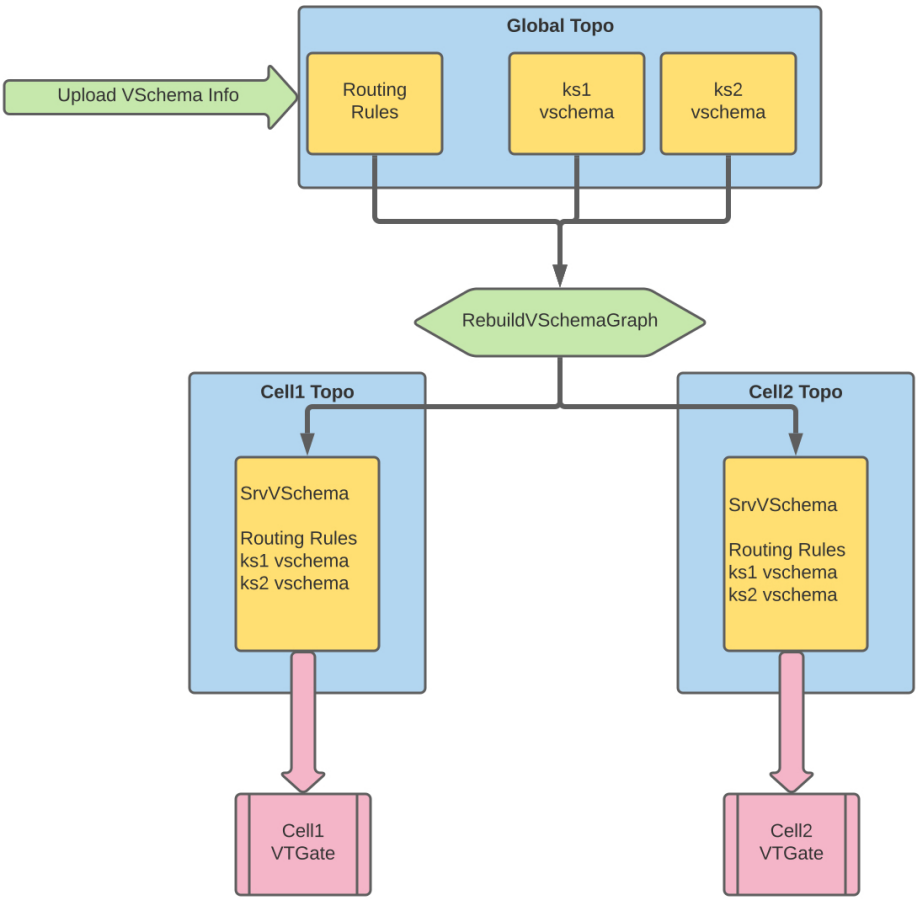


Figure 4: VSchema Architecture

Database Access Model

A Vitess Keyspace is the logical equivalent to a MySQL database. The usual syntax used to access databases in mysql work in Vitess as well. For example, you can connect to a specific database by specifying the database name in the connection parameter.

You can also change the database you are connected to through the `use` statement. While connected to a database, you can access a table from a different keyspace by qualifying the table name in your query, like `select * from other_keyspace.table`.

Tablet Types Unlike MySQL, the vitess servers unify all types of mysql servers. You can ask vitess to target a specific tablet type by qualifying it as part of the database name. For example, to access replica tablets, you may specify the database name as `keyspace@replica`. This name can also be specified in the connection string. If no tablet type is specified in the database name, then the value specified in VTGate's `-default_tablet_type` flag is used.

Unspecified Mode Additionally, you can connect without specifying a database name and still access tables without qualifying them by their keyspace. If the table name is unique across all keyspaces, then VTGate automatically sends the query to the associated keyspace. Otherwise, it returns an error. This mode is useful if you start off with a single keyspace and plan on splitting it into multiple parts.

You can still specify a tablet type for the unspecified mode. For example, you can connect to `@replica` if you want to access the replica tablets in unspecified mode.

Some frameworks require you to specify an explicit database name while connecting. In order to make them work in unspecified mode, you can specify the database name as `@replica` or `@master` instead of a blank one.

Sharded keyspaces require a VSchema

A VSchema is needed to tie together all the databases that Vitess manages. For a very trivial setup where there is only one unsharded keyspace, there is no need to specify a VSchema because Vitess will know that there is no other place to route a query.

If you have multiple unsharded keyspaces, you can still avoid defining a VSchema in one of two ways:

1. Connect to a keyspace and all queries are sent to it.
2. Connect to Vitess without specifying a keyspace (unspecified mode), but use qualified names for tables, like `keyspace.table` in your queries.

However, once the setup exceeds the above complexity, VSchemas become a necessity. Vitess has a working demo of VSchemas.

Sharding Model

In Vitess, a `keyspace` is sharded by `keyspace ID` ranges. Each row is assigned a keyspace ID, which acts like a street address, and it determines the shard where the row lives. In some respect, one could say that the `keyspace ID` is the equivalent of a NoSQL sharding key. However, there are some differences:

1. The `keyspace ID` is a concept that is internal to Vitess. The application does not need to know anything about it.
2. There is no physical column that stores the actual `keyspace ID`. This value is computed as needed.

This difference is significant enough that we do not refer to the keyspace ID as the sharding key. A Primary Vindex more closely resembles the NoSQL sharding key.

Mapping to a `keyspace ID`, and then to a shard, gives us the flexibility to reshard the data with minimal disruption because the `keyspace ID` of each row remains unchanged through the process.

Vindexes

The Vschema contains the Vindex for any sharded tables. The Vindex tells Vitess where to find the shard that contains a particular row for a sharded table. Every VSchema must have at least one Vindex, called the Primary Vindex, defined. The Primary Vindex is unique: given an input value, it produces a single keyspace ID, or value in the keyspace used to shard the table. The Primary Vindex is typically a functional Vindex: Vitess computes the keyspace ID as needed from a column in the sharded table.

Sequences

Auto-increment columns do not work very well for sharded tables. Vitess sequences solve this problem. Sequence tables must be specified in the VSchema, and then tied to table columns. At the time of insert, if no value is specified for such a column, VTGate will generate a number for it using the sequence table.

Reference tables

Vitess allows you to create an unsharded table and deploy it into all shards of a sharded keyspace. The data in such a table is assumed to be identical for all shards. In this case, you can specify that the table is of type `reference`, and should not specify any vindex for it. Any joins of this table with an unsharded table will be treated as a local join.

Typically, such a table has a canonical source in an unsharded keyspace, and the copies in the sharded keyspace are kept up-to-date through VReplication.

Per-Keyspace VSchema

The VSchema uses a flexible proto JSON format. Essentially, you can use `snake_case` or `camelCase` for the keys.

The configuration of your VSchema reflects the desired sharding configuration for your database, including whether or not your tables are sharded and whether you want to implement a secondary Vindex.

Commands You can use the following commands for maintaining the VSchema:

- `GetVSchema <keyspace>`
- `ApplyVSchema {-vschema=<vschema> || -vschema_file=<vschema file> || -sql=<sql> || -sql_file=<sql file>} [-cells=c1,c2,...] [-skip_rebuild] [-dry-run] <keyspace>`
- `RebuildVSchemaGraph [-cells=c1,c2,...]`
- `GetSrvVSchema <cell>`
- `DeleteSrvVSchema <cell>`

In order to verify that a VTGate has loaded SrvVSchema correctly, you can visit the `/debug/vschema` URL on the VTGate's http port.

Unsharded Table The following snippets show the necessary configs for creating a table in an unsharded keyspace:

Schema:

```
# lookup keyspace
create table name_user_idx(name varchar(128), user_id bigint, primary key(name, user_id));
```

VSchema:

```
// lookup keyspace
{
  "sharded": false,
  "tables": {
    "name_user_idx": {}
  }
}
```

For a normal unsharded table, the VSchema only needs to know the table name. No additional metadata is needed.

Sharded Table With Simple Primary Vindex To create a sharded table with a simple Primary Vindex, the VSchema requires more information:

Schema:

```
# user keyspace
create table user(user_id bigint, name varchar(128), primary key(user_id));
```

VSchema:

```
// user keyspace
{
  "sharded": true,
  "vindexes": {
    "hash": {
      "type": "hash"
    }
  },
  "tables": {
    "user": {
      "column_vindexes": [
        {
          "column": "user_id",
          "name": "hash"
        }
      ]
    }
  }
}
```

Because Vindexes can be shared, the JSON requires them to be specified in a separate `vindexes` section, and then referenced by name from the `tables` section. The VSchema above simply states that `user_id` uses `hash` as Primary Vindex. The first Vindex of every table must be the Primary Vindex.

Specifying A Sequence Since user is a sharded table, it will be beneficial to tie it to a Sequence. However, the sequence must be defined in the lookup (unsharded) keyspace. It is then referred from the user (sharded) keyspace. In this example, we are designating the `user_id` (Primary Vindex) column as the auto-increment.

Schema:

```
# lookup keyspace
create table user_seq(id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
insert into user_seq(id, next_id, cache) values(0, 1, 3);
```

For the sequence table, `id` is always 0. `next_id` starts off as 1, and the cache is usually a medium-sized number like 1000. In our example, we are using a small number to demonstrate how it works.

VSchema:

```
// lookup keyspace
{
  "sharded": false,
  "tables": {
    "user_seq": {
      "type": "sequence"
    }
  }
}
```

```
// user keyspace
{
  "sharded": true,
  "vindexes": {
    "hash": {
      "type": "hash"
    }
  },
  "tables": {
    "user": {
      "column_vindexes": [
        {
          "column": "user_id",
          "name": "hash"
        }
      ],
      "auto_increment": {
        "column": "user_id",
        "sequence": "lookup.user_seq"
      }
    }
  }
}
```

If necessary, the reference to the sequence table `lookup.user_seq` can be escaped using backticks.

Specifying A Secondary Vindex The following snippet shows how to configure a Secondary Vindex that is backed by a lookup table. In this case, the lookup table is configured to be in the unsharded lookup keyspace:

Schema:

```
# lookup keyspace
create table name_user_idx(name varchar(128), user_id bigint, primary key(name, user_id));
```

VSchema:

```
// lookup keyspace
{
  "sharded": false,
  "tables": {
    "name_user_idx": {}
  }
}

// user keyspace
{
  "sharded": true,
  "vindexes": {
    "name_user_idx": {
      "type": "lookup_hash",
      "params": {
        "table": "name_user_idx",
        "from": "name",
        "to": "user_id"
      },
      "owner": "user"
    }
  },
```

```
  "tables": {
    "user": {
      "column_vindexes": [
        {
          "column": "name",
          "name": "name_user_idx"
        }
      ]
    }
  }
}
```

To recap, a checklist for creating the shared Secondary Vindex is:

- Create physical `name_user_idx` table in lookup database.
- Define a routing for it in the lookup VSchema.
- Define a Vindex as type `lookup_hash` that points to it. Ensure that the `params` match the table name and columns.
- Define the owner for the Vindex as the `user` table.
- Specify that `name` uses the Vindex.

Currently, these steps have to be currently performed manually. However, extended DDLs backed by improved automation will simplify these tasks in the future.

The columns field For a table, you can specify an additional columns field. This can be used for two purposes:

- Specifying that a column contains text. If so, the VTGate planner can rewrite queries to leverage mysql's collation where possible.
- If the full list of columns is specified, then VTGate can resolve columns to their tables where needed, and also authoritative expand column lists, like in the case of a `select *` or `insert` statements with no column list.

Here is an example:

```
  "tables": {
    "user": {
      "column_vindexes": [
        {
          "column": "name",
          "name": "name_user_idx"
        }
      ],
      "columns": [
        {
          "name": "name",
          "type": "VARCHAR"
        },
        {
          "name": "keyspace_id",
          "type": "VARBINARY"
        }
      ],
      "column_list_authoritative": true
    }
  }
```

If a query goes across multiple shards and ordering is needed on the `name` column that is now specified as `VARCHAR`, then VTGate will leverage mysql to additionally the `weigh_string` of that column and use that value to order the merged results.

If `column_list_authoritative` is false or not specified, then VTGate will treat the list of columns as partial and will not automatically expand open-ended constructs like `select *`.

Advanced usage The examples/demo also shows more tricks you can perform:

- The `music` table uses a secondary lookup vindex `music_user_idx`. However, this lookup vindex is itself a sharded table.
- `music_extra` shares `music_user_idx` with `music`, and uses it as Primary Vindex.
- `music_extra` defines an additional Functional Vindex called `keyspace_id` which the demo auto-populates using the reverse mapping capability.
- There is also a `name_info` table that showcases a case-insensitive Vindex `unicode_loose_md5`.

Routing Rules

The `RoutingRules` section of the VSchema can be used to dynamically route traffic of a tablet to a different table than originally referenced in the query. This feature is used by the `MoveTables` workflow allowing you to change the application independently of when the actual traffic is moved from the old source table to the new target table.

Here is an example of `RoutingRules`

```
{
  "rules": [
    {
      "from_table": "customer",
      "to_tables": [
        "commerce.customer"
      ]
    },
    {
      "from_table": "customer.customer",
      "to_tables": [
        "commerce.customer"
      ]
    },
    {
      "from_table": "customer.customer@replica",
      "to_tables": [
        "commerce.customer"
      ]
    }
  ]
}
```

In the above JSON data structure, each rule maps an input table to a target. If the input table name is unqualified, then any unqualified reference to that table gets redirected to the fully qualified `to_tables`.

The `to_tables` field must contain only one entry and the table name must be fully qualified.

If the `from_table` is qualified by a keyspace, then a query that references that table will get redirected to the corresponding target table. The reference need not be explicit. For example, if you are connected to the `customer` keyspace, then an unqualified reference to the `customer` table is interpreted as a qualified reference to `customer.customer`.

You may further add a tablet type to the `from_table` field. If so, only queries that target that tablet type will get redirected. Although you can qualify a table by its keyspace in a query, there is no equivalent syntax for specifying the tablet type. The only way to choose a tablet type is through the `use` statement, like `use @replica`, or by specifying it in the connection string.

The more specific rules supercede the less specific one. For example, `customer.customer@replica` is chosen over `customer.customer` if the current tablet type is a `replica`.

If the `to_tables` have special characters that need escaping, you can use the mysql backtick syntax to do so. As for the `from_tables`, the table name should not be escaped. Instead, you should just concatenate the table with the keyspace without the backticks. In the following example, we are redirecting the `b.c` table to the `c.b` table in keyspace `a`:

```
{
  "rules": [
    {
      "from_table": "a.b.c",
      "to_tables": [
        "a.`c.b`"
      ]
    }
  ]
}
```

Commands You can use the following commands to maintain routing rules:

- `GetRoutingRules`
- `ApplyRoutingRules {-rules=<rules> || -rules_file=<rules_file>} [-cells=c1,c2,...] [-skip_rebuild] [-dry-run]`

Programs

description: Reference documents for list of Vitess programs

mysqlctl

`mysqlctl` is a command-line tool used for starting `mysqld` binaries. It is responsible for bootstrapping tasks such as generating a configuration file for `mysqld` and ensuring that `mysql_upgrade` is run in the data directory when restoring from backup.

`mysqld_safe` will be also be utilized when present. This helps ensure that `mysqld` is automatically restarted after failures.

Commands

init [-wait_time=5m] [-init_db_sql_file=(default)] Bootstraps a new `mysqld` instance. The MySQL version and flavor will be auto-detected, with a minimal configuration file applied. For example:

```
export VTDATAROOT=/tmp
mysqlctl \
 -alsologtostderr \
 -tablet_uid 101 \
 -mysql_port 12345 \
 init
```

init_config Bootstraps the configuration for a new `mysqld` instance. This command is the same as `init` except the `mysqld` server will not be started. For example:

```
export VTDATAROOT=/tmp
mysqlctl \
 -alsologtostderr \
 -tablet_uid 101 \
 -mysql_port 12345 \
 init_config
```

reinit_config Regenerate new configuration files for an existing `mysqld` instance. This could be helpful to revert configuration changes, or to pick up changes made to the bundled config in newer Vitess versions. For example:

```
export VTDATAROOT=/tmp
mysqlctl \
 -alsologtostderr \
 -tablet_uid 101 \
 -mysql_port 12345 \
 reinit_config
```

teardown [-wait_time=5m] [-force] Remove the data files for a previously shutdown `mysqld` instance. This is a destructive operation:

```
export VTDATAROOT=/tmp
mysqlctl -tablet_uid 101 -alsologtostderr teardown
```

start [-wait_time=5m] Resume an existing `mysqld` instance that was previously bootstrapped with `init` or `init_config`:

```
export VTDATAROOT=/tmp
mysqlctl -tablet_uid 101 -alsologtostderr start
```

shutdown [-wait_time=5m] Stop a `mysqld` instance that was previously started with `init` or `start`.

For large `mysqld` instances, you may need to extend the `-wait_time` as flushing dirty pages.

```
export VTDATAROOT=/tmp
mysqlctl -tablet_uid 101 -alsologtostderr shutdown
```

Options

The following global parameters apply to `mysqlctl`:

Name	Type	Definition
alsologtostderr	boolean	log to standard error as well as files
app_idle_timeout	duration	Idle timeout for app connections (default 1m0s)
app_pool_size	int	Size of the connection pool for app connections (default 40)
backup_engine_implementation	string	Specifies which implementation to use for creating new backups (builtin or xtrabackup). Restores will always be done with whichever engine created a given backup. (default “builtin”)

Name	Type	Definition
backup_storage_block_size	int	if backup_storage_compress is true, backup_storage_block_size sets the byte size for each block while compressing (default is 250000). (default 250000)
backup_storage_compress	boolean	if set, the backup files will be compressed (default is true). Set to false for instance if a backup_storage_hook is specified and it compresses the data. (default true)
backup_storage_hook	string	if set, we send the contents of the backup files through this hook.
backup_storage_implementation	string	which implementation to use for the backup storage feature
backup_storage_number_blocks	int	if backup_storage_compress is true, backup_storage_number_bloc sets the number of blocks that can be processed, at once, before the writer blocks, during compression (default is 2). It should be equal to the number of CPUs available for compression (default 2)
cpu_profile	string	write cpu profile to file
datadog-agent-host	string	host to send spans to. if empty, no tracing will be done
datadog-agent-port	string	port to send spans to. if empty, no tracing will be done
db-credentials-file	string	db credentials file; send SIGHUP to reload this file

Name	Type	Definition
db-credentials-server	string	db credentials server type (use 'file' for the file implementation) (default "file")
db_charset	string	Character set. Only utf8 or latin1 based character sets are supported.
db_connect_timeout_ms	int	connection timeout to mysqld in milliseconds (0 for no timeout)
db_dba_password	string	db dba password
db_dba_use_ssl	boolean	Set this flag to false to make the dba connection to not use ssl (default true)
db_dba_user	string	db dba user userKey (default "vt_dba")
db_flags	uint	Flag values as defined by MySQL.
db_flavor	string	Flavor overrid. Valid value is FilePos.
db_host	string	The host name for the tcp connection.
db_port	int	tcp port
db_server_name	string	server name of the DB we are connecting to.
db_socket	string	The unix socket to connect on. If this is specified, host and port will not be used.
db_ssl_ca	string	connection ssl ca
db_ssl_ca_path	string	connection ssl ca path
db_ssl_cert	string	connection ssl certificate
db_ssl_key	string	connection ssl key
dba_idle_timeout	duration	Idle timeout for dba connections (default 1m0s)
dba_pool_size	int	Size of the connection pool for dba connections (default 20)
disable_active_reparents	boolean	if set, do not allow active reparents. Use this to protect a cluster using external reparents.

Name	Type	Definition
emit_stats	boolean	true iff we should emit stats to push-based monitoring/stats backends
grpc_auth_mode	string	Which auth plugin implementation to use (eg: static)
grpc_auth_mtls_allowed_substrings	string	List of substrings of at least one of the client certificate names (separated by colon).
grpc_auth_static_client_creds	string	when using grpc_static_auth in the server, this file provides the credentials to use to authenticate with server
grpc_auth_static_password_file	string	JSON File to read the users/passwords from.
grpc_ca	string	ca to use, requires TLS, and enforces client cert check
grpc_cert	string	certificate to use, requires grpc_key, enables TLS
grpc_compression	string	how to compress gRPC, default: nothing, supported: snappy
grpc_enable_tracing	boolean	Enable GRPC tracing
grpc_initial_conn_window_size	int	grpc initial connection window size
grpc_initial_window_size	int	grpc initial window size
grpc_keepalive_time	duration	After a duration of this time, if the client doesn't see any activity, it pings the server to see if the transport is still alive. (default 10s)
grpc_keepalive_timeout	duration	After having pinged for keepalive check, the client waits for a duration of Timeout and if no activity is seen even after that the connection is closed. (default 10s)

Name	Type	Definition
grpc_key	string	key to use, requires grpc_cert, enables TLS
grpc_max_connection_age	duration	Maximum age of a client connection before GoAway is sent. (default 2562047h47m16.854775807s)
grpc_max_connection_age_grace	duration	Additional grace period after grpc_max_connection_age, after which connections are forcibly closed. (default 2562047h47m16.854775807s)
grpc_max_message_size	int	Maximum allowed RPC message size. Larger messages will be rejected by gRPC with the error 'exceeding the max size'. (default 16777216)
grpc_port	int	Port to listen on for gRPC calls
grpc_prometheus	boolean	Enable gRPC monitoring with Prometheus
grpc_server_initial_conn_window_size	int	gRPC server initial connection window size
grpc_server_initial_window_size	int	gRPC server initial window size
grpc_server_keepalive_enforcement_policy_min_time	duration	gRPC server minimum keepalive time (default 5m0s)
grpc_server_keepalive_enforcement_policy_permit_without_stream	boolean	gRPC server permit client keepalive pings even when there are no active streams (RPCs)
jaeger-agent-host	string	host and port to send spans to. if empty, no tracing will be done
keep_logs	duration	keep logs for this long (using ctime) (zero to keep forever)
keep_logs_by_mtime	duration	keep logs for this long (using mtime) (zero to keep forever)

Name	Type	Definition
lameduck-period	duration	keep running at least this long after SIGTERM before stopping (default 50ms)
log_backtrace_at	value	when logging hits line file:N, emit a stack trace
log_dir	string	If non-empty, write log files in this directory
log_err_stacks	boolean	log stack traces for errors
log_rotate_max_size	uint	size in bytes at which logs are rotated (glog.MaxSize) (default 1887436800)
logtostderr	boolean	log to standard error instead of files
master_connect_retry	duration	how long to wait in between replica reconnect attempts. Only precise to the second. (default 10s)
mem-profile-rate	int	profile every n bytes allocated (default 524288)
mutex-profile-fraction	int	profile every n mutex contention events (see runtime.SetMutexProfileFractio
mysql_auth_server_static_file	string	JSON File to read the users/passwords from.
mysql_auth_server_static_string	string	JSON representation of the users/passwords config.
mysql_auth_static_reload_interval	duration	Ticker to reload credentials
mysql_clientcert_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default “mysql_clear_password”)
mysql_port	int	mysql port (default 3306)
mysql_server_flush_delay	duration	Delay after which buffered response will be flushed to the client. (default 100ms)
mysql_socket	string	path to the mysql socket

Name	Type	Definition
mysqlctl_client_protocol	string	the protocol to use to talk to the mysqlctl server (default “grpc”)
mysqlctl_mycnf_template	string	template file to use for generating the my.cnf file during server init
mysqlctl_socket	string	socket file to use for remote mysqlctl actions (empty for local actions)
onterm_timeout	duration	wait no more than this for OnTermSync handlers before stopping (default 10s)
pid_file	string	If set, the process will write its pid to the named file, and delete it on graceful shutdown.
pool_hostname_resolve_interval	duration	if set force an update to all hostnames and reconnect if changed, defaults to 0 (disabled)
port	int	vttablet port (default 6612)
purge_logs_interval	duration	how often try to remove old logs (default 1h0m0s)
remote_operation_timeout	duration	time to wait for a remote operation (default 30s)
security_policy	string	the name of a registered security policy to use for controlling access to URLs - empty means allow all for anyone (built-in policies: deny-all, read-only)
service_map	value	comma separated list of services to enable (or disable if prefixed with ‘-’) Example: grpc-vtworker
sql-max-length-errors	int	truncate queries in error logs to the given length (default unlimited)

Name	Type	Definition
sql-max-length-ui	int	truncate queries in debug UIs to the given length (default 512) (default 512)
stats_backend	string	The name of the registered push-based monitoring/stats backend to use
stats_combine_dimensions	string	List of dimensions to be combined into a single “all” value in exported stats vars
stats_drop_variables	string	Variables to be dropped from the list of exported variables.
stats_emit_period	duration	Interval between emitting stats to all registered backends (default 1m0s)
stderrthreshold	value	logs at or above this threshold go to stderr (default 1)
tablet_dir	string	The directory within the vtdataroot to store vttablet/mysql files. Defaults to being generated by the tablet uid.
tablet_manager_protocol	string	the protocol to use to talk to vttablet (default “grpc”)
tablet_uid	uint	tablet uid (default 41983)
topo_global_root	string	the path of the global topology data in the global topology server
topo_global_server_address	string	the address of the global topology server
topo_implementation	string	the topology implementation to use
tracer	string	tracing service to use (default “noop”)
tracing-sampling-rate	float	sampling rate for the probabilistic jaeger sampler (default 0.1)
v	value	log level for V logs
version	boolean	print binary version
vmodule	value	comma-separated list of pattern=N settings for file-filtered logging

Name	Type	Definition
xbstream_restore_flags	string	flags to pass to xbstream command during restore. These should be space separated and will be added to the end of the command. These need to match the ones used for backup e.g. –compress / –decompress, –encrypt / –decrypt
xtrabackup_backup_flags	string	flags to pass to backup command. These should be space separated and will be added to the end of the command
xtrabackup_prepare_flags	string	flags to pass to prepare command. These should be space separated and will be added to the end of the command
xtrabackup_root_path	string	directory location of the xtrabackup executable, e.g., /usr/bin
xtrabackup_stream_mode	string	which mode to use if streaming, valid values are tar and xbstream (default “tar”)
xtrabackup_stripe_block_size	uint	Size in bytes of each block that gets sent to a given stripe before rotating to the next stripe (default 102400)
xtrabackup_stripes	uint	If greater than 0, use data striping across this many destination files to parallelize data transfer and decompression
xtrabackup_user	string	User that xtrabackup will use to connect to the database server. This user must have all necessary privileges. For details, please refer to xtrabackup documentation.

vtctl

description: vtctl Command Index

`vtctl` is a command-line tool used to administer a Vitess cluster. It is available as both a standalone tool (`vtctl`) and client-server (`vtctlclient` in combination with `vtctld`). Using client-server is recommended, as it provides an additional layer of security when using the client remotely.

Commands

Tablets

Name	Example Usage
InitTablet	`InitTablet [-allow_update] [-allow_different_shard] [-allow_master_override] [-parent] [-db_name_override=<db name>] [-hostname=<hostname>] [-mysql_port=<port>] [-port=<port>] [-grpc_port=<port>] [-tags=tag1:value1,tag2:value2] -keyspace=<keyspace> -shard=<shard> <tablet alias> <tablet type>`
GetTablet	`GetTablet <tablet alias>`
UpdateTabletAddrs	`UpdateTabletAddrs [-hostname <hostname>] [-ip-addr <ip addr>] [-mysql-port <mysql port>] [-vt-port <vt port>] [-grpc-port <grpc port>] <tablet alias>`
DeleteTablet	`DeleteTablet [-allow_master] <tablet alias> ...`
SetReadOnly	`SetReadOnly <tablet alias>`
SetReadWrite	`SetReadWrite <tablet alias>`
StartReplication	`StartReplication <tablet alias>`
StopReplication	`StopReplication <tablet alias>`
ChangeTabletType	`ChangeTabletType [-dry-run] <tablet alias> <tablet type>`
Ping	`Ping <tablet alias>`
RefreshState	`RefreshState <tablet alias>`
RefreshStateByShard	`RefreshStateByShard [-cells=c1,c2,...] <keyspace/shard>`
RunHealthCheck	`RunHealthCheck <tablet alias>`
IgnoreHealthError	`IgnoreHealthError <tablet alias> <ignore regexp>`
Sleep	`Sleep <tablet alias> <duration>`
ExecuteHook	`ExecuteHook <tablet alias> <hook name> [<param1=value1> <param2=value2> ...]`
ExecuteFetchAsApp	`ExecuteFetchAsApp [-max_rows=10000] [-json] [-use_pool] <tablet alias> <sql command>`
ExecuteFetchAsDba	`ExecuteFetchAsDba [-max_rows=10000] [-disable_binlogs] [-json] <tablet alias> <sql command>`
VReplicationExec	`VReplicationExec [-json] <tablet alias> <sql command>`
Backup	`Backup [-concurrency=4] [-allow_master=false] <tablet alias>`
RestoreFromBackup	`RestoreFromBackup <tablet alias>`
ReparentTablet	`ReparentTablet <tablet alias>`

Shards

Name	Example Usage
CreateShard	`CreateShard [-force] [-parent] <keyspace/shard>`
GetShard	`GetShard <keyspace/shard>`
ValidateShard	`ValidateShard [-ping-tablets] <keyspace/shard>`
ShardReplicationPositions	`ShardReplicationPositions <keyspace/shard>`
ListShardTablets	`ListShardTablets <keyspace/shard>`
SetShardIsMasterServing	`SetShardIsMasterServing <keyspace/shard> <is_master_serving>`

Name	Example Usage
SetShardTabletControl	`SetShardTabletControl [--cells=c1,c2,...] [--blacklisted_tables=t1,t2,...] [--remove] [--disable_query_service] <keyspace/shard> <tablet type>`
UpdateSrvKeyspacePartition	`UpdateSrvKeyspacePartition [--cells=c1,c2,...] [--remove] <keyspace/shard> <tablet type>`
SourceShardDelete	`SourceShardDelete <keyspace/shard> <uid>`
SourceShardAdd	`SourceShardAdd [--key_range=<keyrange>] [--tables=<table1,table2,...>] <keyspace/shard> <uid> <source keyspace/shard>`
ShardReplicationFix	`ShardReplicationFix <cell> <keyspace/shard>`
WaitForFilteredReplication	`WaitForFilteredReplication [-max_delay <max_delay, default 30s>] <keyspace/shard>`
RemoveShardCell	`RemoveShardCell [-force] [-recursive] <keyspace/shard> <cell>`
DeleteShard	`DeleteShard [-recursive] [-even_if_serving] <keyspace/shard> ...`
ListBackups	`ListBackups <keyspace/shard>`
BackupShard	`BackupShard [-allow_master=false] <keyspace/shard>`
RemoveBackup	`RemoveBackup <keyspace/shard> <backup name>`
InitShardMaster	`InitShardMaster [-force] [-wait_replicas_timeout=<duration>] <keyspace/shard> <tablet alias>`
PlannedReparentShard	`PlannedReparentShard -keyspace_shard=<keyspace/shard> [-new_master=<tablet alias>] [-avoid_master=<tablet alias>] [-wait_replicas_timeout=<duration>]`
EmergencyReparentShard	`EmergencyReparentShard -keyspace_shard=<keyspace/shard> -new_master=<tablet alias>`
TabletExternallyReparented	`TabletExternallyReparented <tablet alias>`
GenerateShardRanges	`GenerateShardRanges <num shards>`

Keyspaces

Name	Example Usage
CreateKeyspace	`CreateKeyspace [-sharding_column_name=name] [-sharding_column_type=type] [-served_from=tablettype1:ks1,tablettype2:ks2,...] [-force] [-keyspace_type=type] [-base_keyspace=base_keyspace] [-snapshot_time=time] <keyspace name>`
DeleteKeyspace	`DeleteKeyspace [-recursive] <keyspace>`
RemoveKeyspaceCell	`RemoveKeyspaceCell [-force] [-recursive] <keyspace> <cell>`
GetKeyspace	`GetKeyspace <keyspace>`
GetKeyspaces	`GetKeyspaces`
SetKeyspaceShardingInfo	`SetKeyspaceShardingInfo [-force] <keyspace name> [<column name>] [<column type>]`
SetKeyspaceServedFrom	`SetKeyspaceServedFrom [-source=<source keyspace name>] [-remove] [-cells=c1,c2,...] <keyspace name> <tablet type>`
RebuildKeyspaceGraph	`RebuildKeyspaceGraph [-cells=c1,c2,...] <keyspace> ...`
ValidateKeyspace	`ValidateKeyspace [-ping-tablets] <keyspace name>`
Reshard	`Reshard [-skip_schema_copy] <keyspace.workflow> <source_shards> <target_shards>`
MoveTables	`MoveTables [-cell=<cell>] [-tablet_types=<source_tablet_types>] -workflow=<workflow> <source_keyspace> <target_keyspace> <table_specs>`
DropSources	`DropSources [-dry_run] <keyspace.workflow>`
CreateLookupVindex	`CreateLookupVindex [-cell=<cell>] [-tablet_types=<source_tablet_types>] <keyspace> <json_spec>`

Name	Example Usage
ExternalizeVindex	`ExternalizeVindex <keyspace>.<vindex>`
Materialize	`Materialize <json_spec>, example : '{"workflow": "aaa", "source_keyspace": "source", "target_keyspace": "target", "table_settings": [{"target_table": "customer", "source_expression": "select * from customer", "create_ddl": "copy"}]}'`
SplitClone	`SplitClone <keyspace> <from_shards> <to_shards>`
VerticalSplitClone	`VerticalSplitClone <from_keyspace> <to_keyspace> <tables>`
VDiff	`VDiff [-source_cell=<cell>] [-target_cell=<cell>] [-tablet_types=replica] [-filtered_replication_wait_time=30s] <keyspace.workflow>`
MigrateServedTypes	`MigrateServedTypes [-cells=c1,c2,...] [-reverse] [-skip-refresh-state] <keyspace/shard> <served tablet type>`
MigrateServedFrom	`MigrateServedFrom [-cells=c1,c2,...] [-reverse] <destination keyspace/shard> <served tablet type>`
SwitchReads	`SwitchReads [-cells=c1,c2,...] [-reverse] -tablet_type={replica|rdonly} [-dry-run] <keyspace.workflow>`
SwitchWrites	`SwitchWrites [-filtered_replication_wait_time=30s] [-cancel] [-reverse_replication=false] [-dry-run] <keyspace.workflow>`
CancelResharding	`CancelResharding <keyspace/shard>`
ShowResharding	`ShowResharding <keyspace/shard>`
FindAllShardsInKeyspace	`FindAllShardsInKeyspace <keyspace>`
WaitForDrain	`WaitForDrain [-timeout <duration>] [-retry_delay <duration>] [-initial_wait <duration>] <keyspace/shard> <served tablet type>`

Generic

Name	Example Usage
Validate	`Validate [-ping-tablets]`
ListAllTablets	`ListAllTablets <cell name1>, <cell name2>, ...`
ListTablets	`ListTablets <tablet alias> ...`
Help	`Help [command name]`

Schema, Version, Permissions

Name	Example Usage
GetSchema	`GetSchema [-tables=<table1>,<table2>,...] [-exclude_tables=<table1>,<table2>,...] [-include-views] <tablet alias>`
ReloadSchema	`ReloadSchema <tablet alias>`
ReloadSchemaShard	`ReloadSchemaShard [-concurrency=10] [-include_master=false] <keyspace/shard>`
ReloadSchemaKeyspace	`ReloadSchemaKeyspace [-concurrency=10] [-include_master=false] <keyspace>`
ValidateSchemaShard	`ValidateSchemaShard [-exclude_tables=''] [-include-views] <keyspace/shard>`
ValidateSchemaKeyspace	`ValidateSchemaKeyspace [-exclude_tables=''] [-include-views] <keyspace name>`
ApplySchema	`ApplySchema [-allow_long_unavailability] [-wait_replicas_timeout=10s] {-sql=<sql> || -sql-file=<filename>} <keyspace>`

Name	Example Usage
CopySchemaShard	`CopySchemaShard [-tables=<table1>,<table2>,...] [-exclude_tables=<table1>,<table2>,...] [-include-views] [-skip-verify] [-wait_replicas_timeout=10s] {<source keyspace/shard> || <source tablet alias>} <destination keyspace/shard>`
ValidateVersionShard	`ValidateVersionShard <keyspace/shard>`
ValidateVersionKeyspace	`ValidateVersionKeyspace <keyspace name>`
GetPermissions	`GetPermissions <tablet alias>`
ValidatePermissionsShard	`ValidatePermissionsShard <keyspace/shard>`
ValidatePermissionsKeyspace	`ValidatePermissionsKeyspace <keyspace name>`
GetVSchema	`GetVSchema <keyspace>`
ApplyVSchema	`ApplyVSchema {-vschema=<vschema> || -vschema_file=<vschema file> || -sql=<sql> || -sql_file=<sql file>} [-cells=c1,c2,...] [-skip_rebuild] [-dry-run] <keyspace>`
GetRoutingRules	`GetRoutingRules`
ApplyRoutingRules	`ApplyRoutingRules {-rules=<rules> || -rules_file=<rules_file>} [-cells=c1,c2,...] [-skip_rebuild] [-dry-run]`
RebuildVSchemaGraph	`RebuildVSchemaGraph [-cells=c1,c2,...]`

Serving Graph

Name	Example Usage
GetSrvKeyspaceNames	`GetSrvKeyspaceNames <cell>`
GetSrvKeyspace	`GetSrvKeyspace <cell> <keyspace>`
GetSrvVSchema	`GetSrvVSchema <cell>`
DeleteSrvVSchema	`DeleteSrvVSchema <cell>`

Replication Graph

Name	Example Usage
GetShardReplication	`GetShardReplication <cell> <keyspace/shard>`

Cells

Name	Example Usage
AddCellInfo	`AddCellInfo [-server_address <addr>] [-root <root>] <cell>`
UpdateCellInfo	`UpdateCellInfo [-server_address <addr>] [-root <root>] <cell>`
DeleteCellInfo	`DeleteCellInfo [-force] <cell>`
GetCellInfoNames	`GetCellInfoNames`
GetCellInfo	`GetCellInfo <cell>`

CellsAliases

Name	Example Usage
AddCellsAlias	`AddCellsAlias [-cells <cell,cell2...>] <alias>`
UpdateCellsAlias	`UpdateCellsAlias [-cells <cell,cell2,...>] <alias>`
DeleteCellsAlias	`DeleteCellsAlias <alias>`
GetCellsAliases	`GetCellsAliases`

Queries

Name	Example Usage
VtGateExecute	`VtGateExecute -server <vtgate> [-bind_variables <JSON map>] [-keyspace <default keyspace>] [-tablet_type <tablet type>] [-options <proto text options>] [-json] <sql>`
VtTabletExecute	`VtTabletExecute [-username <TableACL user>] [-transaction_id <transaction_id>] [-options <proto text options>] [-json] <tablet alias> <sql>`
VtTabletBegin	`VtTabletBegin [-username <TableACL user>] <tablet alias>`
VtTabletCommit	`VtTabletCommit [-username <TableACL user>] <transaction_id>`
VtTabletRollback	`VtTabletRollback [-username <TableACL user>] <tablet alias> <transaction_id>`
VtTabletStreamHealth	`VtTabletStreamHealth [-count <count, default 1>] <tablet alias>`

Resharding Throttler

Name	Example Usage
ThrottlerMaxRates	`ThrottlerMaxRates -server <vtworker or vttablet>`
ThrottlerSetMaxRate	`ThrottlerSetMaxRate -server <vtworker or vttablet> <rate>`
GetThrottlerConfiguration	`GetThrottlerConfiguration -server <vtworker or vttablet> [<throttler name>]`
UpdateThrottlerConfiguration	`UpdateThrottlerConfiguration -server <vtworker or vttablet> [-copy_zero_values] "<configuration protobuf text>" [<throttler name>]`
ResetThrottlerConfiguration	`ResetThrottlerConfiguration -server <vtworker or vttablet> [<throttler name>]`

Topo

Name	Example Usage
TopoCat	`TopoCat [-cell <cell>] [-decode_proto] [-decode_proto_json] [-long] <path> [<path>...]`
TopoCp	`TopoCp [-cell <cell>] [-to_topo] <src> <dst>`

Workflows

Name	Example Usage
WorkflowCreate	`WorkflowCreate [-skip_start] <factoryName> [parameters...]`
WorkflowStart	`WorkflowStart <uuid>`
WorkflowStop	`WorkflowStop <uuid>`
WorkflowDelete	`WorkflowDelete <uuid>`
WorkflowWait	`WorkflowWait <uuid>`
WorkflowTree	`WorkflowTree`
WorkflowAction	`WorkflowAction <path> <name>`

Options

The following global options apply to `vtctl`:

Name	Type	Definition
-alsologtostderr		log to standard error as well as files
-app_idle_timeout	duration	Idle timeout for app connections (default 1m0s)
-app_pool_size	int	Size of the connection pool for app connections (default 40)
-azblob_backup_account_key_file	string	Path to a file containing the Azure Storage account key; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_KEY will be used as the key itself (NOT a file path)
-azblob_backup_account_name	string	Azure Storage Account name for backups; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_NAME will be used
-azblob_backup_container_name	string	Azure Blob Container Name
-azblob_backup_parallelism	int	Azure Blob operation parallelism (requires extra memory when increased) (default 1)
-azblob_backup_storage_root	string	Root prefix for all backup-related Azure Blobs; this should exclude both initial and trailing '/' (e.g. just 'a/b' not '/a/b/')
-backup_engine_implementation	string	Specifies which implementation to use for creating new backups (builtin or xtrabackup). Restores will always be done with whichever engine created a given backup. (default "builtin")
-backup_storage_block_size	int	if backup_storage_compress is true, backup_storage_block_size sets the byte size for each block while compressing (default is 250000). (default 250000)
-backup_storage_compress		if set, the backup files will be compressed (default is true). Set to false for instance if a backup_storage_hook is specified and it compresses the data. (default true)
-backup_storage_hook	string	if set, we send the contents of the backup files through this hook.
-backup_storage_implementation	string	which implementation to use for the backup storage feature
-backup_storage_number_blocks	int	if backup_storage_compress is true, backup_storage_number_blocks sets the number of blocks that can be processed, at once, before the writer blocks, during compression (default is 2). It should be equal to the number of CPUs available for compression (default 2)
-binlog_player_protocol	string	the protocol to download binlogs from a vttablet (default "grpc")
-binlog_use_v3_resharding_mode		True iff the binlog streamer should use V3-style sharding, which doesn't require a preset sharding key column. (default true)
-ceph_backup_storage_config	string	Path to JSON config file for ceph backup storage (default "ceph_backup_config.json")
-consul_auth_static_file	string	JSON File to read the topos/tokens from.
-cpu_profile	string	write cpu profile to file
-datadog-agent-host	string	host to send spans to. if empty, no tracing will be done
-datadog-agent-port	string	port to send spans to. if empty, no tracing will be done
-db-credentials-file	string	db credentials file; send SIGHUP to reload this file
-db-credentials-server	string	db credentials server type (use 'file' for the file implementation) (default "file")
-dba_idle_timeout	duration	Idle timeout for dba connections (default 1m0s)
-dba_pool_size	int	Size of the connection pool for dba connections (default 20)
-detach		detached mode - run vtcl detached from the terminal
-disable_active_reparents		if set, do not allow active reparents. Use this to protect a cluster using external reparents.
-discovery_high_replication_lag_minimum_serving	duration	the replication lag that is considered too high when selecting the minimum num vttablets for serving (default 2h0m0s)
-discovery_low_replication_lag	duration	the replication lag that is considered low enough to be healthy (default 30s)
-emit_stats		true iff we should emit stats to push-based monitoring/stats backends
-enable-consolidator		This option enables the query consolidator. (default true)

Name	Type	Definition
-enable-consolidator-replicas		This option enables the query consolidator only on replicas.
-enable-query-plan-field-caching		This option fetches & caches fields (columns) when storing query plans (default true)
-enable-tx-throttler		If true replication-lag-based throttling on transactions will be enabled.
-enable_hot_row_protection		If true, incoming transactions for the same row (range) will be queued and cannot consume all txpool slots.
-enable_hot_row_protection_dry_run		If true, hot row protection is not enforced but logs if transactions would have been queued.
-enable_queries		if set, allows vtgate and vttablet queries. May have security implications, as the queries will be run from this process.
-enable_transaction_limit		If true, limit on number of transactions open at the same time will be enforced for all users. User trying to open a new transaction after exhausting their limit will receive an error immediately, regardless of whether there are available slots or not.
-enable_transaction_limit_dry_run		If true, limit on number of transactions open at the same time will be tracked for all users, but not enforced.
-enforce_strict_trans_tables		If true, vttablet requires MySQL to run with STRICT_TRANS_TABLES or STRICT_ALL_TABLES on. It is recommended to not turn this flag off. Otherwise MySQL may alter your supplied values before saving them to the database. (default true)
-file_backup_storage_root	string	root directory for the file backup storage
-gcs_backup_storage_bucket	string	Google Cloud Storage bucket to use for backups
-gcs_backup_storage_root	string	root prefix for all backup-related object names
-grpc_auth_mode	string	Which auth plugin implementation to use (eg: static)
-grpc_auth_mtls_allowed_substrings	string	List of substrings of at least one of the client certificate names (separated by colon).
-grpc_auth_static_client_creds	string	when using grpc_static_auth in the server, this file provides the credentials to use to authenticate with server
-grpc_auth_static_password_file	string	JSON File to read the users/passwords from.
-grpc_ca	string	ca to use, requires TLS, and enforces client cert check
-grpc_cert	string	certificate to use, requires grpc_key, enables TLS
-grpc_compression	string	how to compress gRPC, default: nothing, supported: snappy
-grpc_enable_tracing		Enable GRPC tracing
-grpc_initial_conn_window_size	int	grpc initial connection window size
-grpc_initial_window_size	int	grpc initial window size
-grpc_keepalive_time	duration	After a duration of this time if the client doesn't see any activity it pings the server to see if the transport is still alive. (default 10s)
-grpc_keepalive_timeout	duration	After having pinged for keepalive check, the client waits for a duration of Timeout and if no activity is seen even after that the connection is closed. (default 10s)
-grpc_key	string	key to use, requires grpc_cert, enables TLS
-grpc_max_connection_age	duration	Maximum age of a client connection before GoAway is sent. (default 2562047h47m16.854775807s)
-grpc_max_connection_age_grace	duration	Additional grace period after grpc_max_connection_age, after which connections are forcibly closed. (default 2562047h47m16.854775807s)
-grpc_max_message_size	int	Maximum allowed RPC message size. Larger messages will be rejected by gRPC with the error 'exceeding the max size'. (default 16777216)
-grpc_port	int	Port to listen on for gRPC calls
-grpc_prometheus		Enable gRPC monitoring with Prometheus
-grpc_server_initial_conn_window_size	int	grpc server initial connection window size

Name	Type	Definition
-grpc_server_initial_window_size	int	grpc server initial window size
-grpc_server_keepalive_enforcement_policy_min_time	duration	grpc server minimum keepalive time (default 5m0s)
-grpc_server_keepalive_enforcement_policy_permit_without_stream		grpc server permit client keepalive pings even when there are no active streams (RPCs)
-heartbeat_enable		If true, vttablet records (if master) or checks (if replica) the current time of a replication heartbeat in the table _vt.heartbeat. The result is used to inform the serving state of the vttablet via healthchecks.
-heartbeat_interval	duration	How frequently to read and write replication heartbeat. (default 1s)
-hot_row_protection_concurrent_transactions	int	Number of concurrent transactions let through to the txpool/MySQL for the same hot row. Should be > 1 to have enough 'ready' transactions in MySQL and benefit from a pipelining effect. (default 5)
-hot_row_protection_max_global_queue_size	int	Global queue limit across all row (ranges). Useful to prevent that the queue can grow unbounded. (default 1000)
-hot_row_protection_max_queue_size	int	Maximum number of BeginExecute RPCs which will be queued for the same row (range). (default 20)
-jaeger-agent-host	string	host and port to send spans to. if empty, no tracing will be done
-keep_logs	duration	keep logs for this long (using ctime) (zero to keep forever)
-keep_logs_by_mtime	duration	keep logs for this long (using mtime) (zero to keep forever)
-lameduck-period	duration	keep running at least this long after SIGTERM before stopping (default 50ms)
-legacy_replication_lag_algorithm		use the legacy algorithm when selecting the vttablets for serving (default true)
-log_backtrace_at	value	when logging hits line file:N, emit a stack trace
-log_dir	string	If non-empty, write log files in this directory
-log_err_stacks		log stack traces for errors
-log_rotate_max_size	uint	size in bytes at which logs are rotated (glog.MaxSize) (default 1887436800)
-logtostderr		log to standard error instead of files
-master_connect_retry	duration	how long to wait in between replica reconnect attempts. Only precise to the second. (default 10s)
-mem-profile-rate	int	profile every n bytes allocated (default 524288)
-min_number_serving_vttablets	int	the minimum number of vttablets that will be continue to be used even with low replication lag (default 2)
-mutex-profile-fraction	int	profile every n mutex contention events (see runtime.SetMutexProfileFraction)
-mysql_auth_server_static_file	string	JSON File to read the users/passwords from.
-mysql_auth_server_static_string	string	JSON representation of the users/passwords config.
-mysql_auth_static_reload_interval	duration	Ticker to reload credentials
-mysql_clientcert_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default "mysql_clear_password")
-mysql_server_flush_delay	duration	Delay after which buffered response will flushed to client. (default 100ms)
-mysqlctl_client_protocol	string	the protocol to use to talk to the mysqlctl server (default "grpc")
-mysqlctl_mycnf_template	string	template file to use for generating the my.cnf file during server init
-mysqlctl_socket	string	socket file to use for remote mysqlctl actions (empty for local actions)
-onterm_timeout	duration	wait no more than this for OnTermSync handlers before stopping (default 10s)
-pid_file	string	If set, the process will write its pid to the named file, and delete it on graceful shutdown.
-pool_hostname_resolve_interval	duration	if set force an update to all hostnames and reconnect if changed, defaults to 0 (disabled)

Name	Type	Definition
-purge_logs_interval	duration	how often try to remove old logs (default 1h0m0s)
-query-log-stream-handler	string	URL handler for streaming queries log (default "/debug/querylog")
-querylog-filter-tag	string	string that must be present in the query as a comment for the query to be logged, works for both vtgate and vttablet
-querylog-format	string	format for query logs ("text" or "json") (default "text")
-queryserver-config-acl-exempt-acl	string	an acl that exempt from table acl checking (this acl is free to access any vitess tables).
-queryserver-config-enable-table-acl-dry-run		If this flag is enabled, tabletserver will emit monitoring metrics and let the request pass regardless of table acl check results
-queryserver-config-idle-timeout	int	query server idle timeout (in seconds), vttablet manages various mysql connection pools. This config means if a connection has not been used in given idle timeout, this connection will be removed from pool. This effectively manages number of connection objects and optimize the pool performance. (default 1800)
-queryserver-config-max-dml-rows	int	query server max dml rows per statement, maximum number of rows allowed to return at a time for an update or delete with either 1) an equality where clauses on primary keys, or 2) a subselect statement. For update and delete statements in above two categories, vttablet will split the original query into multiple small queries based on this configuration value.
-queryserver-config-max-result-size	int	query server max result size, maximum number of rows allowed to return from vttablet for non-streaming queries. (default 10000)
-queryserver-config-message-conn-pool-prefill-parallelism	int	DEPRECATED: Unused.
-queryserver-config-message-conn-pool-size	int	DEPRECATED
-queryserver-config-message-postpone-cap	int	query server message postpone cap is the maximum number of messages that can be postponed at any given time. Set this number to substantially lower than transaction cap, so that the transaction pool isn't exhausted by the message subsystem. (default 4)
-queryserver-config-passthrough-dmls		query server pass through all dml statements without rewriting
-queryserver-config-pool-prefill-parallelism	int	query server read pool prefill parallelism, a non-zero value will prefill the pool using the specified parallism.
-queryserver-config-pool-size	int	query server read pool size, connection pool is used by regular queries (non streaming, not in a transaction) (default 16)
-queryserver-config-query-cache-size	int	query server query cache size, maximum number of queries to be cached. vttablet analyzes every incoming query and generate a query plan, these plans are being cached in a lru cache. This config controls the capacity of the lru cache. (default 5000)
-queryserver-config-query-pool-timeout	int	query server query pool timeout (in seconds), it is how long vttablet waits for a connection from the query pool. If set to 0 (default) then the overall query timeout is used instead.
-queryserver-config-query-pool-waiter-cap	int	query server query pool waiter limit, this is the maximum number of queries that can be queued waiting to get a connection (default 5000)
-queryserver-config-query-timeout	int	query server query timeout (in seconds), this is the query timeout in vttablet side. If a query takes more than this timeout, it will be killed. (default 30)
-queryserver-config-schema-reload-time	int	query server schema reload time, how often vttablet reloads schemas from underlying MySQL instance in seconds. vttablet keeps table schemas in its own memory and periodically refreshes it from MySQL. This config controls the reload time. (default 1800)
-queryserver-config-stream-buffer-size	int	query server stream buffer size, the maximum number of bytes sent from vttablet for each stream call. It's recommended to keep this value in sync with vtgate's stream_buffer_size. (default 32768)

Name	Type	Definition
-queryserver-config-stream-pool-prefill-parallelism	int	query server stream pool prefill parallelism, a non-zero value will prefill the pool using the specified parallelism
-queryserver-config-stream-pool-size	int	query server stream connection pool size, stream pool is used by stream queries: queries that return results to client in a streaming fashion (default 200)
-queryserver-config-strict-table-acl		only allow queries that pass table acl checks
-queryserver-config-terse-errors		prevent bind vars from escaping in returned errors
-queryserver-config-transaction-cap	int	query server transaction cap is the maximum number of transactions allowed to happen at any given point of a time for a single vttablet. E.g. by setting transaction cap to 100, there are at most 100 transactions will be processed by a vttablet and the 101th transaction will be blocked (and fail if it cannot get connection within specified timeout) (default 20)
-queryserver-config-transaction-prefill-parallelism	int	query server transaction prefill parallelism, a non-zero value will prefill the pool using the specified parallism.
-queryserver-config-transaction-timeout	int	query server transaction timeout (in seconds), a transaction will be killed if it takes longer than this value (default 30)
-queryserver-config-txpool-timeout	int	query server transaction pool timeout, it is how long vttablet waits if tx pool is full (default 1)
-queryserver-config-txpool-waiter-cap	int	query server transaction pool waiter limit, this is the maximum number of transactions that can be queued waiting to get a connection (default 5000)
-queryserver-config-warn-result-size	int	query server result size warning threshold, warn if number of rows returned from vttablet for non-streaming queries exceeds this
-redact-debug-ui-queries		redact full queries and bind variables from debug UI
-remote_operation_timeout	duration	time to wait for a remote operation (default 30s)
-s3_backup_aws_endpoint	string	endpoint of the S3 backend (region must be provided)
-s3_backup_aws_region	string	AWS region to use (default “us-east-1”)
-s3_backup_aws_retries	int	AWS request retries (default -1)
-s3_backup_force_path_style		force the s3 path style
-s3_backup_log_level	string	determine the S3 loglevel to use from LogOff, LogDebug, LogDebugWithSigning, LogDebugWithHTTPBody, LogDebugWithRequestRetries, LogDebugWithRequestErrors (default “LogOff”)
-s3_backup_server_side_encryption	string	server-side encryption algorithm (e.g., AES256, aws:kms)
-s3_backup_storage_bucket	string	S3 bucket to use for backups
-s3_backup_storage_root	string	root prefix for all backup-related object names
-s3_backup_tls_skip_verify_cert		skip the ‘certificate is valid’ check for SSL connections
-security_policy	string	the name of a registered security policy to use for controlling access to URLs - empty means allow all for anyone (built-in policies: deny-all, read-only)
-service_map	value	comma separated list of services to enable (or disable if prefixed with ‘-’) Example: grpc-vtworker
-sql-max-length-errors	int	truncate queries in error logs to the given length (default unlimited)
-sql-max-length-ui	int	truncate queries in debug UIs to the given length (default 512) (default 512)
-srv_topo_cache_refresh	duration	how frequently to refresh the topology for cached entries (default 1s)
-srv_topo_cache_ttl	duration	how long to use cached entries for topology (default 1s)
-stats_backend	string	The name of the registered push-based monitoring/stats backend to use
-stats_combine_dimensions	string	List of dimensions to be combined into a single “all” value in exported stats vars
-stats_drop_variables	string	Variables to be dropped from the list of exported variables.

Name	Type	Definition
-stats_emit_period	duration	Interval between emitting stats to all registered backends (default 1m0s)
-stderrthreshold	value	logs at or above this threshold go to stderr (default 1)
-tablet_dir	string	The directory within the vtdataroot to store vttablet/mysql files. Defaults to being generated by the tablet uid.
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_manager_grpc_cert	string	the cert to use to connect
-tablet_manager_grpc_concurrency	int	concurrency to use to talk to a vttablet server for performance-sensitive RPCs (like ExecuteFetchAs{Dba,AllPrivs,App}) (default 8)
-tablet_manager_grpc_key	string	the key to use to connect
-tablet_manager_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_protocol	string	the protocol to use to talk to vttablet (default “grpc”)
-tablet_protocol	string	how to talk to the vttablets (default “grpc”)
-tablet_url_template	string	format string describing debug tablet url formatting. See the Go code for getTabletDebugURL() how to customize this. (default “http://{{.GetTabletHostPort}}”)
-throttler_client_grpc_ca	string	the server ca to use to validate servers when connecting
-throttler_client_grpc_cert	string	the cert to use to connect
-throttler_client_grpc_key	string	the key to use to connect
-throttler_client_grpc_server_name	string	the server name to use to validate server certificate
-throttler_client_protocol	string	the protocol to use to talk to the integrated throttler service (default “grpc”)
-topo_consul_watch_poll_duration	duration	time of the long poll for watch queries. (default 30s)
-topo_etcd_lease_ttl	int	Lease TTL for locks and master election. The client will use KeepAlive to keep the lease going. (default 30)
-topo_etcd_tls_ca	string	path to the ca to use to validate the server cert when connecting to the etcd topo server
-topo_etcd_tls_cert	string	path to the client cert to use to connect to the etcd topo server, requires topo_etcd_tls_key, enables TLS
-topo_etcd_tls_key	string	path to the client key to use to connect to the etcd topo server, enables TLS
-topo_global_root	string	the path of the global topology data in the global topology server
-topo_global_server_address	string	the address of the global topology server
-topo_implementation	string	the topology implementation to use
-topo_k8s_context	string	The kubeconfig context to use, overrides the ‘current-context’ from the config
-topo_k8s_kubeconfig	string	Path to a valid kubeconfig file.
-topo_k8s_namespace	string	The kubernetes namespace to use for all objects. Default comes from the context or in-cluster config
-topo_zk_auth_file	string	auth to use when connecting to the zk topo server, file contents should be :, e.g., digest:user:pass
-topo_zk_base_timeout	duration	zk base timeout (see zk.Connect) (default 30s)
-topo_zk_max_concurrency	int	maximum number of pending requests to send to a Zookeeper server. (default 64)
-topo_zk_tls_ca	string	the server ca to use to validate servers when connecting to the zk topo server
-topo_zk_tls_cert	string	the cert to use to connect to the zk topo server, requires topo_zk_tls_key, enables TLS
-topo_zk_tls_key	string	the key to use to connect to the zk topo server, enables TLS
-tracer	string	tracing service to use (default “noop”)

Name	Type	Definition
-tracing-sampling-rate	float	sampling rate for the probabilistic jaeger sampler (default 0.1)
-transaction-log-stream-handler	string	URL handler for streaming transactions log (default "/debug/txlog")
-transaction_limit_by_component		Include CallerID.component when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_principal		Include CallerID.principal when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_by_subcomponent		Include CallerID.subcomponent when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_username		Include VTGateCallerID.username when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_per_user	float	Maximum number of transactions a single user is allowed to use at any time, represented as fraction of -transaction_cap. (default 0.4)
-transaction_shutdown_grace_period	int	how long to wait (in seconds) for transactions to complete during graceful shutdown.
-twopc_abandon_age	float	time in seconds. Any unresolved transaction older than this time will be sent to the coordinator to be resolved.
-twopc_coordinator_address	string	address of the (VTGate) process(es) that will be used to notify of abandoned transactions.
-twopc_enable		if the flag is on, 2pc is enabled. Other 2pc flags must be supplied.
-tx-throttler-config	string	The configuration of the transaction throttler as a text formatted throttlerdata.Configuration protocol buffer message (default "target_replication_lag_sec: 2 max_replication_lag_sec: 10 initial_rate: 100 max_increase: 1 emergency_decrease: 0.5 min_duration_between_increases_sec: 40 max_duration_between_increases_sec: 62 min_duration_between_decreases_sec: 20 spread_backlog_across_sec: 20 age_bad_rate_after_sec: 180 bad_rate_increase: 0.1 max_rate_approach_threshold: 0.9")
-tx-throttler-healthcheck-cells	value	A comma-separated list of cells. Only tabletservers running in these cells will be monitored for replication lag by the transaction throttler.
-v	value	log level for V logs
-version		print binary version
-vmodule	value	comma-separated list of pattern=N settings for file-filtered logging
-vreplication_healthcheck_retry_delay	duration	healthcheck retry delay (default 5s)
-vreplication_healthcheck_timeout	duration	healthcheck retry delay (default 1m0s)
-vreplication_healthcheck_topology_refresh	duration	refresh interval for re-reading the topology (default 30s)
-vreplication_retry_delay	duration	delay before retrying a failed binlog connection (default 5s)
-vreplication_tablet_type	string	comma separated list of tablet types used as a source (default "REPLICA")
-vstream_packet_size	int	Suggested packet size for VReplication streamer. This is used only as a recommendation. The actual packet size may be more or less than this amount. (default 30000)
-vtctl_healthcheck_retry_delay	duration	delay before retrying a failed healthcheck (default 5s)
-vtctl_healthcheck_timeout	duration	the health check timeout period (default 1m0s)
-vtctl_healthcheck_topology_refresh	duration	refresh interval for re-reading the topology (default 30s)
-vtgate_grpc_ca	string	the server ca to use to validate servers when connecting
-vtgate_grpc_cert	string	the cert to use to connect
-vtgate_grpc_key	string	the key to use to connect
-vtgate_grpc_server_name	string	the server name to use to validate server certificate
-vtgate_protocol	string	how to talk to vtgate (default "grpc")
-wait-time	duration	time to wait on an action (default 24h0m0s)

Name	Type	Definition
-wait_for_drain_sleep_rdonly	duration	time to wait before shutting the query service on old RDONLY tablets during MigrateServedTypes (default 5s)
-wait_for_drain_sleep_replica	duration	time to wait before shutting the query service on old REPLICA tablets during MigrateServedTypes (default 15s)
-watch_replication_stream		When enabled, vttablet will stream the MySQL replication stream from the local server, and use it to support the include_event_token ExecuteOptions.
-xbstream_restore_flags	string	flags to pass to xbstream command during restore. These should be space separated and will be added to the end of the command. These need to match the ones used for backup e.g. –compress / –decompress, –encrypt / –decrypt
-xtrabackup_backup_flags	string	flags to pass to backup command. These should be space separated and will be added to the end of the command
-xtrabackup_prepare_flags	string	flags to pass to prepare command. These should be space separated and will be added to the end of the command
-xtrabackup_root_path	string	directory location of the xtrabackup executable, e.g., /usr/bin
-xtrabackup_stream_mode	string	which mode to use if streaming, valid values are tar and xbstream (default “tar”)
-xtrabackup_stripe_block_size	uint	Size in bytes of each block that gets sent to a given stripe before rotating to the next stripe (default 102400)
-xtrabackup_stripes	uint	If greater than 0, use data striping across this many destination files to parallelize data transfer and decompression
-xtrabackup_user	string	User that xtrabackup will use to connect to the database server. This user must have all necessary privileges. For details, please refer to xtrabackup documentation.

vtctl Cell Aliases Command Reference

series: vtctl ## Cell Aliases

The following `vtctl` commands are available for administering Cell Aliases.

Commands

AddCellsAlias Defines a group of cells within which replica/rdonly traffic can be routed across cells. By default, Vitess does not allow traffic between replicas that are part of different cells. Between cells that are not in the same group (alias), only master traffic can be routed.

Example

Flags

Name	Type	Definition
cells	string	The list of cell names that are members of this alias.

Arguments

- <alias> – Required. Alias name for this grouping.

Errors

- the <alias> argument is required for the <AddCellsAlias> command This error occurs if the command is not called with exactly one argument.

UpdateCellsAlias Updates the content of a CellAlias with the provided parameters. Empty values and intersections with other aliases are not supported.

Example

Flags

Name	Type	Definition
cells	string	The list of cell names that are members of this alias.

Arguments

- <alias> – Required. Alias name group to update.

Errors

- the <alias> argument is required for the <UpdateCellsAlias> command This error occurs if the command is not called with exactly one argument.

DeleteCellsAlias Deletes the CellsAlias for the provided alias. After deleting an alias, cells that were part of the group are not going to be able to route replica/rdonly traffic to the rest of the cells that were part of the grouping.

Example

Errors

- the <alias> argument is required for the <DeleteCellsAlias> command This error occurs if the command is not called with exactly one argument.

GetCellsAliases Fetches in json format all the existent cells alias groups.

Example

See Also

- vtctl command index

vtctl Cell Command Reference

series: vtctl ## Cells

The following `vtctl` commands are available for administering Cells.

Commands

AddCellInfo Registers a local topology service in a new cell by creating the CellInfo with the provided parameters. The address will be used to connect to the topology service, and we'll put Vitess data starting at the provided root.

Example

Flags

Name	Type	Definition
root	string	The root path the topology service is using for that cell.
server_address	string	The address the topology service is using for that cell.

Arguments

- <addr> – Required.
- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <cell> argument is required for the <AddCellInfo> command This error occurs if the command is not called with exactly one argument.

DeleteCellInfo Deletes the CellInfo for the provided cell. The cell cannot be referenced by any Shard record.

Example

Errors

- the <cell> argument is required for the <DeleteCellInfo> command This error occurs if the command is not called with exactly one argument.

GetCellInfo Prints a JSON representation of the CellInfo for a cell.

Example

Errors

- the <cell> argument is required for the <GetCellInfo> command This error occurs if the command is not called with exactly one argument.

GetCellInfoNames Lists all the cells for which we have a CellInfo object, meaning we have a local topology service registered.

Example

Errors

- <GetCellInfoNames> command takes no parameter This error occurs if the command is not called with exactly 0 arguments.

UpdateCellInfo Updates the content of a CellInfo with the provided parameters. If a value is empty, it is not updated. The CellInfo will be created if it doesn't exist.

Example

Flags

Name	Type	Definition
root	string	The root path the topology service is using for that cell.
server_address	string	The address the topology service is using for that cell.

Arguments

- <addr> – Required.
- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <cell> argument is required for the <UpdateCellInfo> command This error occurs if the command is not called with exactly one argument.

GetCellInfo Prints a JSON representation of the CellInfo for a cell.

Example

Errors

- the <cell> argument is required for the <GetCellInfo> command This error occurs if the command is not called with exactly one argument.

See Also

- vtctl command index

vtctl Generic Command Reference

series: vtctl ## Generic Commands

The following generic `vtctl` commands are available for administering Vitess.

Commands

Validate Validates that all nodes reachable from the global replication graph and that all tablets in all discoverable cells are consistent.

Example

Flags

Name	Type	Definition
ping-tablets	Boolean	Indicates whether all tablets should be pinged during the validation process

ListAllTablets Lists all tablets in an awk-friendly way.

Example

Arguments

- <cell name> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <cell name> argument is required for the <ListAllTablets> command This error occurs if the command is not called with exactly one argument.

ListTablets Lists specified tablets in an awk-friendly way.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>. To specify multiple values for this argument, separate individual values with a space.

Errors

- the <tablet alias> argument is required for the <ListTablets> command This error occurs if the command is not called with at least one argument.

Help Provides help for a command.

```
Help [command name]
```

See Also

- vtctl command index

vtctl Keyspace Command Reference

series: vtctl ## Keyspaces

The following `vtctl` commands are available for administering Keyspaces.

Commands

CreateKeyspace Creates the specified keyspace.

Example

Flags

Name	Type	Definition
force	Boolean	Proceeds even if the keyspace already exists
served_from	string	Specifies a comma-separated list of dbtype:keyspace pairs used to serve traffic
sharding_column_name	string	Specifies the column to use for sharding operations
sharding_column_type	string	Specifies the type of the column to use for sharding operations

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace name> argument is required for the <CreateKeyspace> command This error occurs if the command is not called with exactly one argument.

DeleteKeyspace Deletes the specified keyspace. In recursive mode, it also recursively deletes all shards in the keyspace. Otherwise, there must be no shards left in the keyspace.

Example

Flags

Name	Type	Definition
recursive	Boolean	Also recursively delete all shards in the keyspace.

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- must specify the <keyspace> argument for <DeleteKeyspace> This error occurs if the command is not called with exactly one argument.

RemoveKeyspaceCell Removes the cell from the Cells list for all shards in the keyspace, and the SrvKeyspace for that keyspace in that cell.

Example

Flags

Name	Type	Definition
force	Boolean	Proceeds even if the cell's topology service cannot be reached. The assumption is that you turned down the entire cell, and just need to update the global topo data.
recursive	Boolean	Also delete all tablets in that cell belonging to the specified keyspace.

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.
- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <keyspace> and <cell> arguments are required for the <RemoveKeyspaceCell> command This error occurs if the command is not called with exactly 2 arguments.

GetKeyspace Outputs a JSON structure that contains information about the Keyspace.

Example

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the <GetKeyspace> command This error occurs if the command is not called with exactly one argument.

GetKeyspaces Outputs a sorted list of all keyspaces.

Errors

- the <destination keyspace/shard> and <served tablet type> arguments are both required for the <MigrateServedFrom> command This error occurs if the command is not called with exactly 2 arguments.

SetKeyspaceShardingInfo Updates the sharding information for a keyspace.

Example

Flags

Name	Type	Definition
force	Boolean	Updates fields even if they are already set. Use caution before calling this command.

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.
- <column name> – Optional.
- <column type> – Optional.

Errors

- the <keyspace name> argument is required for the <SetKeyspaceShardingInfo> command. The <column name> and <column type> arguments are both optional This error occurs if the command is not called with between 1 and 3 arguments.
- both <column name> and <column type> must be set, or both must be unset

SetKeyspaceServedFrom Changes the ServedFromMap manually. This command is intended for emergency fixes. This field is automatically set when you call the *MigrateServedFrom* command. This command does not rebuild the serving graph.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to affect
remove	Boolean	Indicates whether to add (default) or remove the served from record
source	string	Specifies the source keyspace name

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.
- <tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <keyspace name> and <tablet type> arguments are required for the <SetKeyspaceServedFrom> command This error occurs if the command is not called with exactly 2 arguments.

RebuildKeyspaceGraph Rebuilds the serving data for the keyspace. This command may trigger an update to all connected clients.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to update

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace. To specify multiple values for this argument, separate individual values with a space.

Errors

- the <keyspace> argument must be used to specify at least one keyspace when calling the <RebuildKeyspaceGraph> command This error occurs if the command is not called with at least one argument.

ValidateKeyspace Validates that all nodes reachable from the specified keyspace are consistent.

Example

Flags

Name	Type	Definition
ping-tablets	Boolean	Specifies whether all tablets will be pinged during the validation process

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace name> argument is required for the <ValidateKeyspace> command This error occurs if the command is not called with exactly one argument.

```
Reshard  [-skip_schema_copy] <keyspace.workflow> <source_shards> <target_shards>
Start a Resharding process. Example: Reshard -cells='zone1,alias1'
   -tablet_types='master,replica,rdonly'  ks.workflow001 '0' '-80,80-'.
```

```
MoveTables  [-cell=<cell>] [-tablet_types=<source_tablet_types>] -workflow=<workflow>
   <source_keyspace> <target_keyspace> <table_specs>
Move table(s) to another keyspace, table_specs is a list of tables or the tables section of
   the vschema for the target keyspace. Example: '{"t1":{"column_vindexes": [{"column":
   "id1", "name": "hash"}]}, "t2":{"column_vindexes": [{"column": "id2", "name":
   "hash"}]}}'.  In the case of an unsharded target keyspace the vschema for each table may
   be empty. Example: '{"t1":{}, "t2":{}}'.
```

```
DropSources  [-dry_run] [-rename_tables] <keyspace.workflow>
After a MoveTables or Resharding workflow cleanup unused artifacts like source tables,
   source shards and blacklists.
```

```
CreateLookupVindex  [-cell=<cell>] [-tablet_types=<source_tablet_types>] <keyspace>
   <json_spec>
Create and backfill a lookup vindex. the json_spec must contain the vindex and colvindex
   specs for the new lookup.
```

```
ExternalizeVindex  <keyspace>.<vindex>
Externalize a backfilled vindex.
```

```
Materialize  <json_spec>, example : '{"workflow": "aaa", "source_keyspace": "source",
   "target_keyspace": "target", "table_settings": [{"target_table": "customer",
   "source_expression": "select * from customer", "create_ddl": "copy"}]}'
Performs materialization based on the json spec. Is used directly to form VReplication
   rules, with an optional step to copy table structure/DDL.
```

```
SplitClone  <keyspace> <from_shards> <to_shards>
Start the SplitClone process to perform horizontal resharding. Example: SplitClone ks '0'
   '-80,80-'
```

```
VerticalSplitClone  <from_keyspace> <to_keyspace> <tables>
Start the VerticalSplitClone process to perform vertical resharding. Example: SplitClone
   from_ks to_ks 'a,/b.*/'
```

```
VDiff  [-source_cell=<cell>] [-target_cell=<cell>] [-tablet_types=replica]
   [-filtered_replication_wait_time=30s] <keyspace.workflow>
Perform a diff of all tables in the workflow
```

MigrateServedTypes Migrates a serving type from the source shard to the shards that it replicates to. This command also rebuilds the serving graph. The <keyspace/shard> argument can specify any of the shards involved in the migration.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to update
filtered_replication_wait_time	Duration	Specifies the maximum time to wait, in seconds, for filtered replication to catch up on master migrations

Name	Type	Definition
reverse	Boolean	Moves the served tablet type backward instead of forward. Use in case of trouble
skip-refresh-state	Boolean	Skips refreshing the state of the source tablets after the migration, meaning that the refresh will need to be done manually, replica and rdonly only)
reverse_replication	Boolean	For master migration, enabling this flag reverses replication which allows you to rollback

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <served tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <source keyspace/shard> and <served tablet type> arguments are both required for the <MigrateServedTypes> command This error occurs if the command is not called with exactly 2 arguments.
- the <skip-refresh-state> flag can only be specified for non-master migrations

MigrateServedFrom Makes the <destination keyspace/shard> serve the given type. This command also rebuilds the serving graph.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to update

Name	Type	Definition
filtered_replication_wait_time	Duration	Specifies the maximum time to wait, in seconds, for filtered replication to catch up on master migrations
reverse	Boolean	Moves the served tablet type backward instead of forward. Use in case of trouble

Arguments

- <destination keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <served tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <destination keyspace/shard> and <served tablet type> arguments are both required for the <MigrateServedFrom> command This error occurs if the command is not called with exactly 2 arguments.

```
SwitchReads  [-cells=c1,c2,...] [-reverse] -tablet_types={replica|rdonly} [-dry-run]
   <keyspace.workflow>
Switch read traffic for the specified workflow.
```

```
SwitchWrites  [-timeout=30s] [-cancel] [-reverse] [-reverse_replication=false]
   -tablet_types={replica|rdonly} [-dry-run] <keyspace.workflow>
Switch write traffic for the specified workflow.
```

CancelResharding Permanently cancels a resharding in progress. All resharding related metadata will be deleted.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

ShowResharding Displays all metadata about a resharding in progress.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

FindAllShardsInKeyspace Displays all of the shards in the specified keyspace.

Example

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the <FindAllShardsInKeyspace> command This error occurs if the command is not called with exactly one argument.

WaitForDrain Blocks until no new queries were observed on all tablets with the given tablet type in the specified keyspace. This can be used as sanity check to ensure that the tablets were drained after running vtctl MigrateServedTypes and vtgate is no longer using them. If -timeout is set, it fails when the timeout is reached.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to look for tablets
initial_wait	Duration	Time to wait for all tablets to check in
retry_delay	Duration	Time to wait between two checks
timeout	Duration	Timeout after which the command fails

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

- <served tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <keyspace/shard> and <tablet type> arguments are both required for the <WaitForDrain> command This error occurs if the command is not called with exactly 2 arguments.

See Also

- vtctl command index

vtctl Query Command Reference

series: vtctl ## Queries

The following `vtctl` commands are available for administering queries.

Commands

VtGateExecute Executes the given SQL query with the provided bound variables against the vtgate server.

Example

Flags

Name	Type	Definition
json	Boolean	Output JSON instead of human-readable table
options	string	execute options values as a text encoded proto of the ExecuteOptions structure
server	string	VtGate server to connect to
target	string	keyspace:shard@tablet_type

Arguments

- <vtgate> – Required.
- <sql> – Required.

Errors

- the <sql> argument is required for the <VtGateExecute> command This error occurs if the command is not called with exactly one argument.
- query commands are disabled (set the -enable_queries flag to enable)
- error connecting to vtgate '%v': %v
- Execute failed: %v

VtTabletExecute Executes the given query on the given tablet. -transaction_id is optional. Use VtTabletBegin to start a transaction.

Example

Flags

Name	Type	Definition
json	Boolean	Output JSON instead of human-readable table
options	string	execute options values as a text encoded proto of the ExecuteOptions structure
transaction_id	Int	transaction id to use, if inside a transaction.
username	string	If set, value is set as immediate caller id in the request and used by vttablet for TableACL check

Arguments

- <TableACL user> – Required.
- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <sql> – Required.

Errors

- the <tablet_alias> and <sql> arguments are required for the <VtTabletExecute> command This error occurs if the command is not called with exactly 2 arguments.
- query commands are disabled (set the -enable_queries flag to enable)
- cannot connect to tablet %v: %v
- Execute failed: %v

VtTabletBegin Starts a transaction on the provided server.

Example

Flags

Name	Type	Definition
username	string	If set, value is set as immediate caller id in the request and used by vttablet for TableACL check

Arguments

- <TableACL user> – Required.
- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet_alias> argument is required for the <VtTabletBegin> command This error occurs if the command is not called with exactly one argument.
- query commands are disabled (set the -enable_queries flag to enable)
- cannot connect to tablet %v: %v
- Begin failed: %v

VtTabletCommit Commits the given transaction on the provided server.

Example

Flags

Name	Type	Definition
username	string	If set, value is set as immediate caller id in the request and used by vttablet for TableACL check

Arguments

- <TableACL user> – Required.
- <transaction_id> – Required.

Errors

- the <tablet_alias> and <transaction_id> arguments are required for the <VtTabletCommit> command This error occurs if the command is not called with exactly 2 arguments.
- query commands are disabled (set the -enable_queries flag to enable)
- cannot connect to tablet %v: %v

VtTabletRollback Rollbacks the given transaction on the provided server.

Example

Flags

Name	Type	Definition
username	string	If set, value is set as immediate caller id in the request and used by vttablet for TableACL check

Arguments

- <TableACL user> – Required.
- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <transaction_id> – Required.

Errors

- the <tablet_alias> and <transaction_id> arguments are required for the <VtTabletRollback> command This error occurs if the command is not called with exactly 2 arguments.
- query commands are disabled (set the -enable_queries flag to enable)
- cannot connect to tablet %v: %v

VtTabletStreamHealth Executes the StreamHealth streaming query to a vttablet process. Will stop after getting <count> answers.

Example

Flags

Name	Type	Definition
count	Int	number of responses to wait for

Arguments

- <count default 1> – Required.
- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <VtTabletStreamHealth> command This error occurs if the command is not called with exactly one argument.
- query commands are disabled (set the -enable_queries flag to enable)
- cannot connect to tablet %v: %v

See Also

- vtctl command index

vtctl Replication Graph Command Reference

series: vtctl ## Replication Graph

The following `vtctl` commands are available for administering the Replication Graph.

Commands

GetShardReplication Outputs a JSON structure that contains information about the ShardReplication.

Example

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.
- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <cell> and <keyspace/shard> arguments are required for the <GetShardReplication> command This error occurs if the command is not called with exactly 2 arguments.

See Also

- vtctl command index

vtctl Resharding Throttler Command Reference

series: vtctl ## Resharding Throttler

The following `vtctl` commands are available for administering Resharding Throttler.

Commands

ThrottlerMaxRates Returns the current max rate of all active resharding throttlers on the server.

Example

Flags

Name	Type	Definition
server	string	vtworker or vttablet to connect to

Arguments

- <vtworker or vttablet> – Required.

Errors

- the ThrottlerSetMaxRate command does not accept any positional parameters This error occurs if the command is not called with exactly 0 arguments.
- error creating a throttler client for <server> '%v': %v
- failed to get the throttler rate from <server> '%v': %v

ThrottlerSetMaxRate Sets the max rate for all active resharding throttlers on the server.

Example

Flags

Name	Type	Definition
server	string	vtworker or vttablet to connect to

Arguments

- <vtworker or vttablet> – Required.
- <rate> – Required.

Errors

- the <rate> argument is required for the <ThrottlerSetMaxRate> command This error occurs if the command is not called with exactly one argument.
- failed to parse rate '%v' as integer value: %v
- error creating a throttler client for <server> '%v': %v
- failed to set the throttler rate on <server> '%v': %v

GetThrottlerConfiguration Returns the current configuration of the MaxReplicationLag module. If no throttler name is specified, the configuration of all throttlers will be returned.

Example

Flags

Name	Type	Definition
server	string	vtworker or vttablet to connect to

Arguments

- <vtworker or vttablet> – Required.
- <throttler name> – Optional.

Errors

- the <GetThrottlerConfiguration> command accepts only <throttler name> as optional positional parameter This error occurs if the command is not called with more than 1 arguments.

- error creating a throttler client for <server> '%v': %v
- failed to get the throttler configuration from <server> '%v': %v

UpdateThrottlerConfiguration Updates the configuration of the MaxReplicationLag module. The configuration must be specified as protobuf text. If a field is omitted or has a zero value, it will be ignored unless -copy_zero_values is specified. If no throttler name is specified, all throttlers will be updated.

Example

Flags

Name	Type	Definition
copy_zero_values	Boolean	If true, fields with zero values will be copied as well
server	string	vtworker or vttablet to connect to

Arguments

- <vtworker or vttablet> – Required.
- <throttler name> – Optional.

Errors

- Failed to unmarshal the configuration protobuf text (%v) into a protobuf instance: %v
- error creating a throttler client for <server> '%v': %v
- failed to update the throttler configuration on <server> '%v': %v

ResetThrottlerConfiguration Resets the current configuration of the MaxReplicationLag module. If no throttler name is specified, the configuration of all throttlers will be reset.

Example

Flags

Name	Type	Definition
server	string	vtworker or vttablet to connect to

Arguments

- <vtworker or vttablet> – Required.
- <throttler name> – Optional.

Errors

- the <ResetThrottlerConfiguration> command accepts only <throttler name> as optional positional parameter This error occurs if the command is not called with more than 1 arguments.
- error creating a throttler client for <server> '%v': %v
- failed to get the throttler configuration from <server> '%v': %v

See Also

- vtctl command index

vtctl Schema, Version, Permissions Command Reference

series: vtctl ## Schema Versions & Permissions

The following `vtctl` commands are available for administering Schema, Versions and Permissions.

Commands

GetSchema Displays the full schema for a tablet, or just the schema for the specified tables in that tablet.

Example

Flags

Name	Type	Definition
exclude_tables	string	Specifies a comma-separated list of tables to exclude. Each is either an exact match, or a regular expression of the form /regexp/
include-views	Boolean	Includes views in the output
table_names_only	Boolean	Only displays table names that match
tables	string	Specifies a comma-separated list of tables for which we should gather information. Each is either an exact match, or a regular expression of the form /regexp/

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <GetSchema> command This error occurs if the command is not called with exactly one argument.

ReloadSchema Reloads the schema on a remote tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <ReloadSchema> command This error occurs if the command is not called with exactly one argument.

ReloadSchemaShard Reloads the schema on all the tablets in a shard.

Example

Flags

Name	Type	Definition
concurrency	Int	How many tablets to reload in parallel
include_master	Boolean	Include the master tablet

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ReloadSchemaShard> command This error occurs if the command is not called with exactly one argument.

ReloadSchemaKeyspace Reloads the schema on all the tablets in a keyspace.

Example

Flags

Name	Type	Definition
concurrency	Int	How many tablets to reload in parallel
include_master	Boolean	Include the master tablet(s)

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the <ReloadSchemaKeyspace> command This error occurs if the command is not called with exactly one argument.

ValidateSchemaShard Validates that the master schema matches all of the replicas.

Example

Flags

Name	Type	Definition
exclude_tables	string	Specifies a comma-separated list of tables to exclude. Each is either an exact match, or a regular expression of the form /regexp/
include-views	Boolean	Includes views in the validation

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ValidateSchemaShard> command This error occurs if the command is not called with exactly one argument.

ValidateSchemaKeyspace Validates that the master schema from shard 0 matches the schema on all of the other tablets in the keyspace.

Example

Flags

Name	Type	Definition
exclude_tables	string	Specifies a comma-separated list of tables to exclude. Each is either an exact match, or a regular expression of the form /regexp/
include-views	Boolean	Includes views in the validation

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace name> argument is required for the <ValidateSchemaKeyspace> command This error occurs if the command is not called with exactly one argument.

ApplySchema Applies the schema change to the specified keyspace on every master, running in parallel on all shards. The changes are then propagated to replicas via replication. If -allow_long_unavailability is set, schema changes affecting a large number of rows (and possibly incurring a longer period of unavailability) will not be rejected.

Example

Flags

Name	Type	Definition
allow_long_unavailability	Boolean	Allow large schema changes which incur a longer unavailability of the database.
sql	string	A list of semicolon-delimited SQL commands
sql-file	string	Identifies the file that contains the SQL commands. This file needs to exist on the server, rather than on the client.
wait_replicas_timeout	Duration	The amount of time to wait for replicas to receive the schema change via replication.

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the command<ApplySchema> command This error occurs if the command is not called with exactly one argument.

CopySchemaShard Copies the schema from a source shard's master (or a specific tablet) to a destination shard. The schema is applied directly on the master of the destination shard, and it is propagated to the replicas through binlogs.

Example

Flags

Name	Type	Definition
exclude_tables	string	Specifies a comma-separated list of tables to exclude. Each is either an exact match, or a regular expression of the form /regexp/
include-views	Boolean	Includes views in the output
tables	string	Specifies a comma-separated list of tables to copy. Each is either an exact match, or a regular expression of the form /regexp/

Name	Type	Definition
wait_replicas_timeout	Duration	The amount of time to wait for replicas to receive the schema change via replication.

Arguments

- <source tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <destination keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <source keyspace/shard> and <destination keyspace/shard> arguments are both required for the <CopySchemaShard> command. Instead of the <source keyspace/shard> argument, you can also specify <tablet alias> which refers to a specific tablet of the shard in the source keyspace This error occurs if the command is not called with exactly 2 arguments.

ValidateVersionShard Validates that the master version matches all of the replicas.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ValidateVersionShard> command This error occurs if the command is not called with exactly one argument.

ValidateVersionKeyspace Validates that the master version from shard 0 matches all of the other tablets in the keyspace.

Example

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace name> argument is required for the <ValidateVersionKeyspace> command This error occurs if the command is not called with exactly one argument.

GetPermissions Displays the permissions for a tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <GetPermissions> command This error occurs if the command is not called with exactly one argument.

ValidatePermissionsShard Validates that the master permissions match all the replicas.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ValidatePermissionsShard> command This error occurs if the command is not called with exactly one argument.

ValidatePermissionsKeyspace Validates that the master permissions from shard 0 match those of all of the other tablets in the keyspace.

Example

Arguments

- <keyspace name> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace name> argument is required for the <ValidatePermissionsKeyspace> command This error occurs if the command is not called with exactly one argument.

GetVSchema Displays the VTGate routing schema.

Example

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the <GetVSchema> command This error occurs if the command is not called with exactly one argument.

ApplyVSchema Applies the VTGate routing schema to the provided keyspace. Shows the result after application.

Example

Flags

Name	Type	Definition
cells	string	If specified, limits the rebuild to the cells, after upload. Ignored if skipRebuild is set.
dry-run	Boolean	If set, do not save the altered vschema, simply echo to console.
skip_rebuild	Boolean	If set, do not rebuild the SrvSchema objects.
sql	add vindex	A vschema ddl SQL statement (e.g. add vindex, `alter table t add vindex hash(id)`, etc)
sql_file	add vindex	A vschema ddl SQL statement (e.g. add vindex, `alter table t add vindex hash(id)`, etc)
vschema	string	Identifies the VTGate routing schema
vschema_file	string	Identifies the VTGate routing schema file

Arguments

- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <keyspace> argument is required for the <ApplyVSchema> command This error occurs if the command is not called with exactly one argument.
- either the <vschema> or <vschema>File flag must be specified when calling the <ApplyVSchema> command

```
GetRoutingRules
```

ApplyRoutingRules Applies the VSchema routing rules.

```
ApplyRoutingRules  {-rules=<rules> | -rules_file=<rules_file>} [-cells=c1,c2,...]
   [-skip_rebuild] [-dry-run]
```

Flags

Name	Type	Definition
cells	string	If specified, limits the rebuild to the cells, after upload. Ignored if skipRebuild is set.
dry-run	Boolean	If set, do not save the altered vschema, simply echo to console.
skip_rebuild	Boolean	If set, do not rebuild the SrvSchema objects.
-rules	string	Specify rules as a string.
-rules_file	string	Specify rules in a file.

RebuildVSchemaGraph Rebuilds the cell-specific SrvVSchema from the global VSchema objects in the provided cells (or all cells if none provided).

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to look for tablets

Errors

- <RebuildVSchemaGraph> doesn't take any arguments This error occurs if the command is not called with exactly 0 arguments.

See Also

- vtctl command index

vtctl Serving Graph Command Reference

series: vtctl ## Serving Graph

The following `vtctl` commands are available for administering the Serving Graph.

Commands

GetSrvKeyspaceNames Outputs a list of keyspace names.

Example

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <cell> argument is required for the <GetSrvKeyspaceNames> command This error occurs if the command is not called with exactly one argument.

GetSrvKeyspace Outputs a JSON structure that contains information about the SrvKeyspace.

Example

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.
- <keyspace> – Required. The name of a sharded database that contains one or more tables. Vitess distributes keyspace shards into multiple machines and provides an SQL interface to query the data. The argument value must be a string that does not contain whitespace.

Errors

- the <cell> and <keyspace> arguments are required for the <GetSrvKeyspace> command This error occurs if the command is not called with exactly 2 arguments.

GetSrvVSchema Outputs a JSON structure that contains information about the SrvVSchema.

Example

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <cell> argument is required for the <GetSrvVSchema> command This error occurs if the command is not called with exactly one argument.

```
DeleteSrvVSchema  <cell>
```

See Also

- vtctl command index

vtctl Shard Command Reference

series: vtctl ## Shards

The following `vtctl` commands are available for administering shards.

Commands

CreateShard Creates the specified shard.

Example

Flags

Name	Type	Definition
force	Boolean	Proceeds with the command even if the keyspace already exists
parent	Boolean	Creates the parent keyspace if it doesn't already exist

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <CreateShard> command This error occurs if the command is not called with exactly one argument.

GetShard Outputs a JSON structure that contains information about the Shard.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <GetShard> command This error occurs if the command is not called with exactly one argument.

ValidateShard Validates that all nodes that are reachable from this shard are consistent.

Example

Flags

Name	Type	Definition
ping-tablets	Boolean	Indicates whether all tablets should be pinged during the validation process

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ValidateShard> command This error occurs if the command is not called with exactly one argument.

ShardReplicationPositions Shows the replication status of each replica machine in the shard graph. In this case, the status refers to the replication lag between the master vttablet and the replica vttablet. In Vitess, data is always written to the master vttablet first and then replicated to all replica vttablets. Output is sorted by tablet type, then replication position. Use ctrl-C to interrupt command and see partial result if needed.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ShardReplicationPositions> command This error occurs if the command is not called with exactly one argument.

ListShardTablets Lists all tablets in the specified shard.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <ListShardTablets> command This error occurs if the command is not called with exactly one argument.

```
SetShardIsMasterServing <keyspace/shard> <is_master_serving>
```

SetShardTabletControl Sets the TabletControl record for a shard and type. Only use this for an emergency fix or after a finished vertical split. The *MigrateServedFrom* and *MigrateServedType* commands set this field appropriately already. Always specify the blacklisted_tables flag for vertical splits, but never for horizontal splits.To set the DisableQueryServiceFlag, keep 'blacklisted_tables' empty, and set 'disable_query_service' to true or false. Useful to fix horizontal splits gone wrong.To change the blacklisted tables list, specify the 'blacklisted_tables' parameter with the new list. Useful to fix tables that are being blocked after a vertical split.To just remove the ShardTabletControl entirely, use the 'remove' flag, useful after a vertical split is finished to remove serving restrictions.

Example

Flags

Name	Type	Definition
blacklisted_tables	string	Specifies a comma-separated list of tables to blacklist (used for vertical split). Each is either an exact match, or a regular expression of the form '/regexp/'.
cells	string	Specifies a comma-separated list of cells to update
disable_query_service	Boolean	Disables query service on the provided nodes. This flag requires 'blacklisted_tables' and 'remove' to be unset, otherwise it's ignored.
remove	Boolean	Removes cells for vertical splits.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)

  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <keyspace/shard> and <tablet type> arguments are both required for the <SetShardTabletControl> command This error occurs if the command is not called with exactly 2 arguments.

```
UpdateSrvKeyspacePartition [--cells=c1,c2,...] [--remove] <keyspace/shard> <tablet type>
```

SourceShardDelete Deletes the SourceShard record with the provided index. This is meant as an emergency cleanup function. It does not call RefreshState for the shard master.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <uid> – Required.

Errors

- the <keyspace/shard> and <uid> arguments are both required for the <SourceShardDelete> command This error occurs if the command is not called with at least 2 arguments.

SourceShardAdd Adds the SourceShard record with the provided index. This is meant as an emergency function. It does not call RefreshState for the shard master.

Example

Flags

Name	Type	Definition
key_range	string	Identifies the key range to use for the SourceShard

Name	Type	Definition
tables	string	Specifies a comma-separated list of tables to replicate (used for vertical split). Each is either an exact match, or a regular expression of the form /regexp/

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <uid> – Required.
- <source keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard>, <uid>, and <source keyspace/shard> arguments are all required for the <SourceShardAdd> command This error occurs if the command is not called with exactly 3 arguments.

ShardReplicationFix Walks through a ShardReplication object and fixes the first error that it encounters.

Example

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.
- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <cell> and <keyspace/shard> arguments are required for the ShardReplicationRemove command This error occurs if the command is not called with exactly 2 arguments.

WaitForFilteredReplication Blocks until the specified shard has caught up with the filtered replication of its source shard.

Example

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <WaitForFilteredReplication> command This error occurs if the command is not called with exactly one argument.

RemoveShardCell Removes the cell from the shard's Cells list.

Example

Flags

Name	Type	Definition
force	Boolean	Proceeds even if the cell's topology service cannot be reached. The assumption is that you turned down the entire cell, and just need to update the global topo data.
recursive	Boolean	Also delete all tablets in that cell belonging to the specified shard.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.

Errors

- the <keyspace/shard> and <cell> arguments are required for the <RemoveShardCell> command This error occurs if the command is not called with exactly 2 arguments.

DeleteShard Deletes the specified shard(s). In recursive mode, it also deletes all tablets belonging to the shard. Otherwise, there must be no tablets left in the shard.

Example

Flags

Name	Type	Definition
even_if_serving	Boolean	Remove the shard even if it is serving. Use with caution.
recursive	Boolean	Also delete all tablets belonging to the shard.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>. To specify multiple values for this argument, separate individual values with a space.

Errors

- the <keyspace/shard> argument must be used to identify at least one keyspace and shard when calling the <DeleteShard> command This error occurs if the command is not called with at least one argument.

ListBackups Lists all the backups for a shard.

Example

Errors

- action <ListBackups> requires <keyspace/shard> This error occurs if the command is not called with exactly one argument.

```
BackupShard [-allow_master=false] <keyspace/shard>
```

RemoveBackup Removes a backup for the BackupStorage.

Example

Arguments

- <backup name> – Required.

Errors

- action <RemoveBackup> requires <keyspace/shard> <backup name> This error occurs if the command is not called with exactly 2 arguments.

InitShardMaster Sets the initial master for a shard. Will make all other tablets in the shard replicas of the provided master. WARNING: this could cause data loss on an already replicating shard. PlannedReparentShard or EmergencyReparentShard should be used instead.

Example

Flags

Name	Type	Definition
force	Boolean	will force the reparent even if the provided tablet is not a master or the shard master
wait_replicas_timeout	Duration	time to wait for replicas to catch up in reparenting

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.
- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- action <InitShardMaster> requires <keyspace/shard> <tablet alias> This error occurs if the command is not called with exactly 2 arguments.
- active reparent commands disabled (unset the -disable_active_reparents flag to enable)

PlannedReparentShard Reparents the shard to a new master that can either be explicitly specified, or chosen by Vitess. Both the existing master and new master need to be up and running to use this command. If the existing master for the shard is down, you should use EmergencyReparentShard instead.

If the `new_master` flag is not provided, Vitess will try to automatically choose a replica to promote to master, avoiding any replicas specified in the `avoid_master` flag, if provided. Note that Vitess **will not consider any replicas outside the cell the current master is in for promotion**, therefore you **must** pass the `new_master` flag if you need to promote a replica in a different cell from the master. In the automated selection mode Vitess will prefer the most advanced replica for promotion, to minimize failover time.

Example

Flags

Name	Type	Definition
avoid_master	string	alias of a tablet that should not be the master, i.e. reparent to any replica other than this one
keyspace_shard	string	keyspace/shard of the shard that needs to be reparented
new_master	string	alias of a tablet that should be the new master
wait_replicas_timeout	Duration	time to wait for replicas to catch up in reparenting

Errors

- action <PlannedReparentShard> requires -keyspace_shard=<keyspace/shard> [-new_master=<tablet alias>] [-avoid_master=<tablet alias>] This error occurs if the command is not called with exactly 0 arguments.
- active reparent commands disabled (unset the -disable_active_reparents flag to enable)

- cannot use legacy syntax and flags -<keyspace_shard> and -<new_master> for action <PlannedReparentShard> at the same time

EmergencyReparentShard Reparents the shard to the new master. Assumes the old master is dead and not responding.

Example

Flags

Name	Type	Definition
keyspace_shard	string	keyspace/shard of the shard that needs to be reparented
new_master	string	alias of a tablet that should be the new master
wait_replicas_timeout	Duration	time to wait for replicas to catch up in reparenting

Errors

- action <EmergencyReparentShard> requires -keyspace_shard=<keyspace/shard> -new_master=<tablet alias> This error occurs if the command is not called with exactly 0 arguments.
- active reparent commands disabled (unset the -disable_active_reparents flag to enable)
- cannot use legacy syntax and flag -<new_master> for action <EmergencyReparentShard> at the same time

TabletExternallyReparented Changes metadata in the topology service to acknowledge a shard master change performed by an external tool. See Reparenting for more information.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <TabletExternallyReparented> command This error occurs if the command is not called with exactly one argument.

GenerateShardRanges Generates shard ranges assuming a keyspace with N shards.

Example

Flags

Name	Type	Definition
num_shards	int	Number of shards to generate shard ranges for. (default 2)

See Also

- vtctl command index

vtctl Tablet Command Reference

series: vtctl ## Tablets

The following `vtctl` commands are available for administering tablets.

Commands

InitTablet Initializes a tablet in the topology.

Example

Flags

Name	Type	Definition
allow_master_override	Boolean	Use this flag to force initialization if a tablet is created as master, and a master for the keyspace/shard already exists. Use with caution.
allow_update	Boolean	Use this flag to force initialization if a tablet with the same name already exists. Use with caution.
db_name_override	string	Overrides the name of the database that the vttablet uses
grpc_port	Int	The gRPC port for the vttablet process
hostname	string	The server on which the tablet is running
keyspace	string	The keyspace to which this tablet belongs
mysql_host	string	The mysql host for the mysql server
mysql_port	Int	The mysql port for the mysql server
parent	Boolean	Creates the parent shard and keyspace if they don't yet exist
port	Int	The main port for the vttablet process
shard	string	The shard to which this tablet belongs
tags	string	A comma-separated list of key:value pairs that are used to tag the tablet

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <tablet type> – Required. The vttablet's role. Valid values are:

  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <tablet alias> and <tablet type> arguments are both required for the <InitTablet> command This error occurs if the command is not called with exactly 2 arguments.

GetTablet Outputs a JSON structure that contains information about the Tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <GetTablet> command This error occurs if the command is not called with exactly one argument.

IgnoreHealthError Sets the regexp for health check errors to ignore on the specified tablet. The pattern has implicit ^$ anchors. Set to empty string or restart vttablet to stop ignoring anything.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <ignore regexp> – Required.

Errors

- the <tablet alias> and <ignore regexp> arguments are required for the <IgnoreHealthError> command This error occurs if the command is not called with exactly 2 arguments.

UpdateTabletAddrs Updates the IP address and port numbers of a tablet.

Example

Flags

Name	Type	Definition
grpc-port	Int	The gRPC port for the vttablet process
hostname	string	The fully qualified host name of the server on which the tablet is running.
mysql-port	Int	The mysql port for the mysql daemon
mysql_host	string	The mysql host for the mysql server
vt-port	Int	The main port for the vttablet process

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <UpdateTabletAddrs> command This error occurs if the command is not called with exactly one argument.

DeleteTablet Deletes tablet(s) from the topology.

Example

Flags

Name	Type	Definition
allow_master	Boolean	Allows for the master tablet of a shard to be deleted. Use with caution.

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>. To specify multiple values for this argument, separate individual values with a space.

Errors

- the <tablet alias> argument must be used to specify at least one tablet when calling the <DeleteTablet> command This error occurs if the command is not called with at least one argument.

SetReadOnly Sets the tablet as read-only.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <SetReadOnly> command This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

SetReadWrite Sets the tablet as read-write.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <SetReadWrite> command This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

StartReplication Starts replication on the specified tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- action <StartReplication> requires <tablet alias> This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

StopReplication Stops replication on the specified tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- action <StopReplication> requires <tablet alias> This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

ChangeTabletType Changes the db type for the specified tablet, if possible. This command is used primarily to arrange replicas, and it will not convert a master.NOTE: This command automatically updates the serving graph.

Example

Flags

Name	Type	Definition
dry-run	Boolean	Lists the proposed change without actually executing it

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <tablet type> – Required. The vttablet's role. Valid values are:
  - backup – A replicated copy of data that is offline to queries other than for backup purposes
  - batch – A replicated copy of data for OLAP load patterns (typically for MapReduce jobs)
  - drained – A tablet that is reserved for a background process. For example, a tablet used by a vtworker process, where the tablet is likely lagging in replication.
  - experimental – A replicated copy of data that is ready but not serving query traffic. The value indicates a special characteristic of the tablet that indicates the tablet should not be considered a potential master. Vitess also does not worry about lag for experimental tablets when reparenting.
  - master – A primary copy of data
  - rdonly – A replicated copy of data for OLAP load patterns
  - replica – A replicated copy of data ready to be promoted to master
  - restore – A tablet that is restoring from a snapshot. Typically, this happens at tablet startup, then it goes to its right state.
  - spare – A replicated copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <tablet alias> and <db type> arguments are required for the <ChangeTabletType> command This error occurs if the command is not called with exactly 2 arguments.
- failed reading tablet %v: %v
- invalid type transition %v: %v -> %v

Ping hecks that the specified tablet is awake and responding to RPCs. This command can be blocked by other in-flight operations.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <Ping> command This error occurs if the command is not called with exactly one argument.

RefreshState Reloads the tablet record on the specified tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <RefreshState> command This error occurs if the command is not called with exactly one argument.

RefreshStateByShard Runs 'RefreshState' on all tablets in the given shard.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells whose tablets are included. If empty, all cells are considered.

Arguments

- <keyspace/shard> – Required. The name of a sharded database that contains one or more tables as well as the shard associated with the command. The keyspace must be identified by a string that does not contain whitespace, while the shard is typically identified by a string in the format <range start>-<range end>.

Errors

- the <keyspace/shard> argument is required for the <RefreshStateByShard> command This error occurs if the command is not called with exactly one argument.

RunHealthCheck Runs a health check on a remote tablet.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <tablet alias> argument is required for the <RunHealthCheck> command This error occurs if the command is not called with exactly one argument.

IgnoreHealthError Sets the regexp for health check errors to ignore on the specified tablet. The pattern has implicit ^$ anchors. Set to empty string or restart vttablet to stop ignoring anything.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <ignore regexp> – Required.

Errors

- the <tablet alias> and <ignore regexp> arguments are required for the <IgnoreHealthError> command This error occurs if the command is not called with exactly 2 arguments.

Sleep Blocks the action queue on the specified tablet for the specified amount of time. This is typically used for testing.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <duration> – Required. The amount of time that the action queue should be blocked. The value is a string that contains a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as “300ms” or “1h45m”. See the definition of the Go language’s ParseDuration function for more details. Note that, in practice, the value should be a positively signed value.

Errors

- the <tablet alias> and <duration> arguments are required for the <Sleep> command This error occurs if the command is not called with exactly 2 arguments.

ExecuteHook Runs the specified hook on the given tablet. A hook is a script that resides in the $VTROOT/vthook directory. You can put any script into that directory and use this command to run that script.For this command, the param=value arguments are parameters that the command passes to the specified hook.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <hook name> – Required.
- <param1=value1> <param2=value2> . – Optional.

Errors

- the <tablet alias> and <hook name> arguments are required for the <ExecuteHook> command This error occurs if the command is not called with at least 2 arguments.

```
ExecuteFetchAsApp [-max_rows=10000] [-json] [-use_pool] <tablet alias> <sql command>
```

ExecuteFetchAsDba Runs the given SQL command as a DBA on the remote tablet.

Example

Flags

Name	Type	Definition
disable_binlogs	Boolean	Disables writing to binlogs during the query
json	Boolean	Output JSON instead of human-readable table
max_rows	Int	Specifies the maximum number of rows to allow in reset
reload_schema	Boolean	Indicates whether the tablet schema will be reloaded after executing the SQL command. The default value is false, which indicates that the tablet schema will not be reloaded.

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.
- <sql command> – Required.

Errors

- the <tablet alias> and <sql command> arguments are required for the <ExecuteFetchAsDba> command This error occurs if the command is not called with exactly 2 arguments.

```
VReplicationExec [-json] <tablet alias> <sql command>
```

Backup Stops mysqld and uses the BackupStorage service to store a new backup. This function also remembers if the tablet was replicating so that it can restore the same state after the backup completes.

Example

Flags

Name	Type	Definition
concurrency	Int	Specifies the number of compression/checksum jobs to run simultaneously

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- the <Backup> command requires the <tablet alias> argument This error occurs if the command is not called with exactly one argument.

ChangeSlaveType Changes the db type for the specified tablet, if possible. This command is used primarily to arrange replicas, and it will not convert a master.NOTE: This command automatically updates the serving graph.

Example

Flags

Name	Type	Definition
dry-run	Boolean	Lists the proposed change without actually executing it

RestoreFromBackup Stops mysqld and restores the data from the latest backup.

Example

Errors

- the <RestoreFromBackup> command requires the <tablet alias> argument This error occurs if the command is not called with exactly one argument.

ReparentTablet Reparent a tablet to the current master in the shard. This only works if the current replication position matches the last known reparent action.

Example

Errors

- action <ReparentTablet> requires <tablet alias> This error occurs if the command is not called with exactly one argument.
- active reparent commands disabled (unset the -disable_active_reparents flag to enable)

See Also

- vtctl command index

vtctl Topo Command Reference

series: vtctl ## Topology Service

The following `vtctl` commands are available for administering Topology Services.

Commands

TopoCat Retrieves the file(s) at <path> from the topo service, and displays it. It can resolve wildcards, and decode the proto-encoded data.

Example

Flags

Name	Type	Definition
cell	string	topology cell to cat the file from. Defaults to global cell.
decode_proto	Boolean	decode proto files and display them as text
long	Boolean	long listing.

Arguments

- <cell> – Required. A cell is a location for a service. Generally, a cell resides in only one cluster. In Vitess, the terms "cell" and "data center" are interchangeable. The argument value is a string that does not contain whitespace.
- <path> – Required.
- <path>. – Optional.

Errors

- <TopoCat>: no path specified This error occurs if the command is not called with at least one argument.
- <TopoCat>: invalid wildcards: %v
- <TopoCat>: some paths had errors

```
TopoCp  [-cell <cell>] [-to_topo] <src> <dst>
```

See Also

- vtctl command index

vtctl Workflow Command Reference

series: vtctl ## Workflows

{{< info >}} This covers older vtctl specific commands. For workflows associated with VReplication, you should only use the Workflow commands. {{< /info >}}

The following `vtctl` commands are available for administering workflows.

Commands

WorkflowCreate Creates the workflow with the provided parameters. The workflow is also started, unless -skip_start is specified.

Example

Flags

Name	Type	Definition
skip_start	Boolean	If set, the workflow will not be started.

Arguments

- <factoryName> – Required.

Errors

- the <factoryName> argument is required for the <WorkflowCreate> command This error occurs if the command is not called with at least one argument.
- no workflow.Manager registered

WorkflowStart Starts the workflow.

Example

Errors

- the <uuid> argument is required for the <WorkflowStart> command This error occurs if the command is not called with exactly one argument.
- no workflow.Manager registered

WorkflowStop Stops the workflow.

Example

Errors

- the <uuid> argument is required for the <WorkflowStop> command This error occurs if the command is not called with exactly one argument.
- no workflow.Manager registered

WorkflowDelete Deletes the finished or not started workflow.

Example

Errors

- the <uuid> argument is required for the <WorkflowDelete> command This error occurs if the command is not called with exactly one argument.
- no workflow.Manager registered

```
WorkflowWait  <uuid>
```

WorkflowTree Displays a JSON representation of the workflow tree.

Example

Errors

- the <WorkflowTree> command takes no parameter This error occurs if the command is not called with exactly 0 arguments.
- no workflow.Manager registered

```
WorkflowAction  <path> <name>
```

See Also

- vtctl command index

vtctld

description: The Vitess Admin GUI

`vtctld` is a webserver interface to administer a Vitess cluster. It is usually the first Vitess component to be started after a valid global topology service has been created.

Example Usage

The following example launches the `vtctld` daemon on port 15000:

```
export TOPOLOGY_FLAGS="-topo_implementation etcd2 -topo_global_server_address
   localhost:2379 -topo_global_root /vitess/global"
export VTDATAROOT="/tmp"

vtctld \
 $TOPOLOGY_FLAGS \
 -workflow_manager_init \
 -workflow_manager_use_election \
 -service_map 'grpc-vtctl' \
 -backup_storage_implementation file \
 -file_backup_storage_root $VTDATAROOT/backups \
 -log_dir $VTDATAROOT/tmp \
 -port 15000 \
 -grpc_port 15999
```

Options

Name	Type	Definition
-action_timeout	duration	time to wait for an action before resorting to force (default 2m0s)
-alsologtostderr		log to standard error as well as files
-app_idle_timeout	duration	Idle timeout for app connections (default 1m0s)
-app_pool_size	int	Size of the connection pool for app connections (default 40)
-azblob_backup_account_key_file	string	Path to a file containing the Azure Storage account key; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_KEY will be used as the key itself (NOT a file path)
-azblob_backup_account_name	string	Azure Storage Account name for backups; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_NAME will be used
-azblob_backup_container_name	string	Azure Blob Container Name
-azblob_backup_parallelism	int	Azure Blob operation parallelism (requires extra memory when increased) (default 1)
-azblob_backup_storage_root	string	Root prefix for all backup-related Azure Blobs; this should exclude both initial and trailing '/' (e.g. just 'a/b' not '/a/b/')
-backup_engine_implementation	string	Specifies which implementation to use for creating new backups (builtin or xtrabackup). Restores will always be done with whichever engine created a given backup. (default "builtin")
-backup_storage_block_size	int	if backup_storage_compress is true, backup_storage_block_size sets the byte size for each block while compressing (default is 250000). (default 250000)
-backup_storage_compress	boolean	if set, the backup files will be compressed (default is true). Set to false for instance if a backup_storage_hook is specified and it compresses the data. (default true)
-backup_storage_hook	string	if set, we send the contents of the backup files through this hook.
-backup_storage_implementation	string	which implementation to use for the backup storage feature
-backup_storage_number_blocks	int	if backup_storage_compress is true, backup_storage_number_blocks sets the number of blocks that can be processed, at once, before the writer blocks, during compression (default is 2). It should be equal to the number of CPUs available for compression (default 2)

Name	Type	Definition
-binlog_player_protocol	string	the protocol to download binlogs from a vttablet (default “grpc”)
-binlog_use_v3_resharding_mode	boolean	True iff the binlog streamer should use V3-style sharding, which doesn't require a preset sharding key column. (default true)
-cell	string	cell to use
-ceph_backup_storage_config	string	Path to JSON config file for ceph backup storage (default “ceph_backup_config.json”)
-consul_auth_static_file	string	JSON File to read the topos/tokens from.
-cpu_profile	string	write cpu profile to file
-datadog-agent-host	string	host to send spans to. if empty, no tracing will be done
-datadog-agent-port	string	port to send spans to. if empty, no tracing will be done
-db-credentials-file	string	db credentials file; send SIGHUP to reload this file
-db-credentials-server	string	db credentials server type (use ‘file’ for the file implementation) (default “file”)
-dba_idle_timeout	duration	Idle timeout for dba connections (default 1m0s)
-dba_pool_size	int	Size of the connection pool for dba connections (default 20)
-disable_active_reparents	boolean	if set, do not allow active reparents. Use this to protect a cluster using external reparents.
-discovery_high_replication_lag_minimum_serving	duration	the replication lag that is considered too high when selecting the minimum num vttablets for serving (default 2h0m0s)
-discovery_low_replication_lag	duration	the replication lag that is considered low enough to be healthy (default 30s)
-emit_stats	boolean	true iff we should emit stats to push-based monitoring/stats backends
-enable-consolidator	boolean	This option enables the query consolidator. (default true)
-enable-consolidator-replicas	boolean	This option enables the query consolidator only on replicas.
-enable-query-plan-field-caching	boollean	This option fetches & caches fields (columns) when storing query plans (default true)
-enable-tx-throttler	boolean	If true replication-lag-based throttling on transactions will be enabled.
-enable_hot_row_protection	boolean	If true, incoming transactions for the same row (range) will be queued and cannot consume all txpool slots.
-enable_hot_row_protection_dry_run	boolean	If true, hot row protection is not enforced but logs if transactions would have been queued.
-enable_queries	boolean	if set, allows vtgate and vttablet queries. May have security implications, as the queries will be run from this process.
-enable_realtime_stats	boolean	Required for the Realtime Stats view. If set, vtctld will maintain a streaming RPC to each tablet (in all cells) to gather the realtime health stats.
-enable_transaction_limit	boolean	If true, limit on number of transactions open at the same time will be enforced for all users. User trying to open a new transaction after exhausting their limit will receive an error immediately, regardless of whether there are available slots or not.
-enable_transaction_limit_dry_run	boolean	If true, limit on number of transactions open at the same time will be tracked for all users, but not enforced.
-enforce_strict_trans_tables	boolean	If true, vttablet requires MySQL to run with STRICT_TRANS_TABLES or STRICT_ALL_TABLES on. It is recommended to not turn this flag off. Otherwise MySQL may alter your supplied values before saving them to the database. (default true)
-file_backup_storage_root	string	root directory for the file backup storage
-gcs_backup_storage_bucket	string	Google Cloud Storage bucket to use for backups
-gcs_backup_storage_root	string	root prefix for all backup-related object names
-grpc_auth_mode	string	Which auth plugin implementation to use (eg: static)
-grpc_auth_mtls_allowed_substrings	string	List of substrings of at least one of the client certificate names (separated by colon).
-grpc_auth_static_client_creds	string	when using grpc_static_auth in the server, this file provides the credentials to use to authenticate with server

Name	Type	Definition
-grpc_auth_static_password_file	string	JSON File to read the users/passwords from.
-grpc_ca	string	ca to use, requires TLS, and enforces client cert check
-grpc_cert	string	certificate to use, requires grpc_key, enables TLS
-grpc_compression	string	how to compress gRPC, default: nothing, supported: snappy
-grpc_enable_tracing	boolean	Enable GRPC tracing
-grpc_initial_conn_window_size	int	grpc initial connection window size
-grpc_initial_window_size	int	grpc initial window size
-grpc_keepalive_time	duration	After a duration of this time if the client doesn't see any activity it pings the server to see if the transport is still alive. (default 10s)
-grpc_keepalive_timeout	duration	After having pinged for keepalive check, the client waits for a duration of Timeout and if no activity is seen even after that the connection is closed. (default 10s)
-grpc_key	string	key to use, requires grpc_cert, enables TLS
-grpc_max_connection_age	duration	Maximum age of a client connection before GoAway is sent. (default 2562047h47m16.854775807s)
-grpc_max_connection_age_grace	duration	Additional grace period after grpc_max_connection_age, after which connections are forcibly closed. (default 2562047h47m16.854775807s)
-grpc_max_message_size	int	Maximum allowed RPC message size. Larger messages will be rejected by gRPC with the error 'exceeding the max size'. (default 16777216)
-grpc_port	int	Port to listen on for gRPC calls
-grpc_prometheus	boolean	Enable gRPC monitoring with Prometheus
-grpc_server_initial_conn_window_size	int	grpc server initial connection window size
-grpc_server_initial_window_size	int	grpc server initial window size
-grpc_server_keepalive_enforcement_policy_min_time	duration	grpc server minimum keepalive time (default 5m0s)
-grpc_server_keepalive_enforcement_policy_permit_without_strea	m	grpc server permit client keepalive pings even when there are no active streams (RPCs)
-heartbeat_enable	boolean	If true, vttablet records (if master) or checks (if replica) the current time of a replication heartbeat in the table _vt.heartbeat. The result is used to inform the serving state of the vttablet via healthchecks.
-heartbeat_interval	duration	How frequently to read and write replication heartbeat. (default 1s)
-hot_row_protection_concurrent_transactions	int	Number of concurrent transactions let through to the txpool/MySQL for the same hot row. Should be > 1 to have enough 'ready' transactions in MySQL and benefit from a pipelining effect. (default 5)
-hot_row_protection_max_global_queue_size	int	Global queue limit across all row (ranges). Useful to prevent that the queue can grow unbounded. (default 1000)
-hot_row_protection_max_queue_size	int	Maximum number of BeginExecute RPCs which will be queued for the same row (range). (default 20)
-jaeger-agent-host	string	host and port to send spans to. if empty, no tracing will be done
-keep_logs	duration	keep logs for this long (using ctime) (zero to keep forever)
-keep_logs_by_mtime	duration	keep logs for this long (using mtime) (zero to keep forever)
-lameduck-period	duration	keep running at least this long after SIGTERM before stopping (default 50ms)
-legacy_replication_lag_algorithm	boolean	use the legacy algorithm when selecting the vttablets for serving (default true)
-log_backtrace_at	value	when logging hits line file:N, emit a stack trace
-log_dir	string	If non-empty, write log files in this directory
-log_err_stacks	boolean	log stack traces for errors
-log_rotate_max_size	uint	size in bytes at which logs are rotated (glog.MaxSize) (default 1887436800)
-logtostderr	boolean	log to standard error instead of files

Name	Type	Definition
-master_connect_retry	duration	how long to wait in between replica reconnect attempts. Only precise to the second. (default 10s)
-mem-profile-rate	int	profile every n bytes allocated (default 524288)
-min_number_serving_vttablets	int	the minimum number of vttablets that will be continue to be used even with low replication lag (default 2)
-mutex-profile-fraction	int	profile every n mutex contention events (see runtime.SetMutexProfileFraction)
-mysql_auth_server_static_file	string	JSON File to read the users/passwords from.
-mysql_auth_server_static_string	string	JSON representation of the users/passwords config.
-mysql_auth_static_reload_interval	duration	Ticker to reload credentials
-mysql_clientcert_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default “mysql_clear_password”)
-mysql_server_flush_delay	duration	Delay after which buffered response will flushed to client. (default 100ms)
-mysqlctl_client_protocol	string	the protocol to use to talk to the mysqlctl server (default “grpc”)
-mysqlctl_mycnf_template	string	template file to use for generating the my.cnf file during server init
-mysqlctl_socket	string	socket file to use for remote mysqlctl actions (empty for local actions)
-onterm_timeout	duration	wait no more than this for OnTermSync handlers before stopping (default 10s)
-opentsdb_uri	string	URI of opentsdb /api/put method
-pid_file	string	If set, the process will write its pid to the named file, and delete it on graceful shutdown.
-pool_hostname_resolve_interval	duration	if set force an update to all hostnames and reconnect if changed, defaults to 0 (disabled)
-port	int	port for the server
-proxy_tablets	boolean	Setting this true will make vtctld proxy the tablet status instead of redirecting to them
-purge_logs_interval	duration	how often try to remove old logs (default 1h0m0s)
-query-log-stream-handler	string	URL handler for streaming queries log (default “/debug/querylog”)
-querylog-filter-tag	string	string that must be present in the query as a comment for the query to be logged, works for both vtgate and vttablet
-querylog-format	string	format for query logs (“text” or “json”) (default “text”)
-queryserver-config-acl-exempt-acl	string	an acl that exempt from table acl checking (this acl is free to access any vitess tables).
-queryserver-config-enable-table-acl-dry-run	boolean	If this flag is enabled, tabletserver will emit monitoring metrics and let the request pass regardless of table acl check results
-queryserver-config-idle-timeout	int	query server idle timeout (in seconds), vttablet manages various mysql connection pools. This config means if a connection has not been used in given idle timeout, this connection will be removed from pool. This effectively manages number of connection objects and optimize the pool performance. (default 1800)
-queryserver-config-max-dml-rows	int	query server max dml rows per statement, maximum number of rows allowed to return at a time for an update or delete with either 1) an equality where clauses on primary keys, or 2) a subselect statement. For update and delete statements in above two categories, vttablet will split the original query into multiple small queries based on this configuration value.
-queryserver-config-max-result-size	int	query server max result size, maximum number of rows allowed to return from vttablet for non-streaming queries. (default 10000)
-queryserver-config-message-postpone-cap	int	query server message postpone cap is the maximum number of messages that can be postponed at any given time. Set this number to substantially lower than transaction cap, so that the transaction pool isn’t exhausted by the message subsystem. (default 4)

Name	Type	Definition
-queryserver-config-passthrough-dmls	boolean	query server pass through all dml statements without rewriting
-queryserver-config-pool-prefill-parallelism	int	query server read pool prefill parallelism, a non-zero value will prefill the pool using the specified parallism.
-queryserver-config-pool-size	int	query server read pool size, connection pool is used by regular queries (non streaming, not in a transaction) (default 16)
-queryserver-config-query-cache-size	int	query server query cache size, maximum number of queries to be cached. vttablet analyzes every incoming query and generate a query plan, these plans are being cached in a lru cache. This config controls the capacity of the lru cache. (default 5000)
-queryserver-config-query-pool-timeout	int	query server query pool timeout (in seconds), it is how long vttablet waits for a connection from the query pool. If set to 0 (default) then the overall query timeout is used instead.
-queryserver-config-query-pool-waiter-cap	int	query server query pool waiter limit, this is the maximum number of queries that can be queued waiting to get a connection (default 5000)
-queryserver-config-query-timeout	int	query server query timeout (in seconds), this is the query timeout in vttablet side. If a query takes more than this timeout, it will be killed. (default 30)
-queryserver-config-schema-reload-time	int	query server schema reload time, how often vttablet reloads schemas from underlying MySQL instance in seconds. vttablet keeps table schemas in its own memory and periodically refreshes it from MySQL. This config controls the reload time. (default 1800)
-queryserver-config-stream-buffer-size	int	query server stream buffer size, the maximum number of bytes sent from vttablet for each stream call. It's recommended to keep this value in sync with vtgate's stream_buffer_size. (default 32768)
-queryserver-config-stream-pool-prefill-parallelism	int	query server stream pool prefill parallelism, a non-zero value will prefill the pool using the specified parallelism
-queryserver-config-stream-pool-size	int	query server stream connection pool size, stream pool is used by stream queries: queries that return results to client in a streaming fashion (default 200)
-queryserver-config-strict-table-acl	boolean	only allow queries that pass table acl checks
-queryserver-config-terse-errors	boolean	prevent bind vars from escaping in returned errors
-queryserver-config-transaction-cap	int	query server transaction cap is the maximum number of transactions allowed to happen at any given point of a time for a single vttablet. E.g. by setting transaction cap to 100, there are at most 100 transactions will be processed by a vttablet and the 101th transaction will be blocked (and fail if it cannot get connection within specified timeout) (default 20)
-queryserver-config-transaction-prefill-parallelism	int	query server transaction prefill parallelism, a non-zero value will prefill the pool using the specified parallism.
-queryserver-config-transaction-timeout	int	query server transaction timeout (in seconds), a transaction will be killed if it takes longer than this value (default 30)
-queryserver-config-txpool-timeout	int	query server transaction pool timeout, it is how long vttablet waits if tx pool is full (default 1)
-queryserver-config-txpool-waiter-cap	int	query server transaction pool waiter limit, this is the maximum number of transactions that can be queued waiting to get a connection (default 5000)
-queryserver-config-warn-result-size	int	query server result size warning threshold, warn if number of rows returned from vttablet for non-streaming queries exceeds this
-redact-debug-ui-queries	boolean	redact full queries and bind variables from debug UI
-remote_operation_timeout	duration	time to wait for a remote operation (default 30s)
-s3_backup_aws_endpoint	string	endpoint of the S3 backend (region must be provided)
-s3_backup_aws_region	string	AWS region to use (default "us-east-1")
-s3_backup_aws_retries	int	AWS request retries (default -1)
-s3_backup_force_path_style	boolean	force the s3 path style

Name	Type	Definition
-s3_backup_log_level	string	determine the S3 loglevel to use from LogOff, LogDebug, LogDebugWithSigning, LogDebugWithHTTPBody, LogDebugWithRequestRetries, LogDebugWithRequestErrors (default "LogOff")
-s3_backup_server_side_encryption	string	server-side encryption algorithm (e.g., AES256, aws:kms)
-s3_backup_storage_bucket	string	S3 bucket to use for backups
-s3_backup_storage_root	string	root prefix for all backup-related object names
-s3_backup_tls_skip_verify_cert	boolean	skip the 'certificate is valid' check for SSL connections
-schema_change_check_interval	int	this value decides how often we check schema change dir, in seconds (default 60)
-schema_change_controller	string	schema change controller is responsible for finding schema changes and responding to schema change events
-schema_change_dir	string	directory contains schema changes for all keyspaces. Each keyspace has its own directory and schema changes are expected to live in '$KEYSPACE/input' dir. e.g. test_keyspace/input/*sql, each sql file represents a schema change
-schema_change_replicas_timeout	duration	how long to wait for replicas to receive the schema change (default 10s)
-schema_change_user	string	The user who submits this schema change.
-schema_swap_admin_query_timeout	duration	timeout for SQL queries used to save and retrieve meta information for schema swap process (default 30s)
-schema_swap_backup_concurrency	int	number of simultaneous compression/checksum jobs to run for seed backup during schema swap (default 4)
-schema_swap_delay_between_errors	duration	time to wait after a retryable error happened in the schema swap process (default 1m0s)
-schema_swap_reparent_timeout	duration	timeout to wait for replicas when doing reparent during schema swap (default 30s)
-security_policy	string	the name of a registered security policy to use for controlling access to URLs - empty means allow all for anyone (built-in policies: deny-all, read-only)
-service_map	value	comma separated list of services to enable (or disable if prefixed with '-') Example: grpc-vtworker
-sql-max-length-errors	int	truncate queries in error logs to the given length (default unlimited)
-sql-max-length-ui	int	truncate queries in debug UIs to the given length (default 512) (default 512)
-srv_topo_cache_refresh	duration	how frequently to refresh the topology for cached entries (default 1s)
-srv_topo_cache_ttl	duration	how long to use cached entries for topology (default 1s)
-stats_backend	string	The name of the registered push-based monitoring/stats backend to use
-stats_combine_dimensions	string	List of dimensions to be combined into a single "all" value in exported stats vars
-stats_drop_variables	string	Variables to be dropped from the list of exported variables.
-stats_emit_period	duration	Interval between emitting stats to all registered backends (default 1m0s)
-stderrthreshold	value	logs at or above this threshold go to stderr (default 1)
-tablet_dir	string	The directory within the vtdataroot to store vttablet/mysql files. Defaults to being generated by the tablet uid.
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_health_keep_alive	duration	close streaming tablet health connection if there are no requests for this long (default 5m0s)
-tablet_manager_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_manager_grpc_cert	string	the cert to use to connect
-tablet_manager_grpc_concurrency	int	concurrency to use to talk to a vttablet server for performance-sensitive RPCs (like ExecuteFetchAs{Dba,AllPrivs,App}) (default 8)

Name	Type	Definition
-tablet_manager_grpc_key	string	the key to use to connect
-tablet_manager_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_protocol	string	the protocol to use to talk to vttablet (default “grpc”)
-tablet_protocol	string	how to talk to the vttablets (default “grpc”)
-tablet_url_template	string	format string describing debug tablet url formatting. See the Go code for getTabletDebugURL() how to customize this. (default “http://{{.GetTabletHostPort}}”)
-throttler_client_grpc_ca	string	the server ca to use to validate servers when connecting
-throttler_client_grpc_cert	string	the cert to use to connect
-throttler_client_grpc_key	string	the key to use to connect
-throttler_client_grpc_server_name	string	the server name to use to validate server certificate
-throttler_client_protocol	string	the protocol to use to talk to the integrated throttler service (default “grpc”)
-topo_consul_watch_poll_duration	duration	time of the long poll for watch queries. (default 30s)
-topo_etcd_lease_ttl	int	Lease TTL for locks and master election. The client will use KeepAlive to keep the lease going. (default 30)
-topo_etcd_tls_ca	string	path to the ca to use to validate the server cert when connecting to the etcd topo server
-topo_etcd_tls_cert	string	path to the client cert to use to connect to the etcd topo server, requires topo_etcd_tls_key, enables TLS
-topo_etcd_tls_key	string	path to the client key to use to connect to the etcd topo server, enables TLS
-topo_global_root	string	the path of the global topology data in the global topology server
-topo_global_server_address	string	the address of the global topology server
-topo_implementation	string	the topology implementation to use
-topo_k8s_context	string	The kubeconfig context to use, overrides the ‘current-context’ from the config
-topo_k8s_kubeconfig	string	Path to a valid kubeconfig file.
-topo_k8s_namespace	string	The kubernetes namespace to use for all objects. Default comes from the context or in-cluster config
-topo_zk_auth_file	string	auth to use when connecting to the zk topo server, file contents should be :, e.g., digest:user:pass
-topo_zk_base_timeout	duration	zk base timeout (see zk.Connect) (default 30s)
-topo_zk_max_concurrency	int	maximum number of pending requests to send to a Zookeeper server. (default 64)
-topo_zk_tls_ca	string	the server ca to use to validate servers when connecting to the zk topo server
-topo_zk_tls_cert	string	the cert to use to connect to the zk topo server, requires topo_zk_tls_key, enables TLS
-topo_zk_tls_key	string	the key to use to connect to the zk topo server, enables TLS
-tracer	string	tracing service to use (default “noop”)
-tracing-sampling-rate	float	sampling rate for the probabilistic jaeger sampler (default 0.1)
-transaction-log-stream-handler	string	URL handler for streaming transactions log (default “/debug/txlog”)
-transaction_limit_by_component	boolean	Include CallerID.component when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_principal	boolean	Include CallerID.principal when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_by_subcomponent	boolean	Include CallerID.subcomponent when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_username	boolean	Include VTGateCallerID.username when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_per_user	float	Maximum number of transactions a single user is allowed to use at any time, represented as fraction of -transaction_cap. (default 0.4)
-transaction_shutdown_grace_period	int	how long to wait (in seconds) for transactions to complete during graceful shutdown.

Name	Type	Definition
-twopc_abandon_age	float	time in seconds. Any unresolved transaction older than this time will be sent to the coordinator to be resolved.
-twopc_coordinator_address	string	address of the (VTGate) process(es) that will be used to notify of abandoned transactions.
-twopc_enable	boolean	if the flag is on, 2pc is enabled. Other 2pc flags must be supplied.
-tx-throttler-config	string	The configuration of the transaction throttler as a text formatted throttlerdata.Configuration protocol buffer message (default "target_replication_lag_sec: 2 max_replication_lag_sec: 10 initial_rate: 100 max_increase: 1 emergency_decrease: 0.5 min_duration_between_increases_sec: 40 max_duration_between_increases_sec: 62 min_duration_between_decreases_sec: 20 spread_backlog_across_sec: 20 age_bad_rate_after_sec: 180 bad_rate_increase: 0.1 max_rate_approach_threshold: 0.9")
-tx-throttler-healthcheck-cells	value	A comma-separated list of cells. Only tabletservers running in these cells will be monitored for replication lag by the transaction throttler.
-v	value	log level for V logs
-version		print binary version
-vmodule	value	comma-separated list of pattern=N settings for file-filtered logging
-vreplication_healthcheck_retry_delay	duration	healthcheck retry delay (default 5s)
-vreplication_healthcheck_timeout	duration	healthcheck retry delay (default 1m0s)
-vreplication_healthcheck_topology_refresh	duration	refresh interval for re-reading the topology (default 30s)
-vreplication_retry_delay	duration	delay before retrying a failed binlog connection (default 5s)
-vreplication_tablet_type	string	comma separated list of tablet types used as a source (default "REPLICA")
-vstream_packet_size	int	Suggested packet size for VReplication streamer. This is used only as a recommendation. The actual packet size may be more or less than this amount. (default 30000)
-vtctl_client_protocol	string	the protocol to use to talk to the vtctl server (default "grpc")
-vtctl_healthcheck_retry_delay	duration	delay before retrying a failed healthcheck (default 5s)
-vtctl_healthcheck_timeout	duration	the health check timeout period (default 1m0s)
-vtctl_healthcheck_topology_refresh	duration	refresh interval for re-reading the topology (default 30s)
-vtctld_show_topology_crud	boolean	Controls the display of the CRUD topology actions in the vtctld UI. (default true)
-vtgate_grpc_ca	string	the server ca to use to validate servers when connecting
-vtgate_grpc_cert	string	the cert to use to connect
-vtgate_grpc_key	string	the key to use to connect
-vtgate_grpc_server_name	string	the server name to use to validate server certificate
-vtgate_protocol	string	how to talk to vtgate (default "grpc")
-vtworker_client_grpc_ca	string	the server ca to use to validate servers when connecting
-vtworker_client_grpc_cert	string	the cert to use to connect
-vtworker_client_grpc_key	string	the key to use to connect
-vtworker_client_grpc_server_name	string	the server name to use to validate server certificate
-vtworker_client_protocol	string	the protocol to use to talk to the vtworker server (default "grpc")
-wait_for_drain_sleep_rdonly	duration	time to wait before shutting the query service on old RDONLY tablets during MigrateServedTypes (default 5s)
-wait_for_drain_sleep_replica	duration	time to wait before shutting the query service on old REPLICA tablets during MigrateServedTypes (default 15s)
-watch_replication_stream	boolean	When enabled, vttablet will stream the MySQL replication stream from the local server, and use it to support the include_event_token ExecuteOptions.

Name	Type	Definition
-workflow_manager_disable	value	comma separated list of workflow types to disable
-workflow_manager_init	boolean	Initialize the workflow manager in this vtctld instance.
-workflow_manager_use_election	boolean	if specified, will use a topology server-based master election to ensure only one workflow manager is active at a time.
-xbstream_restore_flags	string	flags to pass to xbstream command during restore. These should be space separated and will be added to the end of the command. These need to match the ones used for backup e.g. –compress / –decompress, –encrypt / –decrypt
-xtrabackup_backup_flags	string	flags to pass to backup command. These should be space separated and will be added to the end of the command
-xtrabackup_prepare_flags	string	flags to pass to prepare command. These should be space separated and will be added to the end of the command
-xtrabackup_root_path	string	directory location of the xtrabackup executable, e.g., /usr/bin
-xtrabackup_stream_mode	string	which mode to use if streaming, valid values are tar and xbstream (default “tar”)
-xtrabackup_stripe_block_size	uint	Size in bytes of each block that gets sent to a given stripe before rotating to the next stripe (default 102400)
-xtrabackup_stripes	uint	If greater than 0, use data striping across this many destination files to parallelize data transfer and decompression
-xtrabackup_user	string	User that xtrabackup will use to connect to the database server. This user must have all necessary privileges. For details, please refer to xtrabackup documentation.

vtexplain

`vtexplain` is a command line tool which provides information on how Vitess plans to execute a particular query. It can be used to validate queries for compatibility with Vitess.

For a user guide that describes how to use the `vtexplain` tool to explain how Vitess executes a particular SQL statement, see Analyzing a SQL statement.

Example Usage

Explain how Vitess will execute the query `SELECT * FROM users` using the VSchema contained in `vschemas.json` and the database schema `schema.sql`:

```
vtexplain -vschema-file vschema.json -schema-file schema.sql  -sql "SELECT * FROM users"
```

Explain how the example will execute on 128 shards using Row-based replication:

```
vtexplain -shards 128 -vschema-file vschema.json -schema-file schema.sql -replication-mode
   "ROW" -output-mode text -sql "INSERT INTO users (user_id, name) VALUES(1, 'john')"
```

Options

The following parameters apply to `mysqlctl`:

Name	Type	Definition
-output-mode	string	Output in human-friendly text or json (default “text”)
-normalize		Whether to enable vtgate normalization (default false)
-shards	int	Number of shards per keyspace (default 2)
-replication-mode	string	The replication mode to simulate – must be set to either ROW or STATEMENT (default “ROW”)
-schema	string	The SQL table schema (default ””)
-schema-file	string	Identifies the file that contains the SQL table schema (default ””)
-sql	string	A list of semicolon-delimited SQL commands to analyze (default ””)
-sql-file	string	Identifies the file that contains the SQL commands to analyze (default ””)
-vschema	string	Identifies the VTGate routing schema (default ””)
-vschema-file	string	Identifies the VTGate routing schema file (default ””)
-ks-shard-map	string	Identifies the shard keyranges for unevenly-sharded keyspaces (default ””)
-ks-shard-map-file	string	Identifies the shard keyranges file (default ””)
-dbname	string	Optional database target to override normal routing (default ””)
-queryserver-config-passthrough-dmls		query server pass through all dml statements without rewriting (default false)

Please note that `-ks-shard-map` and `ks-shard-map-file` will supercede `-shards`. If you attempt to `vtexplain` on a keyspace that is included in the keyspace shard map, the shards as defined in the mapping will be used and `-shards` will be ignored.

Limitations

The VSchema must use a keyspace name. VTExplain requires a keyspace name for each keyspace in an input VSChema:

```
"keyspace_name": {
    "_comment": "Keyspace definition goes here."
}
```

If no keyspace name is present, VTExplain will return the following error:

```
ERROR: initVtgateExecutor: json: cannot unmarshal bool into Go value of type
   map[string]json.RawMessage
```

vtgate

VTGate is a stateless proxy responsible for accepting requests from applications and routing them to the appropriate tablet server(s) for query execution. It speaks both the MySQL Protocol and a gRPC protocol.

Example Usage

Start a vtgate proxy:

```
export TOPOLOGY_FLAGS="-topo_implementation etcd2 -topo_global_server_address
   localhost:2379 -topo_global_root /vitess/global"
export VTDATAROOT="/tmp"

vtgate \
  $TOPOLOGY_FLAGS \
  -log_dir $VTDATAROOT/tmp \
  -port 15001 \
```

```
-grpc_port 15991 \
-mysql_server_port 15306 \
-cell test \
-cells_to_watch test \
-tablet_types_to_wait MASTER,REPLICA \
-gateway_implementation discoverygateway \
-service_map 'grpc-vtgateservice' \
-pid_file $VTDATAROOT/tmp/vtgate.pid \
-mysql_auth_server_impl none
```

Options

The following global options apply to `vtgate`:

Name	Type	Definition
-allowed_tablet_types	value	Specifies the tablet types this vtgate is allowed to route queries to
-alsologtostderr	boolean	log to standard error as well as files
-buffer_drain_concurrency	int	Maximum number of requests retried simultaneously. More concurrency will increase the load on the MASTER vttablet when draining the buffer. (default 1)
-buffer_keyspace_shards	string	If not empty, limit buffering to these entries (comma separated). Entry format: keyspace or keyspace/shard. Requires –enable_buffer=true.
-buffer_max_failover_duration	duration	Stop buffering completely if a failover takes longer than this duration. (default 20s)
-buffer_min_time_between_failovers	duration	Minimum time between the end of a failover and the start of the next one (tracked per shard). Faster consecutive failovers will not trigger buffering. (default 1m0s)
-buffer_size	int	Maximum number of buffered requests in flight (across all ongoing failovers). (default 10)
-buffer_window	duration	Duration for how long a request should be buffered at most. (default 10s)
-cell	string	cell to use (default “test_nj”)
-cells_to_watch	string	comma-separated list of cells for watching tablets
-consul_auth_static_file	string	JSON File to read the topos/tokens from.
-cpu_profile	string	write cpu profile to file
-datadog-agent-host	string	host to send spans to. if empty, no tracing will be done
-datadog-agent-port	string	port to send spans to. if empty, no tracing will be done
-ddl_strategy	string	Set default strategy for DDL statements. Override with @@ddl_strategy session variable.
-default_tablet_type	value	The default tablet type to set for queries, when one is not explicitly selected (default MASTER)
-discovery_high_replication_lag_minimum_serving	duration	the replication lag that is considered too high when selecting the minimum num vttablets for serving (default 2h0m0s)
-discovery_low_replication_lag	duration	the replication lag that is considered low enough to be healthy (default 30s)
-emit_stats	boolean	true iff we should emit stats to push-based monitoring/stats backends
-enable_buffer	boolean	Enable buffering (stalling) of master traffic during failovers.
-enable_buffer_dry_run	boolean	Detect and log failover events, but do not actually buffer requests.
-enable_system_settings	boolean	Enables the system settings to be changed per session at the database connection level. Override with @@enable_system_settings session variable.
-gate_query_cache_size	int	gate server query cache size, maximum number of queries to be cached. vtgate analyzes every incoming query and generate a query plan, these plans are being cached in a lru cache. This config controls the capacity of the lru cache. (default 10000)
-gateway_implementation	string	The implementation of gateway (default “discoverygateway”)

Name	Type	Definition
-gateway_initial_tablet_timeout	duration	At startup, the gateway will wait up to that duration to get one tablet per keyspace/shard/tablettype (default 30s)
-grpc_auth_mode	string	Which auth plugin implementation to use (eg: static)
-grpc_auth_mtls_allowed_substrings	string	List of substrings of at least one of the client certificate names (separated by colon).
-grpc_auth_static_client_creds	string	when using grpc_static_auth in the server, this file provides the credentials to use to authenticate with server
-grpc_auth_static_password_file	string	JSON File to read the users/passwords from.
-grpc_ca	string	ca to use, requires TLS, and enforces client cert check
-grpc_cert	string	certificate to use, requires grpc_key, enables TLS
-grpc_compression	string	how to compress gRPC, default: nothing, supported: snappy
-grpc_enable_tracing	boolean	Enable GRPC tracing
-grpc_initial_conn_window_size	int	grpc initial connection window size
-grpc_initial_window_size	int	grpc initial window size
-grpc_keepalive_time	duration	After a duration of this time if the client doesn't see any activity it pings the server to see if the transport is still alive. (default 10s)
-grpc_keepalive_timeout	duration	After having pinged for keepalive check, the client waits for a duration of Timeout and if no activity is seen even after that the connection is closed. (default 10s)
-grpc_key	string	key to use, requires grpc_cert, enables TLS
-grpc_max_connection_age	duration	Maximum age of a client connection before GoAway is sent. (default 2562047h47m16.854775807s)
-grpc_max_connection_age_grace	duration	Additional grace period after grpc_max_connection_age, after which connections are forcibly closed. (default 2562047h47m16.854775807s)
-grpc_max_message_size	int	Maximum allowed RPC message size. Larger messages will be rejected by gRPC with the error 'exceeding the max size'. (default 16777216)
-grpc_port	int	Port to listen on for gRPC calls
-grpc_prometheus	boolean	Enable gRPC monitoring with Prometheus
-grpc_server_initial_conn_window_size	int	grpc server initial connection window size
-grpc_server_initial_window_size	int	grpc server initial window size
-grpc_server_keepalive_enforcement_policy_min_time	duration	grpc server minimum keepalive time (default 5m0s)
-grpc_server_keepalive_enforcement_policy_permit_without_stream	boolean	grpc server permit client keepalive pings even when there are no active streams (RPCs)
-grpc_use_effective_callerid	boolean	If set, and SSL is not used, will set the immediate caller id from the effective caller id's principal.
-healthcheck_retry_delay	duration	health check retry delay (default 2ms)
-healthcheck_timeout	duration	the health check timeout period (default 1m0s)
-jaeger-agent-host	string	host and port to send spans to. if empty, no tracing will be done
-keep_logs	duration	keep logs for this long (using ctime) (zero to keep forever)
-keep_logs_by_mtime	duration	keep logs for this long (using mtime) (zero to keep forever)
-keyspaces_to_watch	value	Specifies which keyspaces this vtgate should have access to while routing queries or accessing the vschema
-lameduck-period	duration	keep running at least this long after SIGTERM before stopping (default 50ms)
-legacy_replication_lag_algorithm	boolean	use the legacy algorithm when selecting the vttablets for serving (default true)
-log_backtrace_at	value	when logging hits line file:N, emit a stack trace
-lock_heartbeat_time	duration	If there is lock function used. This will keep the lock connection active by using this heartbeat. (default 5 seconds)
-log_dir	string	If non-empty, write log files in this directory

Name	Type	Definition
-log_err_stacks	boolean	log stack traces for errors
-log_queries_to_file	string	Enable query logging to the specified file
-log_rotate_max_size	uint	size in bytes at which logs are rotated (glog.MaxSize) (default 1887436800)
-logtostderr	boolean	log to standard error instead of files
-max_memory_rows	int	Maximum number of rows that will be held in memory for intermediate results as well as the final result. (default 300000)
-max_payload_size	int	The threshold for query payloads in bytes. A payload greater than this threshold will result in a failure to handle the query.
-mem-profile-rate	int	profile every n bytes allocated (default 524288)
-message_stream_grace_period	duration	the amount of time to give for a vttablet to resume if it ends a message stream, usually because of a reparent. (default 30s)
-min_number_serving_vttablets	int	the minimum number of vttablets that will be continue to be used even with low replication lag (default 2)
-mutex-profile-fraction	int	profile every n mutex contention events (see runtime.SetMutexProfileFraction)
-mysql_allow_clear_text_without_tls	boolean	If set, the server will allow the use of a clear text password over non-SSL connections.
-mysql_auth_server_impl	string	Which auth server implementation to use. (default “static”)
-mysql_auth_server_static_file	string	JSON File to read the users/passwords from.
-mysql_auth_server_static_string	string	JSON representation of the users/passwords config.
-mysql_auth_static_reload_interval	duration	Ticker to reload credentials
-mysql_clientcert_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default “mysql_clear_password”)
-mysql_default_workload	string	Default session workload (OLTP, OLAP, DBA) (default “UNSPECIFIED”)
-mysql_ldap_auth_config_file	string	JSON File from which to read LDAP server config.
-mysql_ldap_auth_config_string	string	JSON representation of LDAP server config.
-mysql_ldap_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default “mysql_clear_password”)
-mysql_server_bind_address	string	Binds on this address when listening to MySQL binary protocol. Useful to restrict listening to ‘localhost’ only for instance.
-mysql_server_flush_delay	duration	Delay after which buffered response will flushed to client. (default 100ms)
-mysql_server_port	int	If set, also listen for MySQL binary protocol connections on this port. (default -1)
-mysql_server_query_timeout	duration	mysql query timeout
-mysql_server_read_timeout	duration	connection read timeout
-mysql_server_require_secure_transport	boolean	Reject insecure connections but only if mysql_server_ssl_cert and mysql_server_ssl_key are provided
-mysql_server_socket_path	string	This option specifies the Unix socket file to use when listening for local connections. By default it will be empty and it won’t listen to a unix socket
-mysql_server_ssl_ca	string	Path to ssl CA for mysql server plugin SSL. If specified, server will require and validate client certs.
-mysql_server_ssl_cert	string	Path to the ssl cert for mysql server plugin SSL
-mysql_server_ssl_key	string	Path to ssl key for mysql server plugin SSL
-mysql_server_version	string	MySQL server version to advertise. (default “5.7.9-Vitess”)
-mysql_server_write_timeout	duration	connection write timeout
-mysql_slow_connect_warn_threshold	duration	Warn if it takes more than the given threshold for a mysql connection to establish
-mysql_tcp_version	string	Select tcp, tcp4, or tcp6 to control the socket type. (default “tcp”)
-normalize_queries	boolean	Rewrite queries with bind vars. Turn this off if the app itself sends normalized queries with bind vars. (default true)

Name	Type	Definition
-onterm_timeout	duration	wait no more than this for OnTermSync handlers before stopping (default 10s)
-opentsdb_uri	string	URI of opentsdb /api/put method
-pid_file	string	If set, the process will write its pid to the named file, and delete it on graceful shutdown.
-port	int	port for the server
-proxy_protocol	boolean	Enable HAProxy PROXY protocol on MySQL listener socket
-purge_logs_interval	duration	how often try to remove old logs (default 1h0m0s)
-querylog-filter-tag	string	string that must be present in the query as a comment for the query to be logged, works for both vtgate and vttablet
-querylog-format	string	format for query logs (“text” or “json”) (default “text”)
-redact-debug-ui-queries	boolean	redact full queries and bind variables from debug UI
-remote_operation_timeout	duration	time to wait for a remote operation (default 30s)
-retry-count	int	retry count (default 2)
-schema_change_signal	boolean	enable schema tracking
-security_policy	string	the name of a registered security policy to use for controlling access to URLs - empty means allow all for anyone (built-in policies: deny-all, read-only)
-service_map	value	comma separated list of services to enable (or disable if prefixed with ‘-’) Example: grpc-vtworker
-sql-max-length-errors	int	truncate queries in error logs to the given length (default unlimited)
-sql-max-length-ui	int	truncate queries in debug UIs to the given length (default 512) (default 512)
-srv_topo_cache_refresh	duration	how frequently to refresh the topology for cached entries (default 1s)
-srv_topo_cache_ttl	duration	how long to use cached entries for topology (default 1s)
-stats_backend	string	The name of the registered push-based monitoring/stats backend to use
-stats_combine_dimensions	string	List of dimensions to be combined into a single “all” value in exported stats vars
-stats_drop_variables	string	Variables to be dropped from the list of exported variables.
-stats_emit_period	duration	Interval between emitting stats to all registered backends (default 1m0s)
-stderrthreshold	value	logs at or above this threshold go to stderr (default 1)
-stream_buffer_size	int	the number of bytes sent from vtgate for each stream call. It’s recommended to keep this value in sync with vttablet’s query-server-config-stream-buffer-size. (default 32768)
-tablet_filters	value	Specifies a comma-separated list of ’keyspace
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_protocol	string	how to talk to the vttablets (default “grpc”)
-tablet_refresh_interval	duration	tablet refresh interval (default 1m0s)
-tablet_refresh_known_tablets	boolean	tablet refresh reloads the tablet address/port map from topo in case it changes (default true)
-tablet_types_to_wait	string	wait till connected for specified tablet types during Gateway initialization
-tablet_url_template	string	format string describing debug tablet url formatting. See the Go code for getTabletDebugURL() how to customize this. (default “http://{{.GetTabletHostPort}}”)
-topo_consul_watch_poll_duration	duration	time of the long poll for watch queries. (default 30s)
-topo_etcd_lease_ttl	int	Lease TTL for locks and master election. The client will use KeepAlive to keep the lease going. (default 30)
-topo_etcd_tls_ca	string	path to the ca to use to validate the server cert when connecting to the etcd topo server
-topo_etcd_tls_cert	string	path to the client cert to use to connect to the etcd topo server, requires topo_etcd_tls_key, enables TLS
-topo_etcd_tls_key	string	path to the client key to use to connect to the etcd topo server, enables TLS
-topo_global_root	string	the path of the global topology data in the global topology server

Name	Type	Definition
-topo_global_server_address	string	the address of the global topology server
-topo_implementation	string	the topology implementation to use
-topo_k8s_context	string	The kubeconfig context to use, overrides the 'current-context' from the config
-topo_k8s_kubeconfig	string	Path to a valid kubeconfig file.
-topo_k8s_namespace	string	The kubernetes namespace to use for all objects. Default comes from the context or in-cluster config
-topo_read_concurrency	int	concurrent topo reads (default 32)
-topo_zk_auth_file	string	auth to use when connecting to the zk topo server, file contents should be :, e.g., digest:user:pass
-topo_zk_base_timeout	duration	zk base timeout (see zk.Connect) (default 30s)
-topo_zk_max_concurrency	int	maximum number of pending requests to send to a Zookeeper server. (default 64)
-topo_zk_tls_ca	string	the server ca to use to validate servers when connecting to the zk topo server
-topo_zk_tls_cert	string	the cert to use to connect to the zk topo server, requires topo_zk_tls_key, enables TLS
-topo_zk_tls_key	string	the key to use to connect to the zk topo server, enables TLS
-tracer	string	tracing service to use (default "noop")
-tracing-sampling-rate	float	sampling rate for the probabilistic jaeger sampler (default 0.1)
-transaction_mode	string	SINGLE: disallow multi-db transactions, MULTI: allow multi-db transactions with best effort commit, TWOPC: allow multi-db transactions with 2pc commit (default "MULTI")
-v	value	log level for V logs
-version	boolean	print binary version
-vmodule	value	comma-separated list of pattern=N settings for file-filtered logging
-vschema_ddl_authorized_users	string	List of users authorized to execute vschema ddl operations, or '%' to allow all users.
-vtctld_addr	string	address of a vtctld instance
-vtgate-config-terse-errors	boolean	prevent bind vars from escaping in returned errors
-warn_memory_rows	int	Warning threshold for in-memory results. A row count higher than this amount will cause the VtGateWarnings.ResultsExceeded counter to be incremented. (default 30000)
-warn_payload_size	int	The warning threshold for query payloads in bytes. A payload greater than this threshold will cause the VtGateWarnings.WarnPayloadSizeExceeded counter to be incremented.

vttablet

A VTTablet server *controls* a running MySQL server. VTTablet supports two primary types of deployments:

- Managed MySQL (most common)
- Unmanaged or Remote MySQL

In addition to these deployment types, a partially managed VTTablet is also possible by setting `-disable_active_reparents`.

Example Usage

Managed MySQL In this mode, Vitess actively manages MySQL:

```
export TOPOLOGY_FLAGS="-topo_implementation etcd2 -topo_global_server_address
   localhost:2379 -topo_global_root /vitess/global"
export VTDATAROOT="/tmp"
```

```
vttablet \
$TOPOLOGY_FLAGS
-tablet-path $alias
-init_keyspace $keyspace
-init_shard $shard
-init_tablet_type $tablet_type
-port $port
-grpc_port $grpc_port
-service_map grpc-queryservice,grpc-tabletmanager,grpc-updatestream
```

`$alias` needs to be of the form: `<cell>-id`, and the cell should match one of the local cells that was created in the topology. The id can be left padded with zeroes: `cell-100` and `cell-000000100` are synonymous.

Unmanaged or Remote MySQL In this mode, an external MySQL can be used such as AWS RDS, AWS Aurora, Google CloudSQL; or just an existing (vanilla) MySQL installation.

See Unmanaged Tablet for the full guide.

Partially managed MySQL Even if a MySQL is remote, you can still make vttablet perform some management functions. They are as follows:

- `-disable_active_reparents`: If this flag is set, then any reparent or replica commands will not be allowed. These are InitShardMaster, PlannedReparent, PlannedReparent, EmergencyReparent, and ReparentTablet. In this mode, you should use the TabletExternallyReparented command to inform vitess of the current master.
- `-master_connect_retry`: This value is give to mysql when it connects a replica to the master as the retry duration parameter.
- `-enable_replication_reporter`: If this flag is set, then vttablet will transmit replica lag related information to the vtgates, which will allow it to balance load better. Additionally, enabling this will also cause vttablet to restart replication if it was stopped. However, it will do this only if -disable_active_reparents was not turned on.
- `-enable_semi_sync`: This option will automatically enable semi-sync on new replicas as well as on any tablet that transitions into a replica type. This includes the demotion of a master to a replica.
- `-heartbeat_enable` and `-heartbeat interval duration`: cause vttablet to write heartbeats to the sidecar database. This information is also used by the replication reporter to assess replica lag.

Options

The following global options apply to `vttablet`:

Name	Type	Definition
-alsologtostderr	boolean	log to standard error as well as files
-app_idle_timeout	duration	Idle timeout for app connections (default 1m0s)
-app_pool_size	int	Size of the connection pool for app connections (default 40)
-azblob_backup_account_key_file	string	Path to a file containing the Azure Storage account key; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_KEY will be used as the key itself (NOT a file path)
-azblob_backup_account_name	string	Azure Storage Account name for backups; if this flag is unset, the environment variable VT_AZBLOB_ACCOUNT_NAME will be used
-azblob_backup_container_name	string	Azure Blob Container Name
-azblob_backup_parallelism	int	Azure Blob operation parallelism (requires extra memory when increased) (default 1)
-azblob_backup_storage_root	string	Root prefix for all backup-related Azure Blobs; this should exclude both initial and trailing '/' (e.g. just 'a/b' not '/a/b/')

Name	Type	Definition
-backup_engine_implementation	string	Specifies which implementation to use for creating new backups (builtin or xtrabackup). Restores will always be done with whichever engine created a given backup. (default "builtin")
-backup_storage_block_size	int	if backup_storage_compress is true, backup_storage_block_size sets the byte size for each block while compressing (default is 250000). (default 250000)
-backup_storage_compress		if set, the backup files will be compressed (default is true). Set to false for instance if a backup_storage_hook is specified and it compresses the data. (default true)
-backup_storage_hook	string	if set, we send the contents of the backup files through this hook.
-backup_storage_implementation	string	which implementation to use for the backup storage feature
-backup_storage_number_blocks	int	if backup_storage_compress is true, backup_storage_number_blocks sets the number of blocks that can be processed, at once, before the writer blocks, during compression (default is 2). It should be equal to the number of CPUs available for compression (default 2)
-binlog_player_grpc_ca	string	the server ca to use to validate servers when connecting
-binlog_player_grpc_cert	string	the cert to use to connect
-binlog_player_grpc_key	string	the key to use to connect
-binlog_player_grpc_server_name	string	the server name to use to validate server certificate
-binlog_player_protocol	string	the protocol to download binlogs from a vttablet (default "grpc")
-binlog_use_v3_resharding_mode		True iff the binlog streamer should use V3-style sharding, which doesn't require a preset sharding key column. (default true)
-ceph_backup_storage_config	string	Path to JSON config file for ceph backup storage (default "ceph_backup_config.json")
-consul_auth_static_file	string	JSON File to read the topos/tokens from.
-cpu_profile	string	write cpu profile to file
-datadog-agent-host	string	host to send spans to. if empty, no tracing will be done
-datadog-agent-port	string	port to send spans to. if empty, no tracing will be done
-db-credentials-file	string	db credentials file; send SIGHUP to reload this file
-db-credentials-server	string	db credentials server type (use 'file' for the file implementation) (default "file")
-db_allprivs_password	string	db allprivs password
-db_allprivs_use_ssl		Set this flag to false to make the allprivs connection to not use ssl (default true)
-db_allprivs_user	string	db allprivs user userKey (default "vt_allprivs")
-db_app_password	string	db app password
-db_app_use_ssl		Set this flag to false to make the app connection to not use ssl (default true)
-db_app_user	string	db app user userKey (default "vt_app")
-db_appdebug_password	string	db appdebug password
-db_appdebug_use_ssl		Set this flag to false to make the appdebug connection to not use ssl (default true)
-db_appdebug_user	string	db appdebug user userKey (default "vt_appdebug")
-db_charset	string	Character set. Only utf8 or latin1 based character sets are supported.
-db_connect_timeout_ms	int	connection timeout to mysqld in milliseconds (0 for no timeout)
-db_dba_password	string	db dba password
-db_dba_use_ssl		Set this flag to false to make the dba connection to not use ssl (default true)
-db_dba_user	string	db dba user userKey (default "vt_dba")
-db_erepl_password	string	db erepl password
-db_erepl_use_ssl		Set this flag to false to make the erepl connection to not use ssl (default true)
-db_erepl_user	string	db erepl user userKey (default "vt_erepl")
-db_filtered_password	string	db filtered password

Name	Type	Definition
-db_filtered_use_ssl		Set this flag to false to make the filtered connection to not use ssl (default true)
-db_filtered_user	string	db filtered user userKey (default “vt_filtered”)
-db_flags	uint	Flag values as defined by MySQL.
-db_flavor	string	Flavor overrid. Valid value is FilePos.
-db_host	string	The host name for the tcp connection.
-db_port	int	tcp port
-db_repl_password	string	db repl password
-db_repl_use_ssl		Set this flag to false to make the repl connection to not use ssl (default true)
-db_repl_user	string	db repl user userKey (default “vt_repl”)
-db_server_name	string	server name of the DB we are connecting to.
-db_socket	string	The unix socket to connect on. If this is specified, host and port will not be used.
-db_ssl_ca	string	connection ssl ca
-db_ssl_ca_path	string	connection ssl ca path
-db_ssl_cert	string	connection ssl certificate
-db_ssl_key	string	connection ssl key
-dba_idle_timeout	duration	Idle timeout for dba connections (default 1m0s)
-dba_pool_size	int	Size of the connection pool for dba connections (default 20)
-degraded_threshold	duration	replication lag after which a replica is considered degraded (only used in status UI) (default 30s)
-demote_master_type	string	the tablet type a demoted master will transition to (default “REPLICA”)
-disable_active_reparents		if set, do not allow active reparents. Use this to protect a cluster using external reparents.
-discovery_high_replication_lag_minimum_serving	duration	the replication lag that is considered too high when selecting the minimum num vttablets for serving (default 2h0m0s)
-discovery_low_replication_lag	duration	the replication lag that is considered low enough to be healthy (default 30s)
-emit_stats		true iff we should emit stats to push-based monitoring/stats backends
-enable-consolidator		This option enables the query consolidator. (default true)
-enable-consolidator-replicas		This option enables the query consolidator only on replicas.
-enable-query-plan-field-caching		This option fetches & caches fields (columns) when storing query plans (default true)
-enable-tx-throttler		If true replication-lag-based throttling on transactions will be enabled.
-enable_hot_row_protection		If true, incoming transactions for the same row (range) will be queued and cannot consume all txpool slots.
-enable_hot_row_protection_dry_run		If true, hot row protection is not enforced but logs if transactions would have been queued.
-enable_replication_reporter		Register the health check module that monitors MySQL replication
-enable_semi_sync		Enable semi-sync when configuring replication, on master and replica tablets only (rdonly tablets will not ack).
-enable_transaction_limit		If true, limit on number of transactions open at the same time will be enforced for all users. User trying to open a new transaction after exhausting their limit will receive an error immediately, regardless of whether there are available slots or not.
-enable_transaction_limit_dry_run		If true, limit on number of transactions open at the same time will be tracked for all users, but not enforced.
-enforce-tableacl-config		if this flag is true, vttablet will fail to start if a valid tableacl config does not exist
-enforce_strict_trans_tables		If true, vttablet requires MySQL to run with STRICT_TRANS_TABLES or STRICT_ALL_TABLES on. It is recommended to not turn this flag off. Otherwise MySQL may alter your supplied values before saving them to the database. (default true)
-file_backup_storage_root	string	root directory for the file backup storage
-filecustomrules	string	file based custom rule path

Name	Type	Definition
-finalize_external_reparent_timeout	duration	Timeout for the finalize stage of a fast external reparent reconciliation. (default 30s)
-gcs_backup_storage_bucket	string	Google Cloud Storage bucket to use for backups
-gcs_backup_storage_root	string	root prefix for all backup-related object names
-grpc_auth_mode	string	Which auth plugin implementation to use (eg: static)
-grpc_auth_mtls_allowed_substrings	string	List of substrings of at least one of the client certificate names (separated by colon).
-grpc_auth_static_client_creds	string	when using grpc_static_auth in the server, this file provides the credentials to use to authenticate with server
-grpc_auth_static_password_file	string	JSON File to read the users/passwords from.
-grpc_ca	string	ca to use, requires TLS, and enforces client cert check
-grpc_cert	string	certificate to use, requires grpc_key, enables TLS
-grpc_compression	string	how to compress gRPC, default: nothing, supported: snappy
-grpc_enable_tracing		Enable GRPC tracing
-grpc_initial_conn_window_size	int	grpc initial connection window size
-grpc_initial_window_size	int	grpc initial window size
-grpc_keepalive_time	duration	After a duration of this time if the client doesn't see any activity it pings the server to see if the transport is still alive. (default 10s)
-grpc_keepalive_timeout	duration	After having pinged for keepalive check, the client waits for a duration of Timeout and if no activity is seen even after that the connection is closed. (default 10s)
-grpc_key	string	key to use, requires grpc_cert, enables TLS
-grpc_max_connection_age	duration	Maximum age of a client connection before GoAway is sent. (default 2562047h47m16.854775807s)
-grpc_max_connection_age_grace	duration	Additional grace period after grpc_max_connection_age, after which connections are forcibly closed. (default 2562047h47m16.854775807s)
-grpc_max_message_size	int	Maximum allowed RPC message size. Larger messages will be rejected by gRPC with the error 'exceeding the max size'. (default 16777216)
-grpc_port	int	Port to listen on for gRPC calls
-grpc_prometheus		Enable gRPC monitoring with Prometheus
-grpc_server_initial_conn_window_size	int	grpc server initial connection window size
-grpc_server_initial_window_size	int	grpc server initial window size
-grpc_server_keepalive_enforcement_policy_min_time	duration	grpc server minimum keepalive time (default 5m0s)
-grpc_server_keepalive_enforcement_policy_permit_without_stream		grpc server permit client keepalive pings even when there are no active streams (RPCs)
-health_check_interval	duration	Interval between health checks (default 20s)
-heartbeat_enable		If true, vttablet records (if master) or checks (if replica) the current time of a replication heartbeat in the table _vt.heartbeat. The result is used to inform the serving state of the vttablet via healthchecks.
-heartbeat_interval	duration	How frequently to read and write replication heartbeat. (default 1s)
-hot_row_protection_concurrent_transactions	int	Number of concurrent transactions let through to the txpool/MySQL for the same hot row. Should be > 1 to have enough 'ready' transactions in MySQL and benefit from a pipelining effect. (default 5)
-hot_row_protection_max_global_queue_size	int	Global queue limit across all row (ranges). Useful to prevent that the queue can grow unbounded. (default 1000)
-hot_row_protection_max_queue_size	int	Maximum number of BeginExecute RPCs which will be queued for the same row (range). (default 20)
-init_db_name_override	string	(init parameter) override the name of the db used by vttablet. Without this flag, the db name defaults to vt_
-init_keyspace	string	(init parameter) keyspace to use for this tablet

Name	Type	Definition
-init_populate_metadata		(init parameter) populate metadata tables even if restore_from_backup is disabled. If restore_from_backup is enabled, metadata tables are always populated regardless of this flag.
-init_shard	string	(init parameter) shard to use for this tablet
-init_tablet_type	string	(init parameter) the tablet type to use for this tablet.
-init_tags	value	(init parameter) comma separated list of key:value pairs used to tag the tablet
-init_timeout	duration	(init parameter) timeout to use for the init phase. (default 1m0s)
-jaeger-agent-host	string	host and port to send spans to. If empty, no tracing will be done
-keep_logs	duration	keep logs for this long (using ctime) (zero to keep forever). Note ctime here means "change time" and can be essentially equivalent to mtime. Read #5318
-keep_logs_by_mtime	duration	keep logs for this long (using mtime) (zero to keep forever)
-lameduck-period	duration	keep running at least this long after SIGTERM before stopping (default 50ms)
-legacy_replication_lag_algorithm		use the legacy algorithm when selecting the vttablets for serving (default true)
-lock_tables_timeout	duration	How long to keep the table locked before timing out (default 1m0s)
-log_backtrace_at	value	when logging hits line file:N, emit a stack trace
-log_dir	string	If non-empty, write log files in this directory
-log_err_stacks		log stack traces for errors
-log_queries		Enable query logging to syslog.
-log_queries_to_file	string	Enable query logging to the specified file
-log_rotate_max_size	uint	size in bytes at which logs are rotated (glog.MaxSize) (default 1887436800)
-logtostderr		log to standard error instead of files
-master_connect_retry	duration	how long to wait in between replica reconnect attempts. Only precise to the second. (default 10s)
-mem-profile-rate	int	profile every n bytes allocated (default 524288)
-min_number_serving_vttablets	int	the minimum number of vttablets that will be continue to be used even with low replication lag (default 2)
-mutex-profile-fraction	int	profile every n mutex contention events (see runtime.SetMutexProfileFraction)
-mycnf-file	string	path to my.cnf, if reading all config params from there
-mycnf_bin_log_path	string	mysql binlog path
-mycnf_data_dir	string	data directory for mysql
-mycnf_error_log_path	string	mysql error log path
-mycnf_general_log_path	string	mysql general log path
-mycnf_innodb_data_home_dir	string	Innodb data home directory
-mycnf_innodb_log_group_home_dir	string	Innodb log group home directory
-mycnf_master_info_file	string	mysql master.info file
-mycnf_mysql_port	int	port mysql is listening on
-mycnf_pid_file	string	mysql pid file
-mycnf_relay_log_index_path	string	mysql relay log index path
-mycnf_relay_log_info_path	string	mysql relay log info path
-mycnf_relay_log_path	string	mysql relay log path
-mycnf_server_id	int	mysql server id of the server (if specified, mycnf-file will be ignored)
-mycnf_slow_log_path	string	mysql slow query log path
-mycnf_socket_file	string	mysql socket file
-mycnf_tmp_dir	string	mysql tmp directory
-mysql_auth_server_static_file	string	JSON File to read the users/passwords from.
-mysql_auth_server_static_string	string	JSON representation of the users/passwords config.

Name	Type	Definition
-mysql_auth_static_reload_interval	duration	Ticker to reload credentials
-mysql_clientcert_auth_method	string	client-side authentication method to use. Supported values: mysql_clear_password, dialog. (default “mysql_clear_password”)
-mysql_server_flush_delay	duration	Delay after which buffered response will flushed to client. (default 100ms)
-mysqlctl_client_protocol	string	the protocol to use to talk to the mysqlctl server (default “grpc”)
-mysqlctl_mycnf_template	string	template file to use for generating the my.cnf file during server init
-mysqlctl_socket	string	socket file to use for remote mysqlctl actions (empty for local actions)
-onterm_timeout	duration	wait no more than this for OnTermSync handlers before stopping (default 10s)
-opentsdb_uri	string	URI of opentsdb /api/put method
-orc_api_password	string	(Optional) Basic auth password to authenticate with Orchestrator’s HTTP API.
-orc_api_url	string	Address of Orchestrator’s HTTP API (e.g. http://host:port/api/). Leave empty to disable Orchestrator integration.
-orc_api_user	string	(Optional) Basic auth username to authenticate with Orchestrator’s HTTP API. Leave empty to disable basic auth.
-orc_discover_interval	duration	How often to ping Orchestrator’s HTTP API endpoint to tell it we exist. 0 means never.
-orc_timeout	duration	Timeout for calls to Orchestrator’s HTTP API (default 30s)
-pid_file	string	If set, the process will write its pid to the named file, and delete it on graceful shutdown.
-pool_hostname_resolve_interval	duration	if set force an update to all hostnames and reconnect if changed, defaults to 0 (disabled)
-port	int	port for the server
-purge_logs_interval	duration	how often try to remove old logs (default 1h0m0s)
-query-log-stream-handler	string	URL handler for streaming queries log (default “/debug/querylog”)
-querylog-filter-tag	string	string that must be present in the query as a comment for the query to be logged, works for both vtgate and vttablet
-querylog-format	string	format for query logs (“text” or “json”) (default “text”)
-queryserver-config-acl-exempt-acl	string	an acl that exempt from table acl checking (this acl is free to access any vitess tables).
-queryserver-config-enable-table-acl-dry-run		If this flag is enabled, tabletserver will emit monitoring metrics and let the request pass regardless of table acl check results
-queryserver-config-idle-timeout	int	query server idle timeout (in seconds), vttablet manages various mysql connection pools. This config means if a connection has not been used in given idle timeout, this connection will be removed from pool. This effectively manages number of connection objects and optimize the pool performance. (default 1800)
-queryserver-config-max-dml-rows	int	query server max dml rows per statement, maximum number of rows allowed to return at a time for an update or delete with either 1) an equality where clauses on primary keys, or 2) a subselect statement. For update and delete statements in above two categories, vttablet will split the original query into multiple small queries based on this configuration value.
-queryserver-config-max-result-size	int	query server max result size, maximum number of rows allowed to return from vttablet for non-streaming queries. (default 10000)
-queryserver-config-message-postpone-cap	int	query server message postpone cap is the maximum number of messages that can be postponed at any given time. Set this number to substantially lower than transaction cap, so that the transaction pool isn’t exhausted by the message subsystem. (default 4)
-queryserver-config-passthrough-dmls		query server pass through all dml statements without rewriting
-queryserver-config-pool-prefill-parallelism	int	query server read pool prefill parallelism, a non-zero value will prefill the pool using the specified parallism.

Name	Type	Definition
-queryserver-config-pool-size	int	query server read pool size, connection pool is used by regular queries (non streaming, not in a transaction) (default 16)
-queryserver-config-query-cache-size	int	query server query cache size, maximum number of queries to be cached. vttablet analyzes every incoming query and generate a query plan, these plans are being cached in a lru cache. This config controls the capacity of the lru cache. (default 5000)
-queryserver-config-query-pool-timeout	int	query server query pool timeout (in seconds), it is how long vttablet waits for a connection from the query pool. If set to 0 (default) then the overall query timeout is used instead.
-queryserver-config-query-pool-waiter-cap	int	query server query pool waiter limit, this is the maximum number of queries that can be queued waiting to get a connection (default 5000)
-queryserver-config-query-timeout	int	query server query timeout (in seconds), this is the query timeout in vttablet side. If a query takes more than this timeout, it will be killed. (default 30)
-queryserver-config-schema-reload-time	int	query server schema reload time, how often vttablet reloads schemas from underlying MySQL instance in seconds. vttablet keeps table schemas in its own memory and periodically refreshes it from MySQL. This config controls the reload time. (default 1800)
-queryserver-config-schema-change-signal-interval	int	interval used to periodically send schema change updates to vtgate (default 5 seconds)
-queryserver-config-schema-change-signal	boolean	enable schema tracking
-queryserver-config-stream-buffer-size	int	query server stream buffer size, the maximum number of bytes sent from vttablet for each stream call. It's recommended to keep this value in sync with vtgate's stream_buffer_size. (default 32768)
-queryserver-config-stream-pool-prefill-parallelism	int	query server stream pool prefill parallelism, a non-zero value will prefill the pool using the specified parallelism
-queryserver-config-stream-pool-size	int	query server stream connection pool size, stream pool is used by stream queries: queries that return results to client in a streaming fashion (default 200)
-queryserver-config-strict-table-acl		only allow queries that pass table acl checks
-queryserver-config-terse-errors		prevent bind vars from escaping in returned errors
-queryserver-config-transaction-cap	int	query server transaction cap is the maximum number of transactions allowed to happen at any given point of a time for a single vttablet. E.g. by setting transaction cap to 100, there are at most 100 transactions will be processed by a vttablet and the 101th transaction will be blocked (and fail if it cannot get connection within specified timeout) (default 20)
-queryserver-config-transaction-prefill-parallelism	int	query server transaction prefill parallelism, a non-zero value will prefill the pool using the specified parallism.
-queryserver-config-transaction-timeout	int	query server transaction timeout (in seconds), a transaction will be killed if it takes longer than this value (default 30)
-queryserver-config-txpool-timeout	int	query server transaction pool timeout, it is how long vttablet waits if tx pool is full (default 1)
-queryserver-config-txpool-waiter-cap	int	query server transaction pool waiter limit, this is the maximum number of transactions that can be queued waiting to get a connection (default 5000)
-queryserver-config-warn-result-size	int	query server result size warning threshold, warn if number of rows returned from vttablet for non-streaming queries exceeds this
-redact-debug-ui-queries		redact full queries and bind variables from debug UI
-remote_operation_timeout	duration	time to wait for a remote operation (default 30s)
-restore_concurrency	int	(init restore parameter) how many concurrent files to restore at once (default 4)
-restore_from_backup		(init restore parameter) will check BackupStorage for a recent backup at startup and start there
-s3_backup_aws_endpoint	string	endpoint of the S3 backend (region must be provided)
-s3_backup_aws_region	string	AWS region to use (default "us-east-1")

Name	Type	Definition
-s3_backup_aws_retries	int	AWS request retries (default -1)
-s3_backup_force_path_style		force the s3 path style
-s3_backup_log_level	string	determine the S3 loglevel to use from LogOff, LogDebug, LogDebugWithSigning, LogDebugWithHTTPBody, LogDebugWithRequestRetries, LogDebugWithRequestErrors (default "LogOff")
-s3_backup_server_side_encryption	string	server-side encryption algorithm (e.g., AES256, aws:kms)
-s3_backup_storage_bucket	string	S3 bucket to use for backups
-s3_backup_storage_root	string	root prefix for all backup-related object names
-s3_backup_tls_skip_verify_cert		skip the 'certificate is valid' check for SSL connections
-security_policy	string	the name of a registered security policy to use for controlling access to URLs - empty means allow all for anyone (built-in policies: deny-all, read-only)
-service_map	value	comma separated list of services to enable (or disable if prefixed with '-') Example: grpc-vtworker
-serving_state_grace_period	duration	how long to pause after broadcasting health to vtgate, before enforcing a new serving state
-shard_sync_retry_delay	duration	delay between retries of updates to keep the tablet and its shard record in sync (default 30s)
-sql-max-length-errors	int	truncate queries in error logs to the given length (default unlimited)
-sql-max-length-ui	int	truncate queries in debug UIs to the given length (default 512) (default 512)
-srv_topo_cache_refresh	duration	how frequently to refresh the topology for cached entries (default 1s)
-srv_topo_cache_ttl	duration	how long to use cached entries for topology (default 1s)
-stats_backend	string	The name of the registered push-based monitoring/stats backend to use
-stats_combine_dimensions	string	List of dimensions to be combined into a single "all" value in exported stats vars
-stats_drop_variables	string	Variables to be dropped from the list of exported variables.
-stats_emit_period	duration	Interval between emitting stats to all registered backends (default 1m0s)
-stderrthreshold	value	logs at or above this threshold go to stderr (default 1)
-table-acl-config	string	path to table access checker config file; send SIGHUP to reload this file
-table-acl-config-reload-interval	duration	Ticker to reload ACLs
-tablet-path	string	tablet alias
-tablet_config	string	YAML file config for tablet
-tablet_dir	string	The directory within the vtdataroot to store vttablet/mysql files. Defaults to being generated by the tablet uid.
-tablet_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_grpc_cert	string	the cert to use to connect
-tablet_grpc_key	string	the key to use to connect
-tablet_grpc_server_name	string	the server name to use to validate server certificate
-tablet_hostname	string	if not empty, this hostname will be assumed instead of trying to resolve it
-tablet_manager_grpc_ca	string	the server ca to use to validate servers when connecting
-tablet_manager_grpc_cert	string	the cert to use to connect
-tablet_manager_grpc_concurrency	int	concurrency to use to talk to a vttablet server for performance-sensitive RPCs (like ExecuteFetchAs{Dba,AllPrivs,App}) (default 8)
-tablet_manager_grpc_key	string	the key to use to connect
-tablet_manager_grpc_server_name	string	the server name to use to validate server certificate
-tablet_manager_protocol	string	the protocol to use to talk to vttablet (default "grpc")
-tablet_protocol	string	how to talk to the vttablets (default "grpc")
-tablet_url_template	string	format string describing debug tablet url formatting. See the Go code for getTabletDebugURL() how to customize this. (default "http://{{.GetTabletHostPort}}")
-topo_consul_watch_poll_duration	duration	time of the long poll for watch queries. (default 30s)

Name	Type	Definition
-topo_etcd_lease_ttl	int	Lease TTL for locks and master election. The client will use KeepAlive to keep the lease going. (default 30)
-topo_etcd_tls_ca	string	path to the ca to use to validate the server cert when connecting to the etcd topo server
-topo_etcd_tls_cert	string	path to the client cert to use to connect to the etcd topo server, requires topo_etcd_tls_key, enables TLS
-topo_etcd_tls_key	string	path to the client key to use to connect to the etcd topo server, enables TLS
-topo_global_root	string	the path of the global topology data in the global topology server
-topo_global_server_address	string	the address of the global topology server
-topo_implementation	string	the topology implementation to use
-topo_k8s_context	string	The kubeconfig context to use, overrides the 'current-context' from the config
-topo_k8s_kubeconfig	string	Path to a valid kubeconfig file.
-topo_k8s_namespace	string	The kubernetes namespace to use for all objects. Default comes from the context or in-cluster config
-topo_zk_auth_file	string	auth to use when connecting to the zk topo server, file contents should be :, e.g., digest:user:pass
-topo_zk_base_timeout	duration	zk base timeout (see zk.Connect) (default 30s)
-topo_zk_max_concurrency	int	maximum number of pending requests to send to a Zookeeper server. (default 64)
-topo_zk_tls_ca	string	the server ca to use to validate servers when connecting to the zk topo server
-topo_zk_tls_cert	string	the cert to use to connect to the zk topo server, requires topo_zk_tls_key, enables TLS
-topo_zk_tls_key	string	the key to use to connect to the zk topo server, enables TLS
-topocustomrule_cell	string	topo cell for customrules file. (default "global")
-topocustomrule_path	string	path for customrules file. Disabled if empty.
-tracer	string	tracing service to use (default "noop")
-tracing-sampling-rate	float	sampling rate for the probabilistic jaeger sampler (default 0.1)
-transaction-log-stream-handler	string	URL handler for streaming transactions log (default "/debug/txlog")
-transaction_limit_by_component		Include CallerID.component when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_principal		Include CallerID.principal when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_by_subcomponent		Include CallerID.subcomponent when considering who the user is for the purpose of transaction limit.
-transaction_limit_by_username		Include VTGateCallerID.username when considering who the user is for the purpose of transaction limit. (default true)
-transaction_limit_per_user	float	Maximum number of transactions a single user is allowed to use at any time, represented as fraction of -transaction_cap. (default 0.4)
-shutdown_grace_period	float	how long to wait (in seconds) for queries and transactions to complete during graceful shutdown.
-twopc_abandon_age	float	time in seconds. Any unresolved transaction older than this time will be sent to the coordinator to be resolved.
-twopc_coordinator_address	string	address of the (VTGate) process(es) that will be used to notify of abandoned transactions.
-twopc_enable		if the flag is on, 2pc is enabled. Other 2pc flags must be supplied.
-tx-throttler-config	string	The configuration of the transaction throttler as a text formatted throttlerdata.Configuration protocol buffer message (default "target_replication_lag_sec: 2 max_replication_lag_sec: 10 initial_rate: 100 max_increase: 1 emergency_decrease: 0.5 min_duration_between_increases_sec: 40 max_duration_between_increases_sec: 62 min_duration_between_decreases_sec: 20 spread_backlog_across_sec: 20 age_bad_rate_after_sec: 180 bad_rate_increase: 0.1 max_rate_approach_threshold: 0.9")

Name	Type	Definition
-tx-throttler-healthcheck-cells	value	A comma-separated list of cells. Only tabletservers running in these cells will be monitored for replication lag by the transaction throttler.
-unhealthy_threshold	duration	replication lag after which a replica is considered unhealthy (default 2h0m0s)
-use_super_read_only		Set super_read_only flag when performing planned failover.
-v	value	log level for V logs
-version		print binary version
-vmodule	value	comma-separated list of pattern=N settings for file-filtered logging
-vreplication_healthcheck_retry_delay	duration	healthcheck retry delay (default 5s)
-vreplication_healthcheck_timeout	duration	healthcheck retry delay (default 1m0s)
-vreplication_healthcheck_topology_refresh	duration	refresh interval for re-reading the topology (default 30s)
-vreplication_retry_delay	duration	delay before retrying a failed binlog connection (default 5s)
-vreplication_tablet_type	string	comma separated list of tablet types used as a source (default “REPLICA”)
-vstream_packet_size	int	Suggested packet size for VReplication streamer. This is used only as a recommendation. The actual packet size may be more or less than this amount. (default 30000)
-vtctld_addr	string	address of a vtctld instance
-vtgate_protocol	string	how to talk to vtgate (default “grpc”)
-wait_for_backup_interval	duration	(init restore parameter) if this is greater than 0, instead of starting up empty when no backups are found, keep checking at this interval for a backup to appear
-watch_replication_stream		When enabled, vttablet will stream the MySQL replication stream from the local server, and use it to support the include_event_token ExecuteOptions.
-xbstream_restore_flags	string	flags to pass to xbstream command during restore. These should be space separated and will be added to the end of the command. These need to match the ones used for backup e.g. –compress / –decompress, –encrypt / –decrypt
-xtrabackup_backup_flags	string	flags to pass to backup command. These should be space separated and will be added to the end of the command
-xtrabackup_prepare_flags	string	flags to pass to prepare command. These should be space separated and will be added to the end of the command
-xtrabackup_root_path	string	directory location of the xtrabackup executable, e.g., /usr/bin
-xtrabackup_stream_mode	string	which mode to use if streaming, valid values are tar and xbstream (default “tar”)
-xtrabackup_stripe_block_size	uint	Size in bytes of each block that gets sent to a given stripe before rotating to the next stripe (default 102400)
-xtrabackup_stripes	uint	If greater than 0, use data striping across this many destination files to parallelize data transfer and decompression
-xtrabackup_user	string	User that xtrabackup will use to connect to the database server. This user must have all necessary privileges. For details, please refer to xtrabackup documentation.

Key Options

- -restore_from_backup: The default value for this flag is false. If set to true, and the my.cnf file was successfully loaded, then vttablet can perform automatic restores as follows:
  - If started against a mysql instance that has no data files, it will search the list of backups for the latest one, and initiate a restore. After this, it will point the mysql to the current master and wait for replication to catch up. Once replication is caught up to the specified tolerance limit, it will advertise itself as serving. This will cause the vtgates to add it to the list of healthy tablets to serve queries from.
  - If this flag is true, but my.cnf was not loaded, then vttablet will fatally exit with an error message.

– You can additionally control the level of concurrency for a restore with the `-restore_concurrency` flag. This is typically useful in cloud environments to prevent the restore process from becoming a 'noisy' neighbor by consuming all available disk IOPS.

VReplication

description: "VReplication vtctl commands (now defaulting to the experimental v2 commands)" skip_sections: true

— Cancel

description: Tear down a workflow where no traffic has been switched

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-keep_data] Cancel <targetKs.workflow>
```

Description `Cancel` can be used if a workflow was created in error or was misconfigured and you prefer to create a new workflow instead of fixing this one. Cancel can only be called if no traffic has been switched. It removes vreplication-related artifacts like rows from vreplication and copy_state tables in the side-car `_vt` database and routing rules and blacklisted tables from the topo and, by default, the target tables/shards from the target keyspace.

Parameters

-keep_data optional
default false

Usually, the target data (tables or shards) are deleted by Cancel. If this flag is used with MoveTables, target tables will not be deleted and, with Reshard, target shards will not be dropped.

— Complete

description: Tear down a workflow after switching all traffic

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-keep_data] [-rename_tables] [-dry_run]
  Complete <targetKs.workflow>
```

Description Alert: This is a destructive command

`Complete` is used after all traffic has been switched. It removes vreplication-related artifacts like rows from vreplication and copy_state tables in the side-car `_vt` database and routing rules and and blacklisted tables (for MoveTables) from the topo. By default, the source tables (or source shards) are also dropped.

Parameters

-keep_data optional
default false

Usually, the source data (tables or shards) are deleted by Complete. If this flag is used, for MoveTables, source tables will not be deleted, for Reshard, source shards will not be dropped.

-rename_tables optional
default false

The rename_tables flag is applicable only for MoveTables. Tables are renamed instead of being deleted. Currently the new name is _<table_name>_old.

We use the same renaming logic used by `pt-online-schema-change`. Such tables are automatically skipped by vreplication if they exist on the source.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by Complete.

"— Create"

description: Initiate a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables [-source=<sourceKs>] [-tables=<tableSpecs>] [-cells=<cells>]
  [-tablet_types=<source_tablet_types>] [-all] [-exclude=<tables>]
   [-auto_start] [-stop_after_copy]
  Create <targetKs.workflow>

Reshard [-source_shards=<source_shards>] [-target_shards=<target_shards>]
  [-cells=<cells>] [-tablet_types=<source_tablet_types>]  [-skip_schema_copy]
  Create <keyspace.workflow>
```

Description `MoveTables/Reshard Create` sets up and creates a new workflow. The workflow name should not conflict with that of an existing workflow.

Parameters

-source mandatory

MoveTables only

Name of existing keyspace (the source keyspace) that contains the tables to be moved.

table_specs mandatory Either `table_specs` or `-all` needs to be specified

MoveTables only

Either

- a comma-separated list of tables
  - if target keyspace is unsharded OR
  - if target keyspace is sharded AND the tables being moved are already defined in the target's vschema

  Example: `MoveTables -source=commerce -tables=customer,corder Create customer.commerce2customer`

Or

- the JSON table section of the vschema for associated tables
  - if target keyspace is sharded AND
  - tables being moved are not yet present in the target's vschema

  Example: `MoveTables -source=commerce -tables='{"t1":{"column_vindexes": [{"column": "id", "name": "hash"}]}}}' Create customer.commerce2customer`

-cells optional
default local cell

A comma-separated list of cell names or cell aliases. This list is used by VReplication to determine which cells should be used to pick a tablet for selecting data from the source keyspace.

Uses

- Improve performance by picking a tablet in cells in network proximity with the target
- Reduce bandwidth costs by skipping cells that are in different availability zones
- Select cells where replica lags are lower

-tablet_types optional
default replica

A comma-separated list of tablet types that are used while picking a tablet for sourcing data. One or more from MASTER, REPLICA, RDONLY.

Uses

- To reduce the load on master tablets by using REPLICAs or RDONLYs
- Reducing lags by pointing to MASTER

-auto_start optional default true

Normally the workflow starts immediately after it is created. If this flag is set to false then the workflow is in a Stopped state until you explicitly start it.

Uses

- Allows updating the rows in `_vt.vreplication` after MoveTables has setup the streams. For example, you can add some filters to specific tables or change the projection clause to modify the values on the target. This provides an easier way to create simpler Materialize workflows by first using MoveTables with auto_start false, updating the BinlogSource as required by your Materialize and then starting the workflow.
- Changing the `copy_state` and/or `pos` values to restart a broken MoveTables workflow from a specific point of time.

-stop_after_copy **optional default** false

If set, the workflow will stop once the Copy phase has been completed i.e. once all tables have been copied and VReplication decides that the lag is small enough to start replicating, the workflow state will be set to Stopped.

Uses

- If you just want a consistent snapshot of all the tables you can set this flag. The workflow will stop once the copy is done and you can then mark the workflow as `Completed`

-all **optional** cannot specify `table_specs` if `-all` is specified

MoveTables only

Move all tables from the source keyspace.

-exclude **optional** only applies if `-all` is specified

MoveTables only

If moving all tables, specifies tables to be skipped.

source_shards **mandatory**

Reshard only

Comma-separated shard names to reshard from.

Example: `Reshard -source_shards=0 -target_shards=-80,80- Create customer.reshard1to2`

target_shards **mandatory**

Reshard only

Comma-separated shard names to reshard to.

-skip_schema_copy **optional**
default false

Reshard only

If true the source schema is copied to the target shards. If false, you need to create the tables before calling `Reshard`.

Flags

description: vttablet flags related to VReplication functionality

There are several flags that can be specified when `vttablet` is launched that are related to the VReplication functionality. Some of the flags are relevant when tablets are acting as targets and others when tablets are acting as sources in a VReplication workflow.

relay_log_max_size **Type** integer
Unit bytes
Default 250000
Applicable on target

The target tablet receives events from the source and applies the corresponding DML to the underlying MySQL database. Depending on the load on the target the query execution times can change. Also during the copy phase, we are doing bulk

inserts. For both of these reasons VReplication introduces a buffer between receiving the events and applying them: the *relay log*.

The relay log buffers events on the target as they are received from the source. This is done in a separate thread concurrently with the thread that applies the events.

relay_log_max_size defines the maximum buffer size (in bytes). As events arrive they are stored in the relay log. The apply thread consumes these events as fast as it can. When the relay log fills up we no longer pull events from the source until some events are consumed. If single rows are larger than the specified buffer size, a single row is buffered at a time.

vstream_packet_size **Type** integer
Unit bytes
Default 250000
Applicable on source

On the source, events are buffered where applicable, to minimize network overhead. For example, multiple row events in a transaction or the set of begin/dml/commit event sets are buffered and sent together. Commits, DDLs, and synthetic events generated by VReplication like heartbeats, resharding journals cause the events buffered on the source to be sent immediately.

vstream_packet_size specifies the suggested packet size for VReplication streamer. This is used only as a recommendation. The actual packet size may be more or less than this amount depending on the number and type of events yet to be sent on the source.

vreplication_heartbeat_update_interval **Type** integer
Unit seconds
Default 1
Maximum 60 (one minute)
Applicable on target

For an idle source shard, the source vstreamer sends a heartbeat. Currently, that is once per second. On receiving the heartbeat the target VReplication module updates the time_updated column of the relevant row of `\_vt.vreplication`. For some setups this is a problem, for example:

- if there are too many streams the extra write QPS or CPU load due to these updates are unacceptable
- if there are too many streams and/or a large source field (lot of participating tables) which generates unacceptable increase in the binlog size
- even for a single stream, if the server is of a lower configuration, then the resulting increase in the QPS or binlog increase may become significant as a percentage of resources

vreplicationHeartbeatUpdateInterval determines how often the time_updated column is updated if there is no activity on the source and the source vstream is only sending heartbeats. Use a low value if you expect a high QPS or you are monitoring this column to alert about potential outages. Keep this high if:

- you have too many streams and the extra write QPS or CPU load due to these updates is unacceptable OR
- you have too many streams and/or a large binlogsource field (i.e., there are a lot of participating tables) which generates unacceptable increase in your binlog size

Some internal processes (like online ddl) depend on the heartbeat updates for operating properly. Hence there is an upper limit on this interval, which is 60 seconds.

watch_replication_stream **Type** bool
Default false
Applicable on source

By default vttablets reload their schema every `queryserver-config-schema-reload-time` seconds (default 30 minutes). This can cause a problem while streaming events if DDLs are applied on the source and streaming is started *after* the DDL was applied but *before* vttablet refreshed its schema. This is alleviated by the *watcher*.

When enabled, vttablet will start the *watcher* which streams the MySQL replication stream from the local database, and uses it to proactively update its schema when it encounters a DDL.

track_schema_versions **Type** bool
Default false
Applicable on source

All vstreams on a tablet share a common engine. vstreams that are lagging might see a newer (and hence incorrect) version of the schema in case DDLs were applied in between. Also, reloading schemas is an expensive operation. If there are multiple vstreams, each of them will separately receive a DDL event resulting in multiple reloads for the same DDL. The tracker addresses these issues.

When enabled, vttablet will start the *tracker* which runs a separate vstream that monitors DDLs and stores the version of the schema at the position that a DDL is applied in the schema version table. So if we are streaming events from the past we can get the corresponding schema and interpret the fields from the event correctly.

vreplication_retry_delay **Type** integer
Unit seconds
Default 5
Applicable on target

The target might encounter connection failures during a workflow. VReplication automatically retries stalled streams after *vreplication_retry_delay* seconds

vreplication_tablet_type **Type** string
Default MASTER,REPLICA
Applicable on target

This parameter specifies the default tablet_types that will be used by the tablet picker to find sources for a VReplication stream. It can be overridden, per workflow, by passing a different list to the workflow commands like MoveTables and Reshard.

vreplication_experimental_flags **Type** bitmask
Default 0
Applicable on target

Features that are not field-tested, that are not backward-compatible, or need to be proven in production environments are put behind *vreplication_experimental_flags*. These features are temporary and will either be made permanent, removed, or put behind a separate vttablet option. Currently, the only experimental features are expected to be performance improvements.

This will be a bit-mask for each such feature. The ones currently defined:

bitmask: *0x1* => If set then we optimize the catchup phase by not sending inserts for rows that are outside the range of primary keys already copied. More details at https://github.com/vitessio/vitess/pull/7708

Life of a stream

Introduction The diagram below outlines how a VReplication workflow is performed. VReplication can be asked to start from a specific GTID or from the start. When starting from a GTID the *replication* mode is used where it streams events from the binlog.

Full table copy When starting from the beginning the simple streaming done by *replication* can create an avalanche of events (think 10s of millions of rows). To speed things up a *copy/catchup* mode is initiated first: data in the tables are copied over in a consistent manner using bulk inserts. Once we have copied enough data so that we are close enough to the current position (when replication lag is low) it switches over (and stays in) the *replication* mode.

While we may have multiple database sources in a workflow each vstream has just one source and one target. The source is always a vttablet (and hence one mysql instance). The target could be another vttablet (resharding) or a streaming grpc response (vstream api clients).

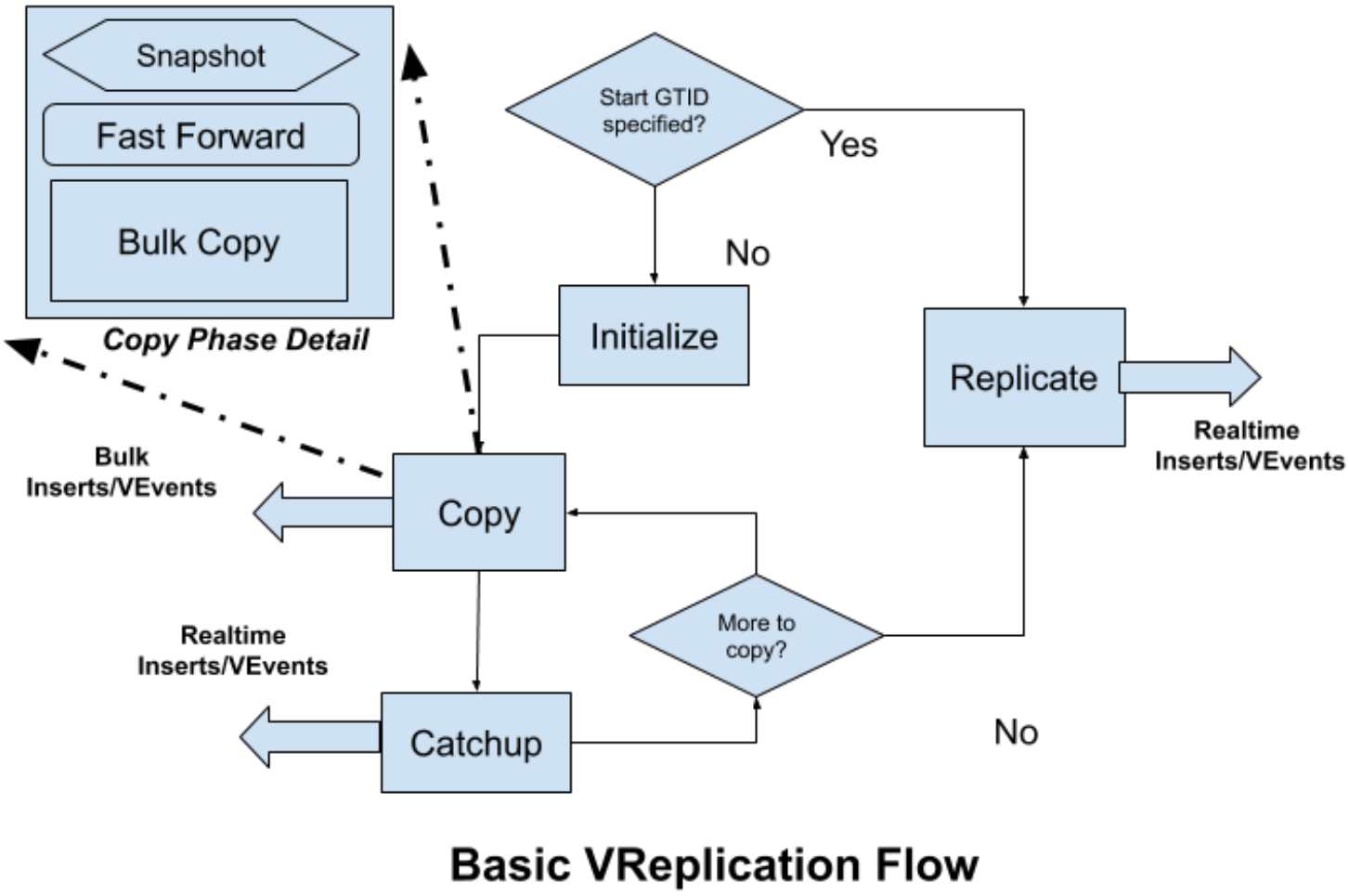


Figure 5: VReplication Flow

Transformation and Filtering Note that for all steps the data selected from the source will only be from the list of tables specified (specified via Match). Furthermore if a Filter is specified for a table it will be applied before being sent to the target. Columns may also be transformed in the Filter's select clause.

Source and Sink Each stream has two parts. The target initiates streaming by making grpc calls to the source tablet. The source sources the data connecting to mysql as a replica or using sql queries and streams it to the target. The target takes appropriate action: in case of resharding it will convert the events into CRUDs and apply it to the target database. In case of vstream clients the events are forwarded by vtgate to the client.

Note that the target always pulls the data. This ensures that there is no problems of buffer overruns that can occur if the source is pushing the data since (especially in sharding) it is possible that the application of events can be substantially cpu intensive especially in the case of bulk inserts.

Modes, in detail

Replicate This is the easiest step to understand. The source stream just mimics a mysql replica and processes events as they are received. Events (after filtering and transformation) are sent to the target. Replication runs continuously with short sleeps when there are no more events to source.

Initialize Initialize is called at the start of the copy phase. For each table to be copied an entry is created in _vt.copy_state with a zero primary key. As each table copy is completed the related entry is deleted and when there are no more entries for this workflow the copy phase is considered complete and the workflow moves into the Replication mode.

Copy Copy works on one table at a time. The source selects a set of rows from the table with higher primary keys that the one copied so far using a consistent snapshot. This results in a stream of rows to be sent to the target which generates a bulk insert of these rows.

However there are a couple of factors which complicate our story::

- Each copy selects all rows until the current position of the binlog.
- Since transactions continue to be applied (presuming the database is online) the gtid positions are continuously moving forward

Consider this example.

We have two tables t1 and t2 and this is how the copy state proceeds: Each has 20 rows and we copy 10 rows at a time. (Queries are not exact but simplified for readability).

If we follow this we get:

```
T1: select * from t1 where pk > 0 limit 10. GTID: 100, Last PK 10

   send rows to target

T2: select * from t1 where pk > 10 limit 10  GTID: 110, Last PK 20

   send rows to target
```

Gotcha: however we see that 10 new txs have occurred since T1. Some of these can potentially modify the rows returned from the query at T1. Hence if we just return the rows from T2 (which have only rows from pk 11 to 20)
we will have an inconsistent state on the target: the updates to rows with PK between 1 and 10 will not be present.

This means that we need to first stream the events between 100 to 110 for PK between 1 and 10 first and then do the second select:

```
T1: select * from t1 where pk > 0 limit 10. GTID: 100, Last PK 10

   send rows to target

T2: replicate from 100 to current position (110 from previous example),

   only pass events for pks 1 to 10

T3: select * from t1 where pk > 10 limit 10  GTID: 112, Last PK 20

   send rows to target
```

Another gotcha!: Note that at T3 when we selected the pks from 11 to 20 the gtid position has moved further! This happened because of transactions that were applied between T2 and T3. So if we just applied the rows from T3 we would still have an inconsistent state, if transactions 111 and 112 affected the rows from pks 1 to 10.

This leads us to the following flow:

```
T1: select * from t1 where pk > 0 limit 10. GTID: 100, Last PK 10

   send rows to target

T2: replicate from 100 to current position (110 from previous example),

   only pass events for pks 1 to 10

T3: select * from t1 where pk > 10 limit 10  GTID: 112, Last PK 20

T4: replicate from 111 to 112

   only pass events for pks 1 to 10

T5: Send rows for pks 11 to 20 to target
```

This flow actually works!

T1 can take a long time (due to the bulk inserts). T3 (which is just a snapshot) is quick. So the position can diverge much more at T2 than at T4. Hence we call the step in T2 as Catchup and Step T4 is called Fast Forward.

Catchup As detailed above the catchup phase runs between two copy phases. During the copy phase the gtid position can move significantly ahead. So we run a replicate till we come close to the current position i.e.the replication lag is small. At this point we call Copy again.

Fast forward During the copy phase we first take a snapshot. Then we fast forward: we run another replicate from the gtid position where we stopped the Catchup to the position of the snapshot.

Finally once we have finished copying all the tables we proceed to replicate until our job is done: for example if we have resharded and switched over the reads and writes to the new shards or when the vstream client closes its connection.

Materialize

description:

```
Materialize [-cells=<cells>] [-tablet_types=<source_tablet_types>] <json_spec>
```

Description Materialize is a low level vreplication API that allows for generalized materialization of tables. The target tables can be copies, aggregations or views. The target tables are kept in sync in near-realtime.

You can specify multiple tables to materialize using the json_spec parameter.

Parameters

-cells optional
default local cell

A comma-separated list of cell names or cell aliases. This list is used by VReplication to determine which cells should be used to pick a tablet for selecting data from the source keyspace.

Uses

- Improve performance by using picking a tablet in cells in network proximity with the target
- To reduce bandwidth costs by skipping cells that are in different availability zones
- Select cells where replica lags are lower

-tablet_types optional
default replica

A comma-separated list of tablet types that are used while picking a tablet for sourcing data. One or more from MASTER, REPLICA, RDONLY.

Uses

- To reduce the load on master tablets by using REPLICAs or RDONLYs
- Reducing lags by pointing to MASTER

JSON spec details

- *workflow* name to refer to this materialization
- *source_keyspace* keyspace containing the source table
- *target_keyspace* keyspace to materialize to
- *table_settings* list of views to be materialized and the associated query
  - *target_table* name of table to which to materialize the data to
  - *source_expression* the materialization query
- Optional parameters:
  - *stop_after_copy* if vreplication should be stopped after the copy phase is complete
  - *cell* name of a cell, or a comma separated list of cells, that should be used for choosing source tablet(s) for the materialization. If this parameter is not specified, only cell(s) local to the target tablet(s) is considered
  - *tablet_types* a Vitess tablet_type, or comma separated list of tablet types, that should be used for choosing source tablet(s) for the materialization. If not specified, this defaults to the tablet type(s) specified by the `-vreplication_tablet_type` VTTablet command line flag

```
Materialize '{"workflow": "product_sales", "source_keyspace": "commerce",
   "target_keyspace": "customer",
    "table_settings": [{"target_table": "sales_by_sku",
    "source_expression": "select sku, count(*), sum(price) from corder group by order_id"}],
    "cell": "zone1", "tablet_types": "REPLICA"}'
```

A Materialize Workflow Once you decide on your materialization requirements, you need to initiate a VReplication workflow as follows:

1. Initiate the migration using Materialize
2. Monitor the workflow using Workflow or VExec
3. Start accessing your views once the workflow has started Replicating

Notes There are special commands to perform common materialization tasks and you should prefer them to using Materialize directly.

- If you just want to copy tables to a different keyspace use MoveTables.
- If you want to change sharding strategies use Reshard instead

Metrics

description: Metrics related to vreplication functionality

VReplication exports several metrics using the expvars interface. These are available at the `debug/vars` endpoint of vttablet's http status pages. More details here

Target Metrics

VReplicationCopyLoopCount, VReplicationCopyLoopCountTotal During the copy phase we run one loop of bulk copy for approximately an hour at a time before running catchup. *VReplicationCopyLoopCount* counts the number of times this loop has run for each stream and *VReplicationCopyLoopCountTotal* the total across all streams.

VReplicationCopyRowCount, VReplicationCopyRowCountTotal *VReplicationCopyRowCount* counts the number of rows copied during the copy phase per stream and *VReplicationCopyRowCountTotal* the total across all streams.

VReplicationErrors *VReplicationErrors* counts the number of times errors occurred during vreplication. Errors are keyed with the type of error.

VReplicationHeartbeat *VReplicationHeartbeat* records, for each stream, the timestamp sent by the last heartbeat event for that stream.

VReplicationMessages *VReplicationMessages* contains a stack of the last N (currently 3) messages of a vreplication stream.

VReplicationMessages *VReplicationMessages* contains a stack of the last N (currently 3) messages of a vreplication stream.

VReplicationPhaseTimings, VReplicationPhaseTimingsCounts, VReplicationPhaseTimingsTotal This metric relates to the times each phase is run during the lifetime of a stream. *VReplicationPhaseTimings* counts the total time taken by the runs, VReplicationPhaseTimingsCounts the number of runs and *VReplicationPhaseTimingsTotals* the total runs across all streams.

VReplicationQPS *VReplicationQPS* is a list of QPS values for each loop of each phase of the workflow.

VReplicationQueryCount, VReplicationQueryCountTotal *VReplicationQueryCount* is the total number of queries in each phase of a workflow. *VReplicationQueryCountTotal* is the total queries across all phases and workflows.

VReplicationSecondsBehindMaster, VReplicationSecondsBehindMasterMax, VReplicationSecondsBehindMasterTotal These metrics show the replication lag of the target stream with respect to the source stream. *VReplicationSecondsBehindMaster* shows the current replication lag and *VReplicationSecondsBehindMasterMax* has the maximum lag in this stream. Note that these values are only valid during the replication phase of a workflow.

VReplicationSource Shows the keyspace and shard of the source from which this target stream is replicating

VReplicationSourceTablet Shows the tablet from which this stream is currently replicating

VReplicationStreamCount The number of streams running on this target

VReplicationStreamState This shows the state of each stream.

Source Metrics

VStreamPacketSize The value of the `vstream_packet_size` flag specified for this tablet

VStreamerErrors The number of errors per category across workflows

VStreamersEndedWithErrors The total number of errors that caused a stream to stall

VStreamerEventsStreamed The total number of events streamed by this vttablet across all workflows

VStreamerNumPackets The total number of packets sent by this vttablet across all workflows

VStreamersCreated The total number of vstreamer created during the lifetime of this tablet

Example

A snippet from tablet 200 from the local example after running the MoveTables step


```
"VReplicationCopyLoopCount": {"commerce.0.commerce2customer.1": 2},
"VReplicationCopyLoopCountTotal": 2,
"VReplicationCopyRowCount": {"commerce.0.commerce2customer.1": 10},
"VReplicationCopyRowCountTotal": 10,
"VReplicationErrors": {},
"VReplicationHeartbeat": {"commerce.0.commerce2customer.1": 1618681048},
"VReplicationMessages": {"1": "2021-04-17T19:36:13.003858838+02:00:Picked source tablet:
   cell:\"zone1\" uid:100 "},
"VReplicationPhaseTimings": {"commerce.0.commerce2customer.1.catchup": 948,
   "commerce.0.commerce2customer.1.fastforward": 555,
   "commerce.0.commerce2customer.1.copy": 615},
"VReplicationPhaseTimingsCounts": {"commerce.0.commerce2customer.1.fastforward": 2,
   "commerce.0.commerce2customer.1.copy": 2, "commerce.0.commerce2customer.1.All": 6,
   "commerce.0.commerce2customer.1.catchup": 2},
"VReplicationPhaseTimingsTotal": 2118,
"VReplicationQPS":
   {"All":[7.2,1,1.2,1,1.2,1,1.2,1.2,1,1.2,1,1.2,1,1.2,1],"Query":[6.6,1,1.2,1,1.2,1,1.2,1.2,1,1.2
"VReplicationQueryCount": {"commerce.0.commerce2customer.1.copy": 2},
"VReplicationQueryCountTotal": 2,
```

```
"VReplicationSecondsBehindMaster": {"commerce.0.commerce2customer.1": 0},
"VReplicationSecondsBehindMasterMax": 0,
"VReplicationSecondsBehindMasterTotal": 0,
"VReplicationSource": {"1": "commerce/0"},
"VReplicationSourceTablet": {"1": "cell:\"zone1\" uid:100 "},
"VReplicationStreamCount": 1,
"VReplicationStreamState": {"commerce2customer.1": "Running"},
"VStreamPacketSize": 250000,
"VStreamerErrors": {"Catchup": 0, "Copy": 0, "Send": 0, "TablePlan": 0},
"VStreamerEventsStreamed": 0,
"VStreamerNumPackets": 0,
"VStreamerPhaseTiming": {"TotalCount":0,"TotalTime":0,"Histograms":{}},
"VStreamersCreated": 0,
"VStreamersEndedWithErrors": 0,
```


Migrate

description: Move tables from an external cluster

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
Migrate <options> <action> <workflow identifier>
```

Description Migrate is used to start and manage workflows to move one or more tables from an external cluster into a new Vitess keyspace. The target keyspace can be unsharded or sharded.

Migrate is typically used for migrating data into Vitess from an external cluster. See Mount command for more information on external clusters.

Parameters

action Migrate is an "umbrella" command. The `action` sub-command defines the operation on the workflow. The only actions supported by Migrate are Create, Complete and Cancel.

options Each `action` has additional options/parameters that can be used to modify its behavior.

The options for the supported commands are the same as for `MoveTables`.

workflow identifier All workflows are identified by `targetKeyspace.workflow` where `targetKeyspace` is the name of the keyspace to which the tables are being moved. `workflow` is a name you assign to the MoveTables workflow to identify it.

A Migrate Workflow lifecycle

1. Mount the external cluster using Mount `Mount -type vitess -topo_type etcd2 -topo_server localhost:12379 -topo_root /vitess/global ext1`
2. Initiate the migration using Create `Migrate -all -source ext1.load Create commerce.wf`
3. Monitor the workflow using Show `Workflow Show commerce.wf`
4. Confirm that data has been copied over correctly using VDiff
5. Cleanup vreplication artifacts and source tables with Complete `Migrate Complete commerce.wf`

Differences between MoveTables and Migrate MoveTables has separate semantics than Migrate. MoveTables can migrate data from one keyspace to another, but both keyspaces need to be in the same Vitess Cluster. Migrate is intended as a one-way copy whereas MoveTables allows you to serve either data from the source or target keyspace and switch between each other until you finalize MoveTables.

- MoveTables sets up routing rules so that Vitess routes queries to the Source keyspace until a cutover
- While switching Write traffic, in MoveTables, is possible to set up a reverse replication workflow so that the Source can be in sync with the Target, allowing you to revert back to the Source.

However this requires that the Target can create vreplication streams (in the _vt database) on the Source database. This may not be always possible, for example, if the Source is a production system.

- In MoveTables the tables already exist, just in a different keyspace. So the VSchema already contains these tables. While migrating, these tables will be available only after the Migrate is completed
- Switching traffic is not meaningful in the case of Migrate since there is no query traffic to the original tables, as the Source is in a different cluster.

Mount

description: Link an external cluster to the current one

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
Mount [-type vitess] [-topo_type=etcd2|consul|zookeeper] [-topo_server=topo_url]
    [-topo_root=root_topo_node> [-unmount] [-list] [-show]  [<cluster_name>]
```

Description Mount is used to link external vitess clusters to the current cluster. (In the future we will also support mounting external MySQL servers.)

Mounting vitess clusters requires the topology information of the external cluster to be specified. Used in conjunction with the Migrate command.

Parameters

cluster_name The name that will be used in VReplication workflows to refer to the mounted cluster. Required when mounting, unmounting or getting details of a cluster.

unmount Unmount an already mounted cluster. Requires `cluster_name` to be specified.

show Show details of an already mounted cluster. Requires `cluster_name` to be specified.

list List all mounted clusters

Topo parameters topo_type=[etcd2|consul|zookeeper]

topo_server=

topo_root=

Mandatory (and only specified) while mounting a Vitess cluster. These should specify the topology parameters of the cluster being mounted.

MoveTables

description: Move tables between keyspaces without downtime

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables <options> <action> <workflow identifier>
```

Description MoveTables is used to start and manage workflows to move one or more tables from an external database or an existing Vitess keyspace into a new Vitess keyspace. The target keyspace can be unsharded or sharded.

MoveTables is typically used for migrating data into Vitess or to implement vertical sharding. You might use the former when you first start using Vitess and the latter if you want to distribute your load across servers without sharding tables.

Parameters

action MoveTables is an top-level command. The `action` sub-command defines the operation on the workflow.

options Each `action` has additional options/parameters that can be used to modify its behavior.

`actions` are common to both MoveTables and Reshard v2 workflows. Only the `create` action has different parameters, all other actions have common options and similar semantics. These actions are documented separately.

workflow identifier All workflows are identified by `targetKeyspace.workflow` where `targetKeyspace` is the name of the keyspace to which the tables are being moved. `workflow` is a name you assign to the MoveTables workflow to identify it.

The most basic MoveTables Workflow lifecycle

1. Initiate the migration using Create `MoveTables -source=<sourceKs> -tables=<tableSpecs> Create <targetKs.workflow>`
2. Monitor the workflow using Show or Progress `MoveTables Show <targetKs.workflow>` or `MoveTables Progress <targetKs.workflow>`
3. Confirm that data has been copied over correctly using VDiff
4. Cutover to the target keyspace with SwitchTraffic `MoveTables SwitchTraffic <targetKs.workflow>`
5. Cleanup vreplication artifacts and source tables with Complete `MoveTables Complete <targetKs.workflow>`

Common use cases for MoveTables

Adopting Vitess For those wanting to try out Vitess for the first time, MoveTables provides an easy way to route part of their workload to Vitess with the ability to migrate back at any time without any risk. You point a vttablet to your existing MySQL installation, spin up an unsharded Vitess cluster and use a MoveTables workflow to start serving some tables from Vitess. You can also go further and use a Reshard workflow to experiment with a sharded version of a part of your database.

See user guide for detailed steps

Vertical Sharding For existing Vitess users you can easily move one or more tables to another keyspace, either for balancing load or as preparation for sharding your tables.

See user guide which describes how MoveTables works in the local example provided in the Vitess repo.

More Reading

- MoveTables in practice

— Progress

description: Show copy progress and replication lag of a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard Progress <targetKs.workflow>
```

Description Workflows start in the copy state, (details in VReplication Internals), doing a bulk copy of the tables involved until they reach a low replication lag, after which we stream binlog events. Tables are copied sequentially.

`Progress` reports the progress of a workflow by showing the percentage of data copied across targets, if workflow is in copy state, and the replication lag between the target and the source once the copy phase is completed.

It is too expensive to get real-time row counts of tables, using _count(*)_, say. So we use the statistics available in the `information_schema` to approximate copy progress. This data can be significantly off (up to 50-60%) depending on the utilization of the underlying mysql server resources. You can manually run `analyze table` to update the statistics if so desired.

Reshard

description: Reshard a keyspace to achieve horizontal scaling

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
Reshard <options> <action> <workflow identifier>
```

Description Reshard is used to create and manage workflows to horizontally shard an existing keyspace. The source keyspace can be unsharded or sharded.

Parameters

action Reshard is an "umbrella" command. The `action` sub-command defines the operation on the workflow.

options Each `action` has additional options/parameters that can be used to modify its behavior.

`actions` are common to both MoveTables and Reshard v2 workflows. Only the `create` action has different parameters, all other actions have common options and similar semantics. These actions are documented separately.

workflow identifier All workflows are identified by `targetKeyspace.workflow` where `targetKeyspace` is the name of the keyspace to which the tables are being moved. `workflow` is a name you assign to the Reshard workflow to identify it.

The most basic Reshard Workflow lifecycle

1. Initiate the migration using Create `Reshard -source_shards=<source_shards> -target_shards=<target_shards> Create <keyspace.workflow>`
2. Monitor the workflow using Show or Progress `Reshard Show <keyspace.workflow>` or `Reshard Progress <keyspace.workflow>`
3. Confirm that data has been copied over correctly using VDiff
4. Cutover to the target keyspace with SwitchTraffic `Reshard SwitchTraffic <keyspace.workflow>`
5. Cleanup vreplication artifacts and source shards with Complete `Reshard Complete <keyspace.workflow>`

— ReverseTraffic

description: Reverse traffic from target to source

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-tablet_types <tablet_types_csv>] [-cells <cells>]
  [-timeout=timeoutDuration] [-dry_run]
  ReverseTraffic <targetKs.workflow>
```

Description `ReverseTraffic` switches traffic in the reverse direction for the tablet_types specified. The traffic should have been previously switched forward using SwitchTraffic for the cells/tablet_types specified.

Parameters

-cells optional
default all cells

A comma-separated list of cell names or cell aliases. Traffic will be reversed only in these cells for the specified tablet types.

-tablet_types optional
default all (replica,rdonly,master)

A comma-separated list of tablet types for which traffic is to be reversed. One or more from master, replica, rdonly.

-timeout optional
default 30s

For master tablets, ReverseTraffic first stops writes on the target master and waits for the replication on the source to catchup with the gtid on the target where the writes were stopped. If the wait time is longer than timeout the command will error out. For setups with high write qps you may need to increase this value.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by ReverseTraffic.

— Show

description: Show information about a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard Show <targetKs.workflow>
```

Description `Show` displays useful information about a workflow. (At this time the Workflow Show command gives more information. This will be improved over time.)

— SwitchTraffic

description: Switch traffic from source to target

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-tablet_types=<tablet_types>] [-cells=<cells>]
  [-timeout=timeoutDuration] [-reverse_replication] [-dry_run]
  SwitchTraffic <targetKs.workflow>
```

Description `SwitchTraffic` switches traffic forward for the tablet_types specified. This replaces the previous SwitchReads and SwitchWrites commands with a single one. It is now possible to switch all traffic with just one command. Also, you can now switch replica, rdonly and primary traffic in any order: earlier you needed to first SwitchReads (for replicas and rdonly tablets) first before SwitchWrites.

Parameters

-cells optional
default all cells

A comma-separated list of cell names or cell aliases. Traffic will be switched only in these cells for the specified tablet types.

-tablet_types optional
default all (replica,rdonly,master)

A comma-separated list of tablet types for which traffic is to be switched. One or more from master,replica,rdonly.

-timeout optional
default 30s

For master tablets, SwitchTraffic first stops writes on the source master and waits for the replication to the target to catchup with the point where the writes were stopped. If the wait time is longer than timeout the command will error out. For setups with high write qps you may need to increase this value.

-reverse_replication optional
default true

SwitchTraffic for master tablet types, by default, starts a reverse replication stream with the current target as the source, replicating back to the original source. This enables a quick and simple rollback using ReverseTraffic. This reverse workflow name is that of the original workflow concatenated with _reverse.

If set to false these reverse replication streams will not be created and you will not be able to rollback once you have switched write traffic over to the target.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchTraffic.

Throttling

Introduction VReplication moves potentially massive amounts of data from one place to another, whether within the same keyspace and shard or across keyspaces. It copies data of entire tables and follows up to apply ongoing changes on those tables by reading the binary logs (aka the changelog).

This places load on both the source side (where VReplication reads data from) as well as on target side (where VReplication writes data to).

On the source side, VReplication reads the full content of tables. This typically means loading pages from disk contending for disk IO, and "polluting" the MySQL buffer pool. The operation competes with normal production traffic for both IO and memory resources. If the source is a replica, the operation may lead to replication lag. If the source is a primary, this may lead to write contention.

On the target side, VReplication writes massive amount of data. If the target server is a primary with replicas, then the replicas may incur replication lag.

To address the above issues, VReplication uses the tablet throttler mechanism to push back both reads and writes.

Target throttling On the target side, VReplication wishes to consult the overall health of the target shard (there can be multiple shards to a VReplication workflow, and here we discuss the single shard at the end of a single VReplication stream). That shard may serve production traffic unrelated to VReplication. VReplication therefore consults the internal equivalent of `/throttler/check` when writing data to the shard's primary. This checks the replication lag on relevant replicas in the shard. The throttler will push back VReplication writes of both table-copy and changelog events.

Source throttling On the source side, VReplication only affects the single MySQL server it reads from, and has no impact on the overall shard. VStreamer, the source endpoint of VReplication, consults the equivalent of `/throttler/check-self`, which looks for replication lag on the source host.

As long as `check-self` fails, VStreamer will not read table data, nor will it pull events from the changelog.

Impact of throttling VReplication throttling is designed to give way to normal production traffic while operating in the background. Production traffic will see less contention. The downside is that VReplication can take longer to operate. Under high load in production VReplication may altogether stall, to resume when the load subsides.

Throttling will push back VReplication on replication lag. On systems where replication lag is normally high this can bring VReplication down to a grinding halt. In such systems consider configuring `-throttle_threshold` to a value that agrees with your constraints. The default throttling threshold is at `1` second replication lag.

VReplication V1 Commands

description: (Deprecated) V1 MoveTables and Reshard vtctld commands

DropSources

description: Cleans up after a MoveTables and Reshard workflow

```
DropSources [-dry_run] [-rename_tables] [-keep_data] <keyspace.workflow>
```

Description Once SwitchWrites has been run DropSources cleans up the source resources by deleting the source tables for a MoveTables workflow or source shards for a Reshard workflow. It also cleans up other artifacts of the workflow, deleting forward and reverse replication streams and blacklisted tables.

Warning: This command actually deletes data. We recommend that you run this with the -dry_run parameter first and reads its output so that you know which actions will be performed.

Parameters

-rename_tables optional
default all

Only applies for a MoveTables workflow. Instead of deleting the tables in the source it renames them using the template _<tableName>_old, the same scheme followed by pt-osc

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchReads.

-keep_data optional
default false

Usually, the source data (tables or shards) are deleted by Complete. If this flag is used, for MoveTables, source tables will not be deleted, for Reshard, source shards will not be dropped.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow to run VDiff on.

MoveTables

description: Move tables between keyspaces without downtime

```
MoveTables -v1 [-cells=<cells>] [-tablet_types=<source_tablet_types>] -workflow=<workflow>
            [-all] [-exclude=<tables>]  [-auto_start] [-stop_after_copy]
            <source_keyspace> <target_keyspace> <table_specs>
```

Description MoveTables is used to start a workflow to move one or more tables from an external database or an existing Vitess keyspace into a new Vitess keyspace. The target keyspace can be unsharded or sharded.

MoveTables is used typically for migrating data into Vitess or to implement vertical sharding. You might use the former when you first start using Vitess and the latter if you want to distribute your load across servers.

Parameters

-cells optional
default local cell

A comma-separated list of cell names or cell aliases. This list is used by VReplication to determine which cells should be used to pick a tablet for selecting data from the source keyspace.

Uses

- Improve performance by using picking a tablet in cells in network proximity with the target
- To reduce bandwidth costs by skipping cells that are in different availability zones
- Select cells where replica lags are lower

-tablet_types optional
default replica

A comma-separated list of tablet types that are used while picking a tablet for sourcing data. One or more from MASTER, REPLICA, RDONLY.

Uses

- To reduce the load on master tablets by using REPLICAs or RDONLYs
- Reducing lags by pointing to MASTER

workflow mandatory

Unique name for the MoveTables-initiated workflow, used in later commands to refer back to this workflow

source_keyspace mandatory

Name of existing keyspace that contains the tables to be moved

target_keyspace mandatory

Name of existing keyspace to which the tables will be moved

table_specs optional one of `table_specs` or `-all` needs to be specified

Either

- a comma-separated list of tables
  - if target keyspace is unsharded OR
  - if target keyspace is sharded AND the tables being moved are already defined in the target's vschema

  Example: `MoveTables -workflow=commerce2customer commerce customer customer,corder`

Or

- the JSON table section of the vschema for associated tables
  - if target keyspace is sharded AND
  - tables being moved are not yet present in the target's vschema

  Example: `MoveTables -workflow=commerce2customer commerce customer '{"t1":{"column_vindexes": [{"column": "id", "name": "hash"}]}}'`

-all **optional** cannot specify `table_specs` if `-all` is specified

Move all tables from the source keyspace.

-exclude **optional** only applies if `-all` is specified

If moving all tables, specifies tables to be skipped.

-auto_start **optional default** true

Normally the workflow starts immediately after it is created. If this flag is set to false then the workflow is in a Stopped state until you explicitly start it.

Uses

- allows updating the rows in `_vt.vreplication` after MoveTables has setup the streams. For example, you can add some filters to specific tables or change the projection clause to modify the values on the target. This provides an easier way to create simpler Materialize workflows by first using MoveTables with auto_start false, updating the BinlogSource as required by your Materialize and then start the workflow.
- changing the `copy_state` and/or `pos` values to restart a broken MoveTables workflow from a specific point of time.

-stop_after_copy **optional default** false

If set, the workflow will stop once the Copy phase has been completed i.e. once all tables have been copied and VReplication decides that the lag is small enough to start replicating, the workflow state will be set to Stopped.

Uses

- If you just want a consistent snapshot of all the tables you can set this flag. The workflow will stop once the copy is done and you can then mark the workflow as `Completed`

A MoveTables Workflow Once you select the set of tables to move from one keyspace to another you need to initiate a VReplication workflow as follows:

1. Initiate the migration using MoveTables
2. Monitor the workflow using Workflow or VExec
3. Confirm that data has been copied over correctly using VDiff
4. Start the cutover by routing all reads from your application to those tables using SwitchReads
5. Complete the cutover by routing all writes using SwitchWrites
6. Optionally clean up the source tables using DropSources

Common use cases for MoveTables

Adopting Vitess For those wanting to try out Vitess for the first time, MoveTables provides an easy way to route part of their workload to Vitess with the ability to migrate back at any time without any risk. You point a vttablet to your existing MySQL installation, spin up an unsharded Vitess cluster and use a MoveTables workflow to start serving some tables from Vitess. You can also go further and use a Reshard workflow to experiment with a sharded version of a part of your database.

See user guide for detailed steps

Vertical Sharding For existing Vitess users you can easily move one or more tables to another keyspace, either for balancing load or as preparation for sharding your tables.

See user guide which describes how MoveTables works in the local example provided in the Vitess repo.

More Reading

- MoveTables in practice

Reshard

description: split or merge shards in a keyspace

```
Reshard -v1 [-cells=<cells>] [-tablet_types=<source_tablet_types>] [-skip_schema_copy]
   <keyspace.workflow> <source_shards> <target_shards>
```

Description Reshard support horizontal sharding by letting you change the sharding ranges of your existing keyspace.

Parameters

-cells optional

Comma separated Cell(s) or CellAlias(es) to replicate from.

-tablet_types optional
default empty

Source Vitess tablet_type, or comma separated list of tablet types, that should be used for choosing source tablet(s) for the reshard.

Note: If replicating from master, you must explicitly use `-tablet_types=master`. If not specified, it defaults to the tablet type(s) specified by the `-vreplication_tablet_type` VTTablet command line flag. `-vreplication_tablet_type` defaults to replica.

-skip_schema_copy optional
default false

If true the source schema is copied to the target shards. If false, you need to create the tables before calling reshard.

keyspace.workflow mandatory

Name of keyspace being sharded and the associated workflow name, used in later commands to refer back to this reshard.

source_shards mandatory

Comma separated shard names to reshard from.

target_shards mandatory

Comma separated shard names to reshard to.

A Reshard Workflow Once you decide on the new resharding strategy for a keyspace, you need to initiate a VReplication workflow as follows:

1. Initiate the migration using Reshard
2. Monitor the workflow using Workflow or VExec
3. Confirm that data has been copied over correctly using VDiff
4. Start the cutover by routing all reads from your application to those tables using SwitchReads
5. Complete the cutover by routing all writes using SwitchWrites
6. Optionally cleanup the source tables using DropSources

SwitchReads

description: Route reads to target keyspace

```
SwitchReads  [-cells=c1,c2,...] [-reverse] -tablet_types={replica|rdonly} [-dry-run]
   <keyspace.workflow>
```

Description SwitchReads is used to switch reads for tables in a MoveTables workflow or for entire keyspace to the target keyspace in a Reshard workflow.

Parameters

-cells optional
default all

Comma separated list of cells or cell aliases in which reads should be switched in the target keyspace

-reverse optional
default false

When a workflow is setup the routing rules are setup so that reads/writes to the target shards still go to the source shard since the target is not yet setup. If SwitchReads is called without -reverse then the routing rules for the target keyspace are setup to actually use it. It is assumed that the workflow was successful and user is ready to use the target keyspace now.

However if, for any reason, we want to abort this workflow using the -reverse flag deletes the rules that were setup and vtgate will route the queries to this table to the the source table. There is no way to reverse the use of the -reverse flag other than by recreating the routing rules again using the vtctl ApplyRoutingRules command.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchReads.

-tablet_types mandatory

Tablet types to switch one or both or rdonly/replica (default “rdonly,replica”)

-tablet_type DEPRECATED

On which type of tablets should be reads be switched to the target keyspace. One of replica or rdonly. rdonly tables should be switched first before replica tablets.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow to SwitchReads for.

SwitchWrites

description: Route writes to target keyspace in a vreplication workflow

```
SwitchWrites  [-timeout=30s] [-cancel] [-reverse] [-reverse_replication=false]
   -tablet_types={replica|rdonly} [-dry-run] <keyspace.workflow>
```

Description SwitchWrites is used to switch writes for tables in a MoveTables workflow or for entire keyspace in the Reshard workflow away from the master in the source keyspace to the master in the target keyspace

Parameters

-timeout optional
default 30s

Specifies the maximum time to wait, in seconds, for vreplication to catch up on primary migrations. The migration will be cancelled on a timeout.

-filtered_replication_wait_time DEPRECATED
default 30s

SwitchWrites first stops writes on the source master and waits for the replication to the target to catchup with the point where the writes were stopped. If the wait time is longer than filtered_replication_wait_time the command will error out. For setups with high write qps you may need to increase this value.

-cancel optional
default false

If a previous SwitchWrites returned with an error you can restart it by running the command again (after fixing the issue that caused the failure) or the SwitchWrites can be canceled using this parameter. Only the SwitchWrites is cancelled: the workflow is set to Running so that replication continues.

-reverse optional
default false

Reverse a previous SwitchWrites serve from source

-reverse_replication optional
default true

SwitchWrites, by default, starts a reverse replication stream with the current target as the source, replicating back to the original source. This enables a quick and simple rollback. This reverse workflow name is that of the original workflow concatenated with _reverse.

-tablet_types mandatory

Tablet types to switch one or both or rdonly/replica (default “rdonly,replica”)

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchReads.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow to SwitchWrites for.

VReplication V2 commands

description: "VReplication V2 vtctld commands (experimental)"

— Cancel

description: Tear down a workflow where no traffic has been switched

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-keep_data] Cancel <targetKs.workflow>
```

Description `Cancel` can be used if a workflow was created in error or was misconfigured and you prefer to create a new workflow instead of fixing this one. Cancel can only be called if no traffic has been switched. It removes vreplication-related artifacts like rows from vreplication and copy_state in the side-car _vt database, routing rules, and blacklisted tables from the topo and, by default, the target tables/shards from the target keyspace.

Parameters

-keep_data optional
default false

Usually, the target data (tables or shards) are deleted by Cancel. If this flag is used with MoveTables, target tables will not be deleted and, with Reshard, target shards will not be dropped.

— Complete

description: Tear down a workflow after switching all traffic

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-keep_data] [-rename_tables] [-dry_run]
  Complete <targetKs.workflow>
```

Description Alert: This is a destructive command

`Complete` is used after all traffic has been switched. It removes vreplication-related artifacts like rows from vreplication and copy_state in the side-car _vt database, routing rules, and blacklisted tables (for MoveTables) from the topo. Also, by default, the source tables (or source shards) are dropped.

Parameters

-keep_data optional
default false

Usually, the source data (tables or shards) are deleted by Complete. If this flag is used, for MoveTables, source tables will not be deleted, for Reshard, source shards will not be dropped.

-rename_tables optional
default false

The rename_tables flag is applicable only for MoveTables. Tables are renamed instead of being deleted. Currently the new name is _<table_name>_old.

We use the same renaming logic used by `pt-online-schema-change`. Such tables are automatically skipped by vreplication if they exist on the source.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by Complete.

"— Create"

description: Initiate a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables [-source=<sourceKs>] [-tables=<tableSpecs>] [-cells=<cells>]
  [-tablet_types=<source_tablet_types>] [-all] [-exclude=<tables>]
   [-auto_start] [-stop_after_copy]
  Create <targetKs.workflow>

Reshard [-source_shards=<source_shards>] [-target_shards=<target_shards>]
  [-cells=<cells>] [-tablet_types=<source_tablet_types>]  [-skip_schema_copy]
  Create <keyspace.workflow>
```

Description `MoveTables/Reshard Create` sets up and creates a new workflow. The workflow name should not conflict with that of an existing workflow.

Parameters

-source mandatory

MoveTables only

Name of existing keyspace (the source keyspace) that contains the tables to be moved.

table_specs optional one of `table_specs` or `-all` needs to be specified

MoveTables only

Either

- a comma-separated list of tables
  - if target keyspace is unsharded OR
  - if target keyspace is sharded AND the tables being moved are already defined in the target's vschema

  Example: `MoveTables -source=commerce -tables=customer,corder Create customer.commerce2customer`

Or

- the JSON table section of the vschema for associated tables
  - if target keyspace is sharded AND
  - tables being moved are not yet present in the target's vschema

  Example: `MoveTables -source=commerce -tables='{"t1":{"column_vindexes": [{"column": "id", "name": "hash"}]}}}' Create customer.commerce2customer`

-cells optional
default local cell

A comma-separated list of cell names or cell aliases. This list is used by VReplication to determine which cells should be used to pick a tablet for selecting data from the source keyspace.

Uses

- Improve performance by picking a tablet in cells in network proximity with the target
- To reduce bandwidth costs by skipping cells that are in different availability zones
- Select cells where replica lags are lower

-tablet_types optional
default replica

A comma-separated list of tablet types that are used while picking a tablet for sourcing data. One or more from MASTER, REPLICA, RDONLY.

Uses

- To reduce the load on master tablets by using REPLICAs or RDONLYs
- Reducing lags by pointing to MASTER

-auto_start optional default true

Normally the workflow starts immediately after it is created. If this flag is set to false then the workflow is in a Stopped state until you explicitly start it.

Uses

- allows updating the rows in `_vt.vreplication` after MoveTables has setup the streams. For example, you can add some filters to specific tables or change the projection clause to modify the values on the target. This provides an easier way to create simpler Materialize workflows by first using MoveTables with auto_start false, updating the BinlogSource as required by your Materialize and then start the workflow.
- changing the `copy_state` and/or `pos` values to restart a broken MoveTables workflow from a specific point of time.

-stop_after_copy **optional default** false

If set, the workflow will stop once the Copy phase has been completed i.e. once all tables have been copied and VReplication decides that the lag is small enough to start replicating, the workflow state will be set to Stopped.

Uses

- If you just want a consistent snapshot of all the tables you can set this flag. The workflow will stop once the copy is done and you can then mark the workflow as `Completed`

-all **optional** cannot specify `table_specs` if `-all` is specified

MoveTables only

Move all tables from the source keyspace.

-exclude **optional** only applies if `-all` is specified

MoveTables only

If moving all tables, specifies tables to be skipped.

source_shards **mandatory**

Reshard only

Comma-separated shard names to reshard from.

Example: `Reshard -source_shards=0 -target_shards=-80,80- Create customer.reshard1to2`

target_shards **mandatory**

Reshard only

Comma-separated shard names to reshard to.

-skip_schema_copy **optional**
default false

Reshard only

If true the source schema is copied to the target shards. If false, you need to create the tables before calling reshard.

MoveTables

description: Move tables between keyspaces without downtime

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables <options> <action> <workflow identifier>
```

Description MoveTables is used to start and manage workflows to move one or more tables from an external database or an existing Vitess keyspace into a new Vitess keyspace. The target keyspace can be unsharded or sharded.

MoveTables is typically used for migrating data into Vitess or to implement vertical sharding. You might use the former when you first start using Vitess and the latter if you want to distribute your load across servers without sharding tables.

Parameters

action MoveTables is an "umbrella" command. The `action` sub-command defines the operation on the workflow.

options Each `action` has additional options/parameters that can be used to modify its behavior.

`actions` are common to both MoveTables and Reshard v2 workflows. Only the `create` action has different parameters, all other actions have common options and similar semantics. These actions are documented separately.

workflow identifier All workflows are identified by `targetKeyspace.workflow` where `targetKeyspace` is the name of the keyspace to which the tables are being moved. `workflow` is a name you assign to the MoveTables workflow to identify it.

The most basic MoveTables Workflow lifecycle

1. Initiate the migration using Create `MoveTables -source=<sourceKs> -tables=<tableSpecs> Create <targetKs.workflow>`
2. Monitor the workflow using Show or Progress `MoveTables Show <targetKs.workflow>` or `MoveTables Progress <targetKs.workflow>`
3. Confirm that data has been copied over correctly using VDiff
4. Cutover to the target keyspace with SwitchTraffic `MoveTables SwitchTraffic <targetKs.workflow>`
5. Cleanup vreplication artifacts and source tables with Complete `MoveTables Complete <targetKs.workflow>`

Common use cases for MoveTables

Adopting Vitess For those wanting to try out Vitess for the first time, MoveTables provides an easy way to route part of their workload to Vitess with the ability to migrate back at any time without any risk. You point a vttablet to your existing MySQL installation, spin up an unsharded Vitess cluster and use a MoveTables workflow to start serving some tables from Vitess. You can also go further and use a Reshard workflow to experiment with a sharded version of a part of your database.

See user guide for detailed steps

Vertical Sharding For existing Vitess users you can easily move one or more tables to another keyspace, either for balancing load or as preparation for sharding your tables.

See user guide which describes how MoveTables works in the local example provided in the Vitess repo.

More Reading

- MoveTables in practice

— Progress

description: Show copy progress and replication lag of a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard Progress <targetKs.workflow>
```

Description Workflows start in the copy state, (details in VReplication Internals), doing a bulk copy of the tables involved until they reach a low replication lag, after which we stream binlog events. Tables are copied sequentially.

`Progress` reports the progress of a workflow by showing the percentage of data copied across targets, if workflow is in copy state, and the replication lag between the target and the source once the copy phase is completed.

It is too expensive to get real-time row counts of tables, using _count(*)_, say. So we use the statistics available in the `information_schema` to approximate copy progress. This data can be significantly off (up to 50-60%) depending on the utilization of the underlying mysql server resources. You can manually run `analyze table` to update the statistics if so desired.

Reshard

description: Reshard a keyspace to achieve horizontal scaling

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
Reshard <options> <action> <workflow identifier>
```

Description Reshard is used to create and manage workflows to horizontally shard an existing keyspace. The source keyspace can be unsharded or sharded.

Parameters

action Reshard is an "umbrella" command. The `action` sub-command defines the operation on the workflow.

options Each `action` has additional options/parameters that can be used to modify its behavior.

`actions` are common to both MoveTables and Reshard v2 workflows. Only the `create` action has different parameters, all other actions have common options and similar semantics. These actions are documented separately.

workflow identifier All workflows are identified by `targetKeyspace.workflow` where `targetKeyspace` is the name of the keyspace to which the tables are being moved. `workflow` is a name you assign to the Reshard workflow to identify it.

The most basic Reshard Workflow lifecycle

1. Initiate the migration using Create `Reshard -source_shards=<source_shards> -target_shards=<target_shards> Create <keyspace.workflow>`
2. Monitor the workflow using Show or Progress `Reshard Show <keyspace.workflow>` or `Reshard Progress <keyspace.workflow>`
3. Confirm that data has been copied over correctly using VDiff
4. Cutover to the target keyspace with SwitchTraffic `Reshard SwitchTraffic <keyspace.workflow>`
5. Cleanup vreplication artifacts and source shards with Complete `Reshard Complete <keyspace.workflow>`

— ReverseTraffic

description: Reverse traffic from target to source

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-tablet_types <tablet_types_csv>] [-cells <cells>]
  [-timeout=timeoutDuration] [-dry_run]
  ReverseTraffic <targetKs.workflow>
```

Description `ReverseTraffic` switches traffic in the reverse direction for the tablet_types specified. The traffic should have been previously switched forward using SwitchTraffic for the cells/tablet_types specified.

Parameters

-cells optional
default all cells

A comma-separated list of cell names or cell aliases. Traffic will be reversed only in these cells for the specified tablet types.

-tablet_types optional
default all (replica,rdonly,master)

A comma-separated list of tablet types for which traffic is to be reversed. One or more from master, replica, rdonly.

-timeout optional
default 30s

For master tablets, ReverseTraffic first stops writes on the target master and waits for the replication to the source to catchup with the point where the writes were stopped. If the wait time is longer than timeout the command will error out. For setups with high write qps you may need to increase this value.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by ReverseTraffic.

— Show

description: Show information about a workflow

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard Show <targetKs.workflow>
```

Description `Show` displays useful information about a workflow. (At this time the Workflow Show command gives more information. This will be improved over time.)

— SwitchTraffic

description: Switch traffic from source to target

Experimental

This documentation is for a new (v2) set of vtctld commands. See RFC for more details.

```
MoveTables/Reshard [-tablet_types=<tablet_types>] [-cells=<cells>]
  [-timeout=timeoutDuration] [-reverse_replication] [-dry_run]
  SwitchTraffic <targetKs.workflow>
```

Description `SwitchTraffic` switches traffic forward for the tablet_types specified. This replaces the previous SwitchReads and SwitchWrites commands with a single one. It is also possible now to switch all traffic with just one command. Also, you can now switch replica, rdonly and primary traffic in any order: earlier you needed to first SwitchReads (for replicas and rdonly tablets) first before SwitchWrites.

Parameters

-cells optional
default all cells

A comma-separated list of cell names or cell aliases. Traffic will be switched only in these cells for the specified tablet types.

-tablet_types optional
default all (replica,rdonly,master)

A comma-separated list of tablet types for which traffic is to be switched. One or more from master,replica,rdonly.

-timeout optional
default 30s

For master tablets, SwitchTraffic first stops writes on the source master and waits for the replication to the target to catchup with the point where the writes were stopped. If the wait time is longer than timeout the command will error out. For setups with high write qps you may need to increase this value.

-reverse_replication optional
default true

SwitchTraffic for master tablet types, by default, starts a reverse replication stream with the current target as the source, replicating back to the original source. This enables a quick and simple rollback using ReverseTraffic. This reverse workflow name is that of the original workflow concatenated with _reverse.

If set to false these reverse replication streams will not be created and you will not be able to rollback once you have switched write traffic over to the target.

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchTraffic.

VDiff

description: Compare the source and target in a workflow to ensure integrity

```
VDiff  [-source_cell=<cell>] [-target_cell=<cell>] [-tablet_types=master,replica,rdonly]
       [-limit=<max rows to diff>] [-table=<table list>] [-format=json]
       [-filtered_replication_wait_time=30s] [-debug_query] [-only_pks] <keyspace.workflow>
```

Description VDiff does a row by row comparison of all tables associated with the workflow, diffing the source keyspace and the target keyspace and reporting counts of missing/extra/unmatched rows.

It is highly recommended that you do this before you finalize a workflow with SwitchWrites.

Parameters

-source_cell optional
default all

VDiff will choose a tablet from this cell to diff the source table(s) with the target tables

-target_cell optional
default all

VDiff will choose a tablet from this cell to diff the source table(s) with the target tables

-tablet_types optional
default master,replica,rdonly

A comma separated list of tablet types that are used while picking a tablet for sourcing data. One or more from MASTER, REPLICA, RDONLY.

-filtered_replication_wait_time optional
default 30s

VDiff finds the current position of the source master and then waits for the target replication to reach that position for *filtered_replication_wait_time*. If the target is much behind the source or if there is a high write qps on the source then this time will need to be increased.

-limit optional
default 9223372036854775807

Maximum number of rows to run vdiff on (across all tables specified). This limit is usually set while diffing a large table as a quick consistency check.

-tables optional
default all tables in the workflow

A comma separated list of tables to run vdiff on.

-format optional
default unstructured text output

Only other format supported is json

Example:

```
[{
  "ProcessedRows": 5,
  "MatchingRows": 5,
  "MismatchedRows": 0,
  "ExtraRowsSource": 0,
  "ExtraRowsTarget": 0
},{
  "ProcessedRows": 3,
```

```
  "MatchingRows": 3,
  "MismatchedRows": 0,
  "ExtraRowsSource": 0,
  "ExtraRowsTarget": 0
}]
```

-debug_query optional

Adds a MySQL query to the report that can be used for further debugging

-only_pks optional

When reporting missing rows, only show primary keys in the report.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow to run VDiff on.

```
$ vtctlclient VDiff customer.commerce2customer

Summary for corder: {ProcessedRows:10 MatchingRows:10 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
Summary for customer: {ProcessedRows:11 MatchingRows:11 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
```

Using VDiff with huge tables Currently VDiff runs within vtctd. Each VDiff will stream rows from all sources and targets and then compare them row by row after assembling the rows in order. Since there are no database transactions, VDiff will run much faster than the actual workflow. However, for huge tables (billions of rows or terabytes in size) this can take several hours or even days depending on the number of rows, row composition, server configurations and the topology of the cluster. If your sources and/or targets are across multiple cells, for example, this can slow down the VDiff considerably.

Actual VDiff speeds are of course dependent on several factors in your cluster. But as a reference, we have seen VDiffs run as fast as 400mrph (million rows per hour) (~9B rows/day) for tables with short rows, or as slow as 60mrph (~1.5B rows/day), for tables with larger width and complex columns.

You may need to use one or more of the following recommendations while running long VDiffs:

- If VDiff takes more than an hour `vtctlclient` will hit grpc/http timeouts of 1 hour. In that case you can use `vtctl` (the bundled `vctlclient` + `vtctld`) instead.
- VDiff also synchronizes sources and targets to get consistent snapshots. If you have a high write QPS then you may encounter timeouts during the sync. Use higher values of `-filtered_replication_wait_time` to prevent that, for example `-filtered_replication_wait_time=4h`.
- If VDiff takes more than a day set the `-wait-time` parameter, which is the maximum time a vtctl command can run for, to a value comfortably higher than the expected run time, for example `-wait_time=168h`.
- You can follow the progress of the command by tailing the vtctld logs. VDiff logs progress every 10 million rows. This can also give you an early indication of how long it will run for, allowing you to increase your settings if needed.

Note

- There is no throttling, so you might see an increased lag in the replica used as the source.
- VDiff is currently not resumable, so any timeouts or errors mean that you will need to rerun the entire VDiff again.
- VDiff runs one table at a time.

VReplication and VDiff performance improvements, resumability and throttling support are on the roadmap!

VExec

description: Wrapper on VReplicationExec to run query on all participating masters

```
VExec  [-dry_run] <keyspace.workflow> <query>
```

Description VExec is a wrapper over VReplicationExec. Given a workflow it executes the provided query on all masters in the target keyspace that participate in the workflow. Internally it calls VReplicationExec for running the query.

Parameters

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs the queries and the tablets on which the query would be run. by VExec.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow

sql query mandatory

SQL query to be run: validations are done to ensure that queries can be run only against vreplication tables. A limited set of queries are allowed.

```
vtctlclient VExec keyspace1.workflow1 'select * from _vt.vreplication'
```

Overview

description: VReplication features, design and options in a nutshell

VReplication is a core component of Vitess that can be used to compose many features. It can be used for the following use cases:

- **Resharding**: Legacy workflows of vertical and horizontal resharding. New workflows of resharding from an unsharded to a sharded keyspace and vice-versa. Resharding from an unsharded to an unsharded keyspace using a different vindex than the source keyspace.

- **Materialized Views**: You can specify a materialization rule that creates a view of the source table into a target keyspace. This materialization can use a different primary vindex than the source. It can also materialize a subset of the source columns, or add new expressions from the source. This view will be kept up-to-date in real time. One can also materialize reference tables onto all shards and have Vitess perform efficient local joins with those materialized tables.
- **Realtime rollups**: The materialization expression can include aggregation expressions in which case, Vitess will create a rolled up version of the source table which can be used for realtime analytics.
- **Backfilling lookup vindexes**: VReplication can be used to backfill a newly created lookup vindex. Workflows can be built to manage the switching from a backfill mode to the vindex itself keeping it up-to-date.
- **Schema deployment**: VReplication can be used to recreate the workflow performed by gh-ost and thereby support zero-downtime schema deployments in Vitess natively.
- **Data migration**: VReplication can be setup to migrate data from an existing system into Vitess. The replication could also be reversed after a cutover giving you the option to rollback a migration cutover if something went wrong, without losing the writes to the migration target.
- **Change notification**: The streamer component of VReplication can be used for the application or a systems operator to subscribe to change notification and use it to keep downstream systems up-to-date with the source.

The VReplication feature itself is a fairly low level one that is expected to be used as a building block for the above use cases. However, it is still possible to directly issue commands to perform some of the activities.

Feature description

VReplication works as a stream or set of streams. Each stream establishes a replication from a source keyspace/shard into a target keyspace/shard.

A given stream can replicate multiple tables. For each table, you can specify a `SELECT` statement that represents both the transformation rule and the filtering rule. The `SELECT` expressions specify the transformation, and the `WHERE` clause specifies the filtering.

The `SELECT` expressions can be any non-aggregate MySQL expression, or they can also be `COUNT` or `SUM` as aggregate expressions. Aggregate expressions combined with the corresponding `GROUP BY` clauses will allow you to materialize real-time rollups of the source table, which can be used for analytics. The target table can have a different name from the source.

For a sharded system like Vitess, multiple VReplication streams may be needed to achieve the objective. This is because there can be multiple source shards and multiple destination shards, and the relationship between them may not be one to one.

VReplication performs the following essential functions:

- Copy data from the source to the destination table in a consistent fashion. For a large table, this copy can be long-running. It can be interrupted and resumed. If interrupted, VReplication can keep the copied portion up-to-date with respect to the source, and it can resume the copy process at a point that is consistent with the current replication position.
- After copying is finished, it can continuously replicate the data from the source to destination.
- The copying rule can be expressed as a `SELECT` statement. The statement should be simple enough that the materialized table can be kept up-to-date from the data coming from the binlog. For example, joins in the `SELECT` statement are not supported.
- Correctness verification (to be implemented): VReplication can verify that the target table is an exact representation of the `SELECT` statement from the source by capturing consistent snapshots of the source and target and comparing them against each other. This step can be done without the need to create special snapshot replicas.
- Journaling: If there is any kind of traffic cut-over where we start writing to a different table than we used to before, VReplication will save the current binlog positions into a journal table. This can be used by other streams to resume replication from the new source.
- Routing rules: Although this feature is itself not a direct functionality of VReplication, it works hand in hand with it. It allows you to specify sophisticated rules about where to route queries depending on the type of workflow being performed. For example, it can be used to control the cut-over during resharding. In the case of materialized views, it can be used to establish equivalence of tables, which will allow VTGate to compute the most optimal plans given the available options.

VReplicationExec

The `VReplicationExec` command is a low-level command used to manage VReplication streams. The commands are issued as SQL statements. For example, a `SELECT` can be used to see the current list of streams. An `INSERT` can be used to create one, etc. By design, the metadata for vreplication streams are stored in a table called `vreplication` in the `_vt` sidecar database. VReplication uses a 'pull' model. This means that a stream is created on the target side, and the target pulls the data by finding the appropriate source. As a result, this metadata is stored on the target shard.

The table schema is as follows:

```
CREATE TABLE _vt.vreplication (
  id INT AUTO_INCREMENT,
  workflow VARBINARY(1000),
  source VARBINARY(10000) NOT NULL,
  pos VARBINARY(10000) NOT NULL,
  stop_pos VARBINARY(10000) DEFAULT NULL,
  max_tps BIGINT(20) NOT NULL,
  max_replication_lag BIGINT(20) NOT NULL,
  cell VARBINARY(1000) DEFAULT NULL,
  tablet_types VARBINARY(100) DEFAULT NULL,
  time_updated BIGINT(20) NOT NULL,
  transaction_timestamp BIGINT(20) NOT NULL,
  state VARBINARY(100) NOT NULL,
  message VARBINARY(1000) DEFAULT NULL,
  db_name VARBINARY(255) NOT NULL,
  PRIMARY KEY (id)
)
```

The fields are explained in the following section.

This is the syntax of the command:

```
VReplicationExec [-json] <tablet alias> <sql command>
```

Here's an example of the command to list all existing streams for a given tablet.

```
vtctlclient -server localhost:15999 VReplicationExec 'tablet-100' 'select * from
   _vt.vreplication'
```

Creating a stream It's generally easier to send the VReplication command programmatically instead of a bash script. This is because of the number of nested encodings involved:

- One of the arguments is an SQL statement, which can contain quoted strings as values.
- One of the strings in the SQL statement is a string encoded protobuf, which can contain quotes.
- One of the parameters within the protobuf is an SQL `SELECT` expression for the materialized view.

However, you can use vreplgen.go to generate a fully escaped bash command.

Alternately, you can use a python program. Here's an example:

```
cmd = [
  'vtctlclient',
  '-server',
  'localhost:15999',
  'VReplicationExec',
  'test-200',
  """insert into _vt.vreplication
```

```
  (db_name, source, pos, max_tps, max_replication_lag, tablet_types, time_updated,
     transaction_timestamp, state) values
  ('vt_keyspace', 'keyspace:"lookup" shard:"0" filter:<rules:<match:"uproduct"
     filter:"select * from product" > >', '', 99999, 99999, 'master', 0, 0, 'Running')""",
]
```

The first argument to the command is the master tablet id of the target keyspace/shard for the VReplication stream.

The second argument is the SQL command. To start a new stream, you need an insert statement. The parameters are as follows:

- `db_name`: This name must match the name of the MySQL database. In the future, this will not be required, and will be automatically filled in by the vttablet.
- `source`: The protobuf representation of the stream source, explained below.
- `pos`: For a brand new stream, this should be empty. To start from a specific position, a flavor-encoded position must be specified. A typical position would look like this `MySQL56/ac6c45eb-71c2-11e9-92ea-0a580a1c1026:1-1296`.
- `max_tps`: 99999, reserved.
- `max_replication_lag`: 99999, reserved.
- `tablet_types`: specifies a comma separated list of tablet types to replicate from. If empty, the default tablet type specified by the `-vreplication_tablet_type` command line flag is used, which in turn defaults to 'REPLICA'.
- `time_updated`: 0, reserved.
- `transaction_timestamp`: 0, reserved.
- `state`: 'Init', 'Copying', 'Running', 'Stopped', 'Error'.
- `cell`: is an optional parameter that specifies the cell from which the stream can be sourced. If no cell is specified, the default is the local/current cell.

The source field The source field is a proto-encoding of the following structure:

```
message BinlogSource {
  // the source keyspace
  string keyspace = 1;
  // the source shard
  string shard = 2;
  // list of filtering rules
  Filter filter = 6;
  // what to do if a DDL is encountered
  OnDDLAction on_ddl = 7;
}

message Filter {
  repeated Rule rules = 1;
}

message Rule {
  // match can be a table name or a regular expression
  // delineated by '/' and '/'.
  string match = 1;
  // filter can be an empty string or keyrange if the match
  // is a regular expression. Otherwise, it must be a select
  // query.
  string filter = 2;
}

enum OnDDLAction {
  IGNORE = 0;
  STOP = 1;
  EXEC = 2;
  EXEC_IGNORE = 3;
```

```
}
```

Here are some examples of proto encodings:

```
keyspace:"lookup" shard:"0" filter:<rules:<match:"uproduct" filter:"select * from product"
   > >
```

Meaning: copy and replicate all columns and rows of product from the source table `lookup/0.product` to the `uproduct` table in target keyspace.

```
keyspace:"user" shard:"-80" filter:<rules:<match:"morder" filter:"select * from uorder
   where in_keyrange(mname, \\'unicode_loose_md5\\', \\'-80\\')" > >
```

The double-backslash for the strings inside the select will first be escaped by the python script, which will cause the expression to internally be `\'unicode_loose_md5\'`. Since the entire source is surrounded by single quotes when being sent as a value inside the outer insert statement, the single `\` will escape the single quotes that follow. The final value in the source will therefore be:

```
keyspace:"user" shard:"-80" filter:<rules:<match:"morder" filter:"select * from uorder
   where in_keyrange(mname, 'unicode_loose_md5', '-80')" > >
```

Meaning: copy and replicate all columns of the source table `user/-80.uorder` where `unicode_loose_md5(mname)` is within `-80` keyrange, to the `morder` table in the the target keyspace.

This particular stream generally wouldn't make sense in isolation. This would typically be one of a set of four streams that combine to create a materialized view of `uorder` from the `user` keyspace into the target (`merchant`) keyspace, but sharded by using `mname` as the primary vindex. The vindex used would be `unicode_loose_md5` which should also match the primary vindex of other tables in the target keyspace.

```
keyspace:"user" shard:"-80" filter:<rules:<match:"sales" filter:"select pid, count(*) as
   kount, sum(price) as amount from uorder group by pid" > >
```

Meaning: create a materialized view of `user/-80.uorder` into `sales` of the target keyspace using the expression: `select pid, count(*)as kount, sum(price)as amount from uorder group by pid`.

This represents only one stream from source shard `-80`. Presumably, there will be one more for the other `-80` shard.

The 'SELECT' features The `SELECT` statement has the following features (and restrictions):

- The `SELECT` expressions can be any deterministic MySQL expression. Subqueries and joins are not supported. Among aggregate expressions, only `count(*)` and `sum(col)` are supported.
- The `WHERE` clause can only contain:
  - Integer or string equality comparisons, like `customer_id = 42 AND somecol='newval'`.
  - The `in_keyrange` construct. It has two forms:
    - `in_keyrange('-80')`: The row's source keyrange matched against `-80`.
    - `in_keyrange(col, 'vindex_func', '-80')`: The keyrange is computed using the specified Vindex function as `vindex_func(col)` and matched against `-80`.
- `GROUP BY`: can be specified if using aggregations. The `GROUP BY` expressions are expected to cover the non-aggregated columns just like regular SQL requires.
- No other constructs like `ORDER BY`, `LIMIT`, etc. are allowed.

The pos field For starting a brand new vreplication stream, the `pos` field must be empty. The empty string signifies that there's no starting point for the vreplication. This causes VReplication to copy the contents of the source table first, and then start the replication.

For large tables, this is done in chunks. After each chunk is copied, replication is resumed until it's caught up. VReplication ensures that only changes that affect existing rows are applied. Following this another chunk is copied, and so on, until all tables are completed. After that, replication runs indefinitely until the VReplication stream is stopped or deleted.

It is a shared row The `vreplication` table row is shared between the operator and Vreplication itself. Once the row is created, the VReplication stream updates various fields of the row to save and report on its own status. For example, the `pos` field is continuously updated as it makes forward progress.

While copying, the `state` field will be `Init` or `Copying`.

Updating a stream You can change any field of the stream by issuing a `VReplicationExec` with an SQL `UPDATE` statement. You are required to specify the id of the row you intend to update. You can only update one row at a time.

Typically, you can update the row and change the state to `Stopped` to stop a stream, or to `Running` to restart a stopped stream.

You can also update the row to set a `stop_pos`, which will make the replication stop once it reaches the specified position.

Deleting a stream You can delete a stream by issuing a `DELETE` statement. This will stop the replication and delete the row. This statement is destructive. All data about the replication state will be permanently deleted. Note that the target table will be left as-is, potentially partially copied, and needs to be cleaned up separately, if desired.

Other properties of VReplication

Fast replay VReplication has the capability to batch transactions if the send rate of the source exceeds the replay rate of the destination. This allows it to catch up very quickly when there is a backlog. Load tests have shown a 3-20X improvement over traditional MySQL replication depending on the workload.

Accurate lag tracking The source vttablet sends its current time along with every event. This allows the target to correct for clock skew while estimating replication lag. Additionally, the source starts sending heartbeats if there is nothing to send. If the target receives no events from the source at all, it knows that it's definitely lagged and starts reporting itself accordingly.

Self-replication VReplication allows you to set the source keyspace/shard to be the same as the target. This is especially useful for performing schema rollouts: you can create the target table with the intended schema and vreplicate from the source table to the new target. Once caught up, you can cutover to write to the target table. In this situation, an apply on the target generates a binlog event that will be picked up by the source and sent to the target. Typically, it will be an empty transaction. In such cases, the target does not generally apply these transactions, because such an application will generate yet another event. However, there are situations where one needs to apply empty transactions, especially if it's a required stopping point. VReplication can differentiate between these situations and apply events only as needed.

Deadlocks and lock wait timeouts It is possible that multiple streams can conflict with each other and cause deadlocks or lock waits. When such things happen, VReplication silently retries such transactions without reporting an error. It does increment a counter so that the frequency of such occurrences can be tracked.

Automatic retries If any other error is encountered, the replication is retried after a short wait. Each time, the stream searches from the full list of available sources and picks one at random.

on_ddl The source specification allows you to specify a value for `on_ddl`. This allows you to specify what to do with DDL SQL statements when they are encountered in the replication stream from the source. The values can be as follows:

- `IGNORE`: Ignore all DDLs (this is also the default, if a value for `on_ddl` is not provided).
- `STOP`: Stop when DDL is encountered. This allows you to make any necessary changes to the target. Once changes are made, updating the state to `Running` will cause VReplication to continue from just after the point where it encountered the DDL.
- `EXEC`: Apply the DDL, but stop if an error is encountered while applying it.
- `EXEC_IGNORE`: Apply the DDL, but ignore any errors and continue replicating.

Failover continuation If a failover is performed on the target keyspace/shard, the new master will automatically resume VReplication from where the previous master left off.

Throttling VReplication throttles operation when the source or target appear to be overloaded, indicated by replication lag. See throttling

Monitoring and troubleshooting

VTTablet /debug/status The first place to look at is the `/debug/status` page of the target master vttablet. The bottom of the page shows the status of all the VReplication streams.

Typically, if there is a problem, the `Last Message` column will display the error. Sometimes, it's possible that the stream cannot find a source. If so, the `Source Tablet` would be empty.

VTTablet logfile If the errors are not clear or if they keep disappearing, the VTTablet logfile will contain information about what it's been doing with each stream.

VReplicationExec select The current status of the streams can also be fetched by issuing a VReplicationExec command with `select * from _vt.vreplication`.

Monitoring variables VReplication also reports the following variables that can be scraped by monitoring tools like prometheus:

- VReplicationStreamCount: Number of VReplication streams.
- VReplicationSecondsBehindMasterMax: Max vreplication seconds behind master.
- VReplicationSecondsBehindMaster: vreplication seconds behind master per stream.
- VReplicationSource: The source for each VReplication stream.
- VReplicationSourceTablet: The source tablet for each VReplication stream.

Thresholds and alerts can be set to draw attention to potential problems.

VReplicationExec

description: Low level command to run a query on vreplication related tables

```
VReplicationExec [-json] <tablet alias> <sql command>
```

Description The VReplicationExec command is used to view or manage vreplication streams. More details are here. You would typically use one of the higher-level commands like the WorkFlow command accomplish the same task.

Parameters

-json optional

The output of the command is json formatted: to be readable by scripts.

tablet alias mandatory

Id of the target tablet on which to run the sql query, specified using the vitess tablet id format cell-uid (see example below).

sql query mandatory

SQL query which will be run: validations are done to ensure that queries can be run only against vreplication tables. A limited set of queries are allowed.

```
vtctlclient  VReplicationExec 'zone1-100' 'select * from _vt.vreplication'
```

Workflow

description: Wrapper on VExec to perform common actions on a workflow

```
Workflow  [-dry_run] <keyspace.workflow> <action>
```

Description Workflow is a convenience command for useful actions on a workflow that you can use instead of actually specifying a query to VExec.

Parameters

-dry-run optional
default false

You can do a dry run where no actual action is taken but the command logs all the actions that would be taken by SwitchReads.

keyspace.workflow mandatory

Name of target keyspace and the associated workflow to SwitchWrites for.

action mandatory

action is one of

- **stop**: sets the state of the workflow to Stopped: no further vreplication will happen until workflow is restarted
- **start**: restarts a Stopped workflows
- **delete**: removes the entries for this workflow in _vt.vreplication
- **show**: returns a JSON object with details about the associated shards and also with all the columns from the _vt.vreplication table
- **listall**: returns a comma separated list of all *running* workflows in a keyspace

```
vtctlclient  Workflow keyspace1.workflow1 stop
vtctlclient  Workflow keyspace1.workflow1 show
vtctlclient  Workflow keyspace1 listall
```

Resources

description: Additional resources including Presentations and Roadmap

Presentations and Videos

Rawkode Live

Introduction to Vitess. Deepthi Sigireddi, Alkin Tezuysal - PlanetScale, David McKay @rawkode

{{< youtube id="Cwpu0y119I0" autoplay="false" >}}

Percona FOSS Talks 2021

All Things Vitess, MySQL, and Sailing: Alkin Tezuysal on Percona's HOSS Talks FOSS!

{{< youtube id="8e3IC-lAgXs" autoplay="false" >}}

FOSDEM 2021

- Open Source Database Infrastructure with Vitess - Shlomi Noach, PlanetScale
- How to shard MariaDB like a Pro? - Alkin Tezuysal, PlanetScale

KubeCon NA 2020

KubeCon featured several Vitess talks, including:

- Keynote: CNCF Projects Update - CNCF
- Vitess: Introduction and New Features - Sugu Sougoumarane & Deepthi Sigireddi, PlanetScale
- Introduction to Vitess - Deepthi Sigireddi, PlanetScale
- Codename VIFL - How to Migrate MySQL Database Clusters to Vitess - Guido Iaquinti & Rafael Chacon , Slack

Perconalive Europe Online 2020

Vitess Online Schema Migration Automation - Shlomi Noach, PlanetScale

{{< youtube id="iQYZ0dRe7O8" autoplay="false" >}}

CNCF Webinars 2020

- Transactional microservices with Vitess - coordination without state - Dan Kozlowski

Quarantine 2020 Database Talks, Carnegie Mellon University

Query Planning for a Sharded System like Vitess - Sugu Sougoumarane - CTO,Planetscale

{{< youtube id="TZb6m5ACIsA" autoplay="false" >}}

KubeCon Europe 2020

KubeCon featured several Vitess talks, including:

- Vitess Intro: How to Migrate a MySQL Database to Vitess - Sugu Sougoumarane & Morgan Tocker
- Zero Downtime Data Relocation with Vitess - Liz van Dijk & Deepthi Sigireddi, PlanetScale
- Multicloud Vitess over Network Service Mesh - Tim Swanson, Cisco & John Watson, PlanetScale

Online Demo

A demo of the new Vitess-native Orchestrator. Sugu Sougoumarane - CTO, PlanetScale

{{< youtube id="JVHi2oN3lxM" autoplay="false" >}}

Perconalive NA Online 2020

Unbounded scale: Using Vitess for scaling write intensive workloads. Morgan Tocker , Daniel Guzman Burgos

{{< youtube id="kHPSp8coQ5g" autoplay="false" >}}

CNCF Webinar 2020

Lizz van Dijk demonstrates how to migrate from a regular MySQL release to Vitess.

{{< youtube id="W8VbiXo39Ik" autoplay="false" >}}

MySQL Pre-FOSDEM Day 2020

Lizz van Dijk presents an introduction to Vitess for MySQL users.

QCon 2019

Sugu Sougoumarane gives an overview of the salient features of Vitess.

Video and Slides

KubeCon San Diego 2019

KubeCon featured several Vitess talks, including:

- Scaling Resilient Systems: A Journey into Slack's Database Service - Rafael Chacon & Guido Iaquinti, Slack
- How to Migrate a MySQL Database to Vitess - Sugu Sougoumarane & Morgan Tocker, PlanetScale
- Building a Database as a Service on Kubernetes - Abhi Vaidyanatha & Lucy Burns, PlanetScale
- Vitess: Stateless Storage in the Cloud - Sugu Sougoumarane, PlanetScale
- Geo-partitioning with Vitess - Deepthi Sigireddi & Jitendra Vaidya, PlanetScale
- Gone in 60 Minutes: Migrating 20 TB from AKS to GKE in an Hour with Vitess - Derek Perkins, Nozzle

Vitess was also featured during the CNCF project updates keynote!

Highload 2019

Sugu Sougoumarane presents an overview of Vitess at Highload in Moscow.

{{< pdf src="/ViewerJS/#../files/2019-sugu-highload.pdf" >}}

Utah Kubernetes Meetup 2019

Jiten Vaidya shows how you can extend Vitess to create jurisdiction-aware database clusters.

{{< pdf src="/ViewerJS/#../files/2019-jiten-utah.pdf" >}}

CNCF Meetup Paris 2019

Sugu Sougoumarane and Morgan Tocker present a three hour Vitess workshop on Kubernetes.

{{< pdf src="/ViewerJS/#../files/2019-paris-cncf.pdf" >}}

Percona Live Europe 2019

My First 90 Days with Vitess

Morgan Tocker talks about his adventures in Vitess, after having come from a MySQL background.

{{< pdf src="/ViewerJS/#../files/2019-morgan-percona-eu.pdf" >}}

Sharded MySQL on Kubernetes

Sugu Sougoumarane presents an overview of running sharded MySQL on Kubernetes.

{{< pdf src="/ViewerJS/#../files/2019-sugu-percona-eu.pdf" >}}

Vitess Meetup 2019 @ Slack HQ

Vitess: New and Coming Soon!

Deepthi Sigireddi shares new features recently introduced in Vitess, and what's on the roadmap moving forward.

{{< pdf src="/ViewerJS/#../files/2019-deepthi-vitess-meetup.pdf" >}}

Deploying multi-cell Vitess

Rafael Chacon Vivas describes how Vitess is used in Slack.

{{< pdf src="/ViewerJS/#../files/2019-rafael-vitess-meetup.pdf" >}}

Vitess at Pinterest

David Weitzman provides an overview of how Vitess is used at Pinterest.

{{< youtube id="1cWWlaqlia8" autoplay="false" >}}

No more Regrets

Sugu Sougoumarane demonstrates new features coming to VReplication.

{{< youtube id="B1Nrtptjtcs" autoplay="false" >}}

Cloud Native Show 2019

Vitess at scale - how Nozzle.io runs MySQL on Kubernetes

Derek Perkins joins the Cloud Native show and explains how Nozzle uses Vitess.

Listen to Podcast

CNCF Webinar 2019

Vitess: Sharded MySQL on Kubernetes

Sugu Sougoumarane provides an overview of Vitess for Kubernetes users.

{{< youtube id="E6H4bgJ3Z6c" autoplay="false" >}}

Kubecon China 2019

How JD.Com runs the World's Largest Vitess

Xuhaihua and Jin Ke Xie present on their experience operating the largest known Vitess cluster, two years in.

{{< youtube id="qww4UVNG3Io" autoplay="false" >}}

RootConf 2019

OLTP or OLAP: why not both?

Jiten Vaidya from PlanetScale explains how you can use both OLTP and OLAP on Vitess.

{{< youtube id="bhzJJF82mFc" autoplay="false" >}}

Kubecon 19 Barcelona

Vitess Deep Dive

Jiten Vaidya and Dan Kozlowski from PlanetScale deep dive on Vitess.

{{< youtube id="OZl4HrB9p-8" autoplay="false" >}}

Percona Live Austin 2019

Vitess: Running Sharded MySQL on Kubernetes

Sugu Sougoumarane shows how you can run sharded MySQL on Kubernetes.

{{< youtube id="v7oxiVmGXp4" autoplay="false" >}}

MySQL, Kubernetes, Business & Enterprise

David Cohen (Intel), Steve Shaw (Intel) and Jiten Vaidya (PlanetScale) discuss Open Source cloud native databases.

View Talk Abstract and Slides

Velocity New York 2018

Smooth scaling: Slack's journey toward a new database

Slack has experienced tremendous growth for a young company, serving over nine million weekly active customers. But with great growth comes greater growth pains. Slack's rapid growth over the last few years outpaced the scaling capacity of its original sharded MySQL database, which negatively impacted the company's customers and engineers.

Ameet Kotian explains how a small team of engineers embarked on a journey for the right database solution, which eventually led them to Vitess, a powerful open source database cluster solution for MySQL. Vitess combines the features of MySQL with the scalability of a NoSQL database. It has been serving Youtube's traffic for numerous years and has a strong community.

Although Vitess meets a lot of Slack's needs, it's not an out-of-the-box solution. Ameet shares how the journey to Vitess was planned and executed, with little customer impact, in the face of piling operational challenges, such as AWS issues, MySQL replication, automatic failovers, deployments strategies, and so forth. Ameet also covers Vitess's architecture, trade-offs, and what the future of Vitess looks like at Slack.

Ameet Kotkian, senior storage operations engineer at Slack, shows us how Slack uses Vitess.

Percona Live Europe 2017

Migrating to Vitess at (Slack) Scale

Slack is embarking on a major migration of the MySQL infrastructure at the core of our service to use Vitess' flexible sharding and management instead of our simple application-based shard routing and manual administration. This effort is driven by the need for an architecture that scales to meet the growing demands of our largest customers and features under the pressure to maintain a stable and performant service that executes billions of MySQL transactions per hour. This talk will present the driving motivations behind the change, why Vitess won out as the best option, and how we went about laying the groundwork for the switch. Finally, we will discuss some challenges and surprises (both good and bad) found during our initial migration efforts, and suggest some ways in which the Vitess ecosystem can improve that will aid future migration efforts.

Michael Demmer shows us how, at Percona Live Europe 2017.

{{< pdf src="/ViewerJS/#../files/2017-demmer-percona.pdf" >}}

Vitess Deep Dive sessions

Start with session 1 and work your way through the playlist. This series focuses on the V3 engine of VTGate.

{{< youtube id="6yOjF7qhmyY" autoplay="false" >}}

Percona Live 2016

Sugu and Anthony showed what it looks like to use Vitess now that Keyspace IDs can be completely hidden from the application. They gave a live demo of resharding the Guestbook sample app, which now knows nothing about shards, and explained how new features in VTGate make all of this possible.

{{< pdf src="/ViewerJS/#../files/percona-2016.pdf" >}}

CoreOS Meetup, January 2016

Vitess team member Anthony Yeh's talk at the January 2016 CoreOS Meetup discussed challenges and techniques for running distributed databases within Kubernetes, followed by a deep dive into the design trade-offs of the Vitess on Kubernetes deployment templates.

Please note the Vitess on Kubernetes deployment templates were removed as of February 27, 2020.

{{< pdf src="/ViewerJS/#../files/coreos-meetup-2016-01-27.pdf" >}}

Oracle OpenWorld 2015

Vitess team member Anthony Yeh's talk at Oracle OpenWorld 2015 focused on what the Cloud Native Computing paradigm means when applied to MySQL in the cloud. The talk also included a deep dive into transparent, live resharding, one of the key features of Vitess that makes it well-adapted for a Cloud Native environment.

{{< pdf src="/ViewerJS/#../files/openworld-2015-vitess.pdf" >}}

Percona Live 2015

Vitess team member Anthony Yeh's talk at Percona Live 2015 provided an overview of Vitess as well as an explanation of how Vitess has evolved to live in a containerized world with Kubernetes and Docker.

{{< pdf src="/ViewerJS/#../files/percona-2015-vitess-and-kubernetes.pdf" >}}

Google I/O 2014 - Scaling with Go: YouTube's Vitess

In this talk, Sugu Sougoumarane from the Vitess team talks about how Vitess solved YouTube's scalability problems as well as about tips and techniques used to scale with Go.

{{< youtube id="midJ6b1LkA0" autoplay="false" >}}

Vitess Roadmap

description: Upcoming features planned for development

As an open source project, Vitess is developed by a community of contributors. Many of the contributors run Vitess in production, and add features to address their specific pain points. As a result of this, we can not guarantee features listed here will be implemented in any specific order.

{{< info >}} If you have a specific question about the Roadmap, we recommend posting in our Slack channel, click the Slack icon in the top right to join. This is a very active community forum and a great place to interact with other users. {{< /info >}}

Last Updated: May 25, 2021

Short Term (1-3 months)

- Improve Documentation
- Improve Usability
- Support more MySQL Syntax (improve compatibility as a drop-in replacement)
  - Certify popular frameworks like Ruby on Rails, Django etc.
- Nightly benchmarking (regression testing)
- VReplication
  - Performance
  - Usability
  - Online schema changes
- Technical debt
  - grpc
  - protobuf
  - golang 1.16

Medium Term (3-9 months)

- MySQL compatibility
  - More frameworks
- Query Planning improvements
  - Performance
  - More supported queries
- Schema changes
  - Usability
- VSchema improvements
  - Vtgates auto-detect schema changes

- Vitess-native unplanned failovers (vtorc)
- Pluggable durability policies (vtorc)
- Rewrite of vtctld UI including visualization of VReplication
- VReplication throttling
- Topology Service: Reduce dependencies on the topology service. i.e. Vitess should be operable normally even if topology service is down for several hours. Topology service should be used only for passive discovery.

Troubleshoot

description: Debug common issues with Vitess

If there is a problem in the system, one or many alerts would typically fire. If a problem was found through means other than an alert, then the alert system needs to be iterated upon.

When an alert fires, you have the following sources of information to perform your investigation:

- Alert values
- Graphs
- Diagnostic URLs
- Log files

```
select @@version;
```

Elevated query latency on master

description: Debug common issues with Vitess

Elevated query latency on master

Diagnosis 1: Inspect the graphs to see if QPS has gone up. If yes, drill down on the more detailed QPS graphs to see which table, or user caused the increase. If a table is identified, look at /debug/queryz for queries on that table.

Action: Inform engineer about about toxic query. If it's a specific user, you can stop their job or throttle them to keep the load manageable. As a last resort, blacklist query to allow the rest of the system to stay healthy.

Diagnosis 2: QPS did not go up, only latency did. Inspect the per-table latency graphs. If it's a specific table, then it's most likely a long-running low QPS query that's skewing the numbers. Identify the culprit query and take necessary steps to get it optimized. Such queries usually do not cause outage. So, there may not be a need to take extreme measures.

Diagnosis 3: Latency seems to be up across the board. Inspect transaction latency. If this has gone up, then something is causing MySQL to run too many concurrent transactions which causes slow-down. See if there are any tx pool full errors. If there is an increase, the INFO logs will dump info about all transactions. From there, you should be able to if a specific sequence of statements is causing the problem. Once that is identified, find out the root cause. It could be network issues, or it could be a recent change in app behavior.

Diagnosis 4: No particular transaction seems to be the culprit. Nothing seems to have changed in any of the requests. Look at system variables to see if there are hardware faults. Is the disk latency too high? Are there memory parity errors? If so, you may have to failover to a new machine.

Master starts up read-only

description: Debug common issues with Vitess

Master starts up read-only

To prevent accidentally accepting writes, our default my.cnf settings tell MySQL to always start up read-only. If the master MySQL gets restarted, it will thus come back read-only until you intervene to confirm that it should accept writes. You can use the `SetReadWrite` command to do that.

However, usually if something unexpected happens to the master, it's better to reparent to a different replica with `EmergencyReparentShard`. If you need to do planned maintenance on the master, it's best to first reparent to another replica with `PlannedReparentShard`.

Vitess sees the wrong tablet as master

description: Debug common issues with Vitess

Vitess sees the wrong tablet as master

If you do a failover manually (not through Vitess), you'll need to tell Vitess which tablet corresponds to the new master MySQL. Until then, writes will fail since they'll be routed to a read-only replica (the old master). Use the `TabletExternallyReparented` command to tell Vitess the new master tablet for a shard.

Tools like Orchestrator can be configured to call this automatically when a failover occurs. See our sample orchestrator.conf.json for an example of this.

User Guides

description: Task-based guides for common usage scenarios

We recommend running through a get started on your favorite platform before running through user guides.

Advanced Configuration

description: User guides covering advanced configuration concepts

Authorization

A common question is how to enforce fine-grained access control in Vitess. This question comes up because Vitess uses connection pooling with fixed MySQL users at the VTTablet level, and implements its own authentication at the VTGate level. As a result, you cannot use the normal MySQL GRANTs system to give certain application-level MySQL users more or less permissions than others.

The MySQL GRANT system is very extensive, and we have not reimplemented all of this functionality in Vitess. What we have done is to enable you to provide authorization via table-level ACLs, with a few basic characteristics:

- Individual users can be assigned 3 levels of permissions:
  - Read (corresponding to read DML, e.g. `SELECT`)
  - Write (corresponding to write DML, e.g. `INSERT`, `UPDATE`, `DELETE`)
  - Admin (corresponding to DDL, e.g. `ALTER TABLE`)
- Permissions are applied on a specified set of tables, which can be enumerated or specified by regex.

VTTablet parameters for table ACLs

Note that the Vitess authorization via ACLs are applied at the VTTablet level, as opposed to on VTGate, where authentication is enforced. There are a number of VTTablet command line parameters that control the behavior of ACLs. Let's review these:

- `-enforce-tableacl-config`: Set this to `true` to ensure VTTablet will not start unless there is a valid ACL configuration. This is used to catch misconfigurations resulting in blanket access to authenticated users.
- `-queryserver-config-enable-table-acl-dry-run`: Set to `true` to check the table ACL at runtime, and only emit the TableACLPseudoDenied metric if a request would have been blocked. The request is then allowed to pass, even if the ACL determined it should be blocked. This can be used for testing new or updated ACL policies. Default is `false`.
- `-queryserver-config-strict-table-acl`: Set to `true` to enforce table ACL checking. **This needs to be enabled for your ACLs to have any effect.** Any users that are not specified in an ACL policy will be denied. Default is `false`.
- `-queryserver-config-acl-exempt-acl`: Allows you to specify the name of an ACL (see below for format) that is exempt from enforcement. Allows you to separate the rollout and the subsequent enforcement of a specific ACL.
- `-table-acl-config`: Path to a file defining the table ACL config.
- `-table-acl-config-reload-interval`: How often the `table-acl-config` should be reloaded. Set this to allow you to update the ACL file on disk, and then have VTTablet automatically reload the file within this period. Default is not to reload the ACL file after VTTablet startup. Note that even if you do not set this parameter, you can always force VTTablet to reload the ACL config file from disk by sending a SIGHUP signal to your VTTablet process.

Format of the table ACL config file

The file specified in the `-table-acl-config` parameter above is a JSON file with the following example to explain the format:

```
{
    "table_groups": [
        {
            "name": "aclname",
            "table_names_or_prefixes": [
                "%"
            ],
            "readers": [
                "vtgate-user1"
            ],
            "writers": [
                "vtgate-user2"
            ],
            "admins": [
```

```
                "vtgate-user3"
            ]
        },
        { "... more ACLs here if necessary ..." }
    ]
}
```

Notes:

- `name`: This is the name of the ACL (`aclname` in the example above) and is what needs to be specified in `-queryserver-config-acl-exempt-acl`, if you need to exempt a specific ACL from enforcement.
- `table_names_or_prefixes`: A list of strings and/or regexes that allow a rule to target a specific table or set of tables. Use `%` as in the example to specify all tables. Note that only the SQL `%` "regex" wildcard is supported here at the moment.
- `readers`: A list of VTGate users, specified by their UserData field in the authentication specification, that are allowed to read the tables targeted by this ACL rule. Typically allows `SELECT`.
- `writers`: A list of VTGate users that are allowed to write to the tables targeted by this ACL rule. Typically allows `INSERT`, `UPDATE` and `DELETE`.
- `admins`: A list of VTGate users that are allowed admin privileges on the tables targeted by this ACL rule. Typically allows DDL privileges, e.g. `ALTER TABLE`. Note that this also includes some commands that might be thought of as DML, which are really DDL, like `TRUNCATE`)
- Note that `writers` privilege does not imply `readers` privilege, and `admins` privilege does not imply `readers` or `writers`. You need to therefore add your users to each list explicitly if you want them to have that level of access.
- You cannot use multiple ACL rules to target the same (sub)set of tables. Therefore the tablenames specified by `table_names_or_prefixes` (or expanded by regexes) need to be non-overlapping between ACL rules. Additionally, you cannot have duplicate tablenames or overlapping regexes in the `table_names_or_prefixes` list in a single ACL rule.

Example

Let's assume your Vitess cluster already has two keyspaces setup:

- `keyspace1` with a single table `t` that should only be accessed by `myuser1`
- `keyspace2` with a single table `t` that should only be accessed by `myuser2`

For the VTTablet configuration for `keyspace1`:

```
$ cat > acls_for_keyspace1.json << EOF
{
  "table_groups": [
    {
      "name": "keyspace1acls",
      "table_names_or_prefixes": ["%"],
      "readers": ["myuser1", "vitess"],
      "writers": ["myuser1", "vitess"],
      "admins": ["myuser1", "vitess"]
    }
  ]
}
EOF

$ vttablet -init_keyspace "keyspace1" -table-acl-config=acls_for_keyspace1.json
   -enforce-tableacl-config -queryserver-config-strict-table-acl ........
```

Note that the `%` specifier for `table_names_or_prefixes` translates to "all tables".

Do the same thing for `keyspace2`:

```
$ cat > acls_for_keyspace2.json << EOF
{
  "table_groups": [
    {
      "name": "keyspace2acls",
      "table_names_or_prefixes": ["%"],
      "readers": ["myuser2", "vitess"],
      "writers": ["myuser2", "vitess"],
      "admins": ["myuser2", "vitess"]
    }
  ]
}
EOF

$ vttablet -init_keyspace "keyspace2" -table-acl-config=acls_for_keyspace2.json
   -enforce-tableacl-config -queryserver-config-strict-table-acl ........
```

With this setup, the `myuser1` and `myuser2` users can only access their respective keyspaces, but the `vitess` user can access both.

```
# Attempt to access keyspace1 with myuser2 credentials through vtgate
$ mysql -h 127.0.0.1 -u myuser2 -ppassword2 -D keyspace1 -e "select * from t"
ERROR 1045 (HY000) at line 1: vtgate: http://vtgate-zone1-7fbfd8cc47-tchbz:15001/: target:
   keyspace1.-80.master, used tablet: zone1-476565201
   (zone1-keyspace1-x-80-replica-1.vttablet): vttablet: rpc error: code = PermissionDenied
   desc = table acl error: "myuser2" [] cannot run PASS_SELECT on table "t" (CallerID:
   myuser2)
target: keyspace1.80-.master, used tablet: zone1-1289569200
   (zone1-keyspace1-80-x-replica-0.vttablet): vttablet: rpc error: code = PermissionDenied
   desc = table acl error: "myuser2" [] cannot run PASS_SELECT on table "t" (CallerID:
   myuser2)
$
```

Whereas myuser1 is able to access its keyspace without error:

```
$ mysql -h 127.0.0.1 -u myuser1 -ppassword1 -D keyspace1 -e "select * from t"
$
```

CreateLookupVindex

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have an Operator or local installation ready. Make sure you are at the point where you have the sharded keyspace called `customer` setup. {{< /info >}}

CreateLookupVindex is a new VReplication workflow in Vitess 6. It is used to create **and** backfill a lookup Vindex automatically for a table that already exists, and may have a significant amount of data in it already.

Internally, the `CreateLookupVindex` process uses VReplication for the backfill process, until the lookup Vindex is "in sync". Then the normal process for adding/deleting/updating rows in the lookup Vindex via the usual transactional flow when updating the "owner" table for the Vindex takes over.

In this guide, we will walk through the process of using the `CreateLookupVindex` workflow, and give some insight into what happens underneath the covers.

`vtctlclient CreateLookupVindex` has the following syntax:

```
CreateLookupVindex [-cell=<cell>] [-tablet_types=<source_tablet_types>] <keyspace> <json_spec>
```

- `<json_spec>`: Use the lookup Vindex specified in `<json_spec>` along with VReplication to populate/backfill the lookup Vindex from the source table.
- `<keyspace>`: The Vitess keyspace we are creating the lookup Vindex in. The source table is expected to also be in this keyspace.
- `-tablet-types`: Provided to specify the shard tablet types (e.g. `MASTER`, `REPLICA`, `RDONLY`) that are acceptable as source tablets for the VReplication stream(s) that this command will create. If not specified, the tablet type used will default to the value of the vttablet `-vreplication_tablet_type` option, which defaults to `REPLICA`.
- `-cell`: By default VReplication streams, such as used by `CreateLookupVindex` will not cross cell boundaries. If you want the VReplication streams to source their data from tablets in a cell other than the local cell, you can use the `-cell` option to specify this.

The `<json_spec>` describes the lookup Vindex to be created, and details about the table it is to be created against (on which column, etc.). However, you do not have to specify details about the actual lookup table, Vitess will create that automatically based on the type of the column you are creating the Vindex column on, etc.

In the context of the regular `customer` database that is part of the Vitess examples we started earlier, let's add some rows into the `customer.corder` table, and then look at an example `<json_spec>`:

```
$ mysql -P 15306 -h 127.0.0.1 -u root --binary-as-hex=false -A
Welcome to the MySQL monitor.  Commands end with ; or \g.
.
.
.
mysql> use customer;
Database changed

mysql> show tables;
+------------------------+
| Tables_in_vt_customer |
+------------------------+
| corder                |
| customer              |
+------------------------+
2 rows in set (0.00 sec)

mysql> desc corder;
+-------------+----------------+------+-----+---------+-------+
| Field       | Type           | Null | Key | Default | Extra |
+-------------+----------------+------+-----+---------+-------+
| order_id    | bigint         | NO   | PRI | NULL    |       |
| customer_id | bigint         | YES  |     | NULL    |       |
| sku         | varbinary(128) | YES  |     | NULL    |       |
| price       | bigint         | YES  |     | NULL    |       |
+-------------+----------------+------+-----+---------+-------+
4 rows in set (0.01 sec)

mysql> insert into corder (order_id, customer_id, sku, price) values (1, 1, "Product_1",
   100);
Query OK, 1 row affected (0.01 sec)

mysql> insert into corder (order_id, customer_id, sku, price) values (2, 1, "Product_2",
   101);
Query OK, 1 row affected (0.01 sec)

mysql> insert into corder (order_id, customer_id, sku, price) values (3, 2, "Product_3",
   102);
```

```
Query OK, 1 row affected (0.01 sec)

mysql> insert into corder (order_id, customer_id, sku, price) values (4, 3, "Product_4",
   103);
Query OK, 1 row affected (0.01 sec)

mysql> insert into corder (order_id, customer_id, sku, price) values (5, 4, "Product_5",
   104);
Query OK, 1 row affected (0.03 sec)

mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        1 |           1 | Product_1 |   100 |
|        2 |           1 | Product_2 |   101 |
|        3 |           2 | Product_3 |   102 |
|        4 |           3 | Product_4 |   103 |
|        5 |           4 | Product_5 |   104 |
+----------+-------------+-----------+-------+
5 rows in set (0.01 sec)
```

If we look at the VSchema for the `customer.corder` table, we will see there is a `hash` index on the `customer_id` table, and 4 of our 5 rows have ended up on the `-80` shard, and the 5th row on the `80-` shard:

```
mysql> use customer/-80
Database changed

mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        1 |           1 | Product_1 |   100 |
|        2 |           1 | Product_2 |   101 |
|        3 |           2 | Product_3 |   102 |
|        4 |           3 | Product_4 |   103 |
+----------+-------------+-----------+-------+
4 rows in set (0.00 sec)

mysql> use customer/80-
Database changed

mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        5 |           4 | Product_5 |   104 |
+----------+-------------+-----------+-------+
1 row in set (0.01 sec)
```

Note that this skewed distribution is completely coincidental, for larger numbers of rows, we would expect the distribution to be approximately even for a `hash` index.

Now let's say we want to add a lookup Vindex on the `sku` column. We can use a `consistent_lookup` or `consistent_lookup_unique` Vindex type. In our example we will use `consistent_lookup_unique`.

Here is our example `<json_spec>`:

```
$ cat lookup_vindex.json
```

```
{
    "sharded": true,
    "vindexes": {
        "corder_lookup": {
            "type": "consistent_lookup_unique",
            "params": {
                "table": "customer.corder_lookup",
                "from": "sku",
                "to": "keyspace_id"
            },
            "owner": "corder"
        }
    },
    "tables": {
        "corder": {
            "column_vindexes": [
                {
                    "column": "sku",
                    "name": "corder_lookup"
                }
            ]
        }
    }
}
```

Note that as mentioned above, we do not have to tell Vitess about how to shard the actual backing table for the lookup Vindex or any schema to create as it will do it automatically. Now, let us actually execute the `CreateLookupVindex` command:

```
$ vtctlclient -server localhost:15999 CreateLookupVindex -tablet_types=RDONLY customer
   "$(cat lookup_vindex.json)"
```

Note:

- We are specifying a tablet_type of `RDONLY`; meaning it is going to run the VReplication streams from tablets of the `RDONLY` type **only**. If tablets of this type cannot be found, in a shard, the lookup Vindex population will fail.

Now, in our case, the table is tiny, so the copy will be instant, but in a real-world case this might take hours. To monitor the process, we can use the usual VReplication commands. However, the VReplication status commands needs to operate on individual tablets. Let's check which tablets we have in our environment, so we know which tablets to issue commands against:

```
$ vtctlclient -server localhost:15999 ListAllTablets | grep customer
zone1-0000000300 customer -80 master localhost:15300 localhost:17300 [] 2020-08-13T01:23:15Z
zone1-0000000301 customer -80 replica localhost:15301 localhost:17301 [] <null>
zone1-0000000302 customer -80 rdonly localhost:15302 localhost:17302 [] <null>
zone1-0000000400 customer 80- master localhost:15400 localhost:17400 [] 2020-08-13T01:23:15Z
zone1-0000000401 customer 80- replica localhost:15401 localhost:17401 [] <null>
zone1-0000000402 customer 80- rdonly localhost:15402 localhost:17402 [] <null>
```

i.e. now we can see what will happen:

- VReplication streams will be setup from the master tablets `zone1-0000000300` and `zone1-0000000400`; pulling data from the `RDONLY` source tablets `zone1-0000000302` and `zone1-0000000402`.
- Note that each master tablet will start streams from each source tablet, for a total of 4 streams in this case.

Lets observe the VReplication streams that got created using the `vtctlclient VReplicationExec` command. First let's look at the streams to the first master tablet `zone1-0000000300`:

```
$ vtctlclient -server localhost:15999 VReplicationExec zone1-0000000300 "select * from
   _vt.vreplication"
+----+-------------------+----------------------------------------+-------------------------------
| id |     workflow      |                source                  |
   pos                        | stop_pos |       max_tps       | max_replication_lag | cell
   | tablet_types | time_updated | transaction_timestamp |  state  |       message       |
     db_name   |
+----+-------------------+----------------------------------------+-------------------------------
|  2 | corder_lookup_vdx | keyspace:"customer" shard:"-80"        |
   MySQL56/68da1cdd-dd03-11ea-95de-68a86d2718b0:1-43 |          | 9223372036854775807 |
   9223372036854775807 |      | RDONLY       |   1597282811 |                     0 |
   Stopped | Stopped after copy. | vt_customer |
|    |                   | filter:<rules:<match:"corder_lookup" |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | filter:"select sku as sku,             |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | keyspace_id() as keyspace_id from      |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | corder where in_keyrange(sku,          |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | 'customer.binary_md5', '-80')          |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | group by sku, keyspace_id" > >         |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | stop_after_copy:true                   |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|  3 | corder_lookup_vdx | keyspace:"customer" shard:"80-"        |
   MySQL56/7d2c819e-dd03-11ea-92e4-68a86d2718b0:1-38 |          | 9223372036854775807 |
   9223372036854775807 |      | RDONLY       |   1597282811 |                     0 |
   Stopped | Stopped after copy. | vt_customer |
|    |                   | filter:<rules:<match:"corder_lookup" |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | filter:"select sku as sku,             |
                                                |          |                     |
                       |      |             |             |                       |
           |                     |             |
|    |                   | keyspace_id() as keyspace_id from      |
                                                |          |                     |
```

```
                        |      |              |              |                          |
          |                     |              |
|    |                    | corder where in_keyrange(sku,        |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | 'customer.binary_md5', '-80')        |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | group by sku, keyspace_id" > >       |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | stop_after_copy:true                 |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
+----+------------------+---------------------------------------+--------------------------------
```

And now the streams to the second master tablet zone1-0000000400:

```
$ vtctlclient -server localhost:15999 VReplicationExec zone1-0000000400 "select * from
   _vt.vreplication"
+----+------------------+---------------------------------------+--------------------------------
| id |     workflow     |                source                 |
   pos                        | stop_pos |       max_tps       | max_replication_lag | cell
   | tablet_types | time_updated | transaction_timestamp |  state  |       message       |
     db_name   |
+----+------------------+---------------------------------------+--------------------------------
|  2 | corder_lookup_vdx | keyspace:"customer" shard:"-80"      |
   MySQL56/68da1cdd-dd03-11ea-95de-68a86d2718b0:1-43 |          | 9223372036854775807 |
   9223372036854775807 |      | RDONLY       |   1597282811 |                     0 |
   Stopped | Stopped after copy. | vt_customer |
|    |                    | filter:<rules:<match:"corder_lookup" |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | filter:"select sku as sku,           |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | keyspace_id() as keyspace_id from    |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | corder where in_keyrange(sku,        |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | 'customer.binary_md5', '80-')        |
                                               |          |                        |
                        |      |              |              |                          |
          |                     |              |
|    |                    | group by sku, keyspace_id" > >       |
                                               |          |                        |
                        |      |              |              |                          |
```

```
           |                      |             |
|    |                  | stop_after_copy:true                |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|  3 | corder_lookup_vdx | keyspace:"customer" shard:"80-"      |
   MySQL56/7d2c819e-dd03-11ea-92e4-68a86d2718b0:1-38 |          | 9223372036854775807 |
   9223372036854775807 |      | RDONLY       |   1597282811 |                    0 |
   Stopped | Stopped after copy. | vt_customer |
|    |                  | filter:<rules:<match:"corder_lookup" |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | filter:"select sku as sku,           |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | keyspace_id() as keyspace_id from    |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | corder where in_keyrange(sku,        |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | 'customer.binary_md5', '80-')        |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | group by sku, keyspace_id" > >       |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
|    |                  | stop_after_copy:true                 |
                                                |          |                       |
                      |      |             |             |                        |
           |                      |             |
+----+------------------+--------------------------------------+-------------------------------
```

There is a lot going on in this output, but the most important parts are the `state` and `message` fields which say `Stopped` and `Stopped after copy.` for all four the streams. This means that the VReplication streams finished their copying/backfill of the lookup table.

Note that if the tables were large and the copy was still in progress, the `state` field would say `Copying`, and you can see the state/progress of the copy by looking at the `_vt.copy_state` table, e.g.:

```
$ vtctlclient -server localhost:15999 VReplicationExec zone1-0000000300 "select * from
   _vt.copy_state"
+----------+------------+--------+
| vrepl_id | table_name | lastpk |
+----------+------------+--------+
+----------+------------+--------+
```

(In this case this table is empty, because the copy has finished already).

We can verify the result of the backfill by looking at the `customer` keyspace again in the MySQL client:

```
mysql> show tables;
+-----------------------+
```

```
| Tables_in_vt_customer |
+-----------------------+
| corder                |
| corder_lookup         |
| customer              |
+-----------------------+
3 rows in set (0.01 sec)
```

Note there is now a new table, `corder_lookup`; which was created as the backing table for the lookup Vindex. Lets look at this table:

```
mysql> desc corder_lookup;
+-------------+----------------+------+-----+---------+-------+
| Field       | Type           | Null | Key | Default | Extra |
+-------------+----------------+------+-----+---------+-------+
| sku         | varbinary(128) | NO   | PRI | NULL    |       |
| keyspace_id | varbinary(128) | YES  |     | NULL    |       |
+-------------+----------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> select sku, hex(keyspace_id) from corder_lookup;
+-----------+------------------+
| sku       | hex(keyspace_id) |
+-----------+------------------+
| Product_2 | 166B40B44ABA4BD6 |
| Product_3 | 06E7EA22CE92708F |
| Product_1 | 166B40B44ABA4BD6 |
| Product_4 | 4EB190C9A2FA169C |
| Product_5 | D2FD8867D50D2DFE |
+-----------+------------------+
```

Basically, this shows exactly what we expected. Now, we have to clean-up the artifacts of the backfill. The `ExternalizeVindex` command will delete the vreplication streams and also clear the `write_only` flag from the vindex indicating that it is not backfilling any more.

```
$ vtctlclient -server localhost:15999 ExternalizeVindex customer.corder_lookup
```

Next, to confirm the lookup Vindex is doing what we think it should, we can use the Vitess MySQL explain format, e.g.:

```
mysql> explain format=vitess select * from corder where customer_id = 1;
+----------+-------------------+----------+-------------+------------+-----------------------------
| operator | variant           | keyspace | destination | tabletType | query
                                  |
+----------+-------------------+----------+-------------+------------+-----------------------------
| Route    | SelectEqualUnique | customer |             | UNKNOWN    | select * from corder
   where customer_id = 1 |
+----------+-------------------+----------+-------------+------------+-----------------------------
1 row in set (0.00 sec)
```

Since the above `select` statement is doing a lookup using the primary Vindex on the `corder` table, this query does not Scatter (variant is `SelectEqualUnique`), as expected. Let's try a scatter query to see what that looks like:

```
mysql> explain format=vitess select * from corder;
+----------+---------------+----------+-------------+------------+----------------------+
| operator | variant       | keyspace | destination | tabletType | query                |
+----------+---------------+----------+-------------+------------+----------------------+
| Route    | SelectScatter | customer |             | UNKNOWN    | select * from corder |
+----------+---------------+----------+-------------+------------+----------------------+
1 row in set (0.00 sec)
```

OK, variant is `SelectScatter` for a scatter query. Let's try a lookup on a column that does not have a primary or secondary (lookup) Vindex, e.g. the `price` column:

```
mysql> explain format=vitess select * from corder where price = 103;
+----------+---------------+----------+-------------+------------+--------------------------------
| operator | variant       | keyspace | destination | tabletType | query
                                    |
+----------+---------------+----------+-------------+------------+--------------------------------
| Route    | SelectScatter | customer |             | UNKNOWN    | select * from corder
   where price = 103 |
+----------+---------------+----------+-------------+------------+--------------------------------
1 row in set (0.00 sec)
```

That also scatters, as expected.

Now, let's try a lookup on the `sku` column, which we have created our lookup Vindex on:

```
mysql> explain format=vitess select * from corder where sku = "Product_1";
+----------+-------------------+----------+-------------+------------+----------------------------
| operator | variant           | keyspace | destination | tabletType | query
                                     |
+----------+-------------------+----------+-------------+------------+----------------------------
| Route    | SelectEqualUnique | customer |             | UNKNOWN    | select * from corder
   where sku = 'Product_1' |
+----------+-------------------+----------+-------------+------------+----------------------------
1 row in set (0.00 sec)
```

As expected, we can see it is not scattering anymore, which it would have before we did `CreateLookupVindex`.

Lastly, let's ensure that the lookup Vindex is being updated appropriately when we insert and delete rows:

```
mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        5 |           4 | Product_5 |   104 |
|        1 |           1 | Product_1 |   100 |
|        2 |           1 | Product_2 |   101 |
|        3 |           2 | Product_3 |   102 |
|        4 |           3 | Product_4 |   103 |
+----------+-------------+-----------+-------+
5 rows in set (0.00 sec)

mysql> delete from corder where customer_id = 1 and sku = "Product_1";
Query OK, 1 row affected (0.03 sec)

mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        2 |           1 | Product_2 |   101 |
|        3 |           2 | Product_3 |   102 |
|        4 |           3 | Product_4 |   103 |
|        5 |           4 | Product_5 |   104 |
+----------+-------------+-----------+-------+
4 rows in set (0.01 sec)

mysql> select sku, hex(keyspace_id) from corder_lookup;
+-----------+------------------+
```

```
| sku       | hex(keyspace_id) |
+-----------+------------------+
| Product_4 | 4EB190C9A2FA169C |
| Product_5 | D2FD8867D50D2DFE |
| Product_2 | 166B40B44ABA4BD6 |
| Product_3 | 06E7EA22CE92708F |
+-----------+------------------+
4 rows in set (0.01 sec)
```

We deleted a row from the `corder` table, and the matching lookup Vindex row is gone.

```
mysql> insert into corder (order_id, customer_id, sku, price) values (6, 1, "Product_6",
    105);
Query OK, 1 row affected (0.02 sec)

mysql> select * from corder;
+----------+-------------+-----------+-------+
| order_id | customer_id | sku       | price |
+----------+-------------+-----------+-------+
|        2 |           1 | Product_2 |   101 |
|        3 |           2 | Product_3 |   102 |
|        4 |           3 | Product_4 |   103 |
|        6 |           1 | Product_6 |   105 |
|        5 |           4 | Product_5 |   104 |
+----------+-------------+-----------+-------+
5 rows in set (0.00 sec)

mysql> select sku, hex(keyspace_id) from corder_lookup;
+-----------+------------------+
| sku       | hex(keyspace_id) |
+-----------+------------------+
| Product_4 | 4EB190C9A2FA169C |
| Product_5 | D2FD8867D50D2DFE |
| Product_6 | 166B40B44ABA4BD6 |
| Product_2 | 166B40B44ABA4BD6 |
| Product_3 | 06E7EA22CE92708F |
+-----------+------------------+
5 rows in set (0.00 sec)
```

We added a new row to the `corder` table, and now we have a new row in the lookup table.

ExternalizeVindex Once the backfill is done,

Integration with Orchestrator

Orchestrator is a tool for managing MySQL replication topologies, including automated failover. It can detect master failure and initiate a recovery in a matter of seconds.

For the most part, Vitess is agnostic to the actions of Orchestrator, which operates below Vitess at the MySQL level. That means you can pretty much set up Orchestrator in the normal way, with just a few additions as described below.

For the Kubernetes example, we provide a sample script to launch Orchestrator for you with these settings applied.

Orchestrator configuration

Orchestrator needs to know some things from the Vitess side, like the tablet aliases and whether semisync is enforced with async fallback disabled. We pass this information by telling Orchestrator to execute certain queries that return local metadata from a non-replicated table, as seen in our sample orchestrator.conf.json:

```
"DetectClusterAliasQuery": "SELECT value FROM _vt.local_metadata WHERE name='ClusterAlias'",
"DetectInstanceAliasQuery": "SELECT value FROM _vt.local_metadata WHERE name='Alias'",
"DetectPromotionRuleQuery": "SELECT value FROM _vt.local_metadata WHERE
   name='PromotionRule'",
"DetectSemiSyncEnforcedQuery": "SELECT @@global.rpl_semi_sync_master_wait_no_slave AND
   @@global.rpl_semi_sync_master_timeout > 1000000",
```

Vitess also needs to know the identity of the master for each shard. This is necessary in case of a failover.

It is important to ensure that orchestrator has access to `vtctlclient` so that orchestrator can trigger the change in topology via the `TabletExternallyReparented` command.

```
"PostMasterFailoverProcesses": [
"echo 'Recovered from {failureType} on {failureCluster}. Failed: {failedHost}:{failedPort};
   Promoted: {successorHost}:{successorPort}' >> /tmp/recovery.log",
"vtctlclient -server vtctld:15999 TabletExternallyReparented {successorAlias}"
  ],
```

VTTablet configuration

Normally, you need to seed Orchestrator by giving it the addresses of MySQL instances in each shard. If you have lots of shards, this could be tedious or error-prone.

Luckily, Vitess already knows everything about all the MySQL instances that comprise your cluster. So we provide a mechanism for tablets to self-register with the Orchestrator API, configured by the following vttablet parameters:

- `orc_api_url`: Address of Orchestrator's HTTP API (e.g. http://host:port/api/). Leave empty to disable Orchestrator integration.
- `orc_discover_interval`: How often (e.g. 60s) to ping Orchestrator's HTTP API endpoint to tell it we exist. 0 means never.

Not only does this relieve you from the initial seeding of addresses into Orchestrator, it also means new instances will be discovered immediately, and the topology will automatically repopulate even if Orchestrator's backing store is wiped out. Note that Orchestrator will forget stale instances after a configurable timeout.

LDAP authentication

Currently, Vitess supports two ways to authenticate to `vtgate` via the MySQL protocol:

- **Static** : You provide a static configuration file to `vtgate` with user names and plaintext passwords or `mysql_native_password` password hashes. This file can be reloaded without restarting `vtgate`. Further details can be found here.
- **LDAP** : You provide the necessary details of an upstream LDAP server, along with credentials and configuration, to query it. Using this information, the LDAP passwords for a user can then be used to authenticate the same user against `vtgate`. You can also integrate with LDAP groups to allow ACLs to be managed using information from the LDAP server.

In this guide, we will examine the capabilities of the `vtgate` LDAP integration and how to configure them.

Requirements

There are a few requirements that are necessary for the `vtgate` LDAP integration to work:

- The communication between `vtgate` and the LDAP server has to be encrypted.
- Encrypted communication to LDAP has to be via LDAP over TLS (STARTTLS) and not via LDAP over SSL (LDAPS). The latter is not a standardized protocol and is not supported by Vitess. Ensure that your LDAP server and the LDAP URI (hostname/port) that you provide supports STARTTLS.
- The application MySQL protocol connections to `vtgate` that use LDAP usernames/passwords need to use TLS. This is required because of the next point, but can be bypassed. We strongly **DO NOT** recommend doing this.
- The application needs to be able to, and configured to, pass its password authentication using the cleartext MySQL authentication protocol. This is why it is required that the MySQL connection to `vtgate` be encrypted first. This is required because LDAP servers do not standardize their password hashes and, as a result, a cleartext password is required by `vtgate` to bind (i.e. authenticate) against the LDAP server to verify the user's password. Note that some applications might not support passing cleartext MySQL passwords without alteration or configuration. An example is recent versions of the MySQL CLI client `mysql` need the additional `--enable-cleartext-plugin` option to allow the passing of cleartext passwords.

Configuration

To configure `vtgate` to integrate with LDAP you will have to perform various tasks:

- Generate/obtain TLS certificate(s) for the `vtgate` server(s), and configure `vtgate` to use them. Further details can be found here.
- Obtain or add the necessary LDAP user/groups for integration with `vtgate`. In general, you will need:
  - LDAP user entries for each of the MySQL users you want to use at the `vtgate` level. An example might be a readonly user, a readwrite user, and an admin/DBA user.
  - Ensure these users are part of one or more LDAP groups. This is not strictly required by Vitess, but is leveraged to obtain group membership that can then be used in Vitess (`vttablet`)ACLs. At the moment if you use an LDAP user that is not a member of an LDAP group, the MySQL client authentication to `vtgate` will fail, even if the password is correct.
- As mentioned above, you also need to have:
  - Your LDAP server setup for STARTTLS
  - Obtained the LDAP URI to connect to the LDAP server
  - The CA certificate, that your LDAP server TLS certificate is signed by, in PEM format
  - Make sure that you are accessing the LDAP server via a hostname or IP SAN that is defined in your LDAP server TLS certificate. If not, you will not be able to use your LDAP server as-is from `vtgate`.

Once you have your prerequisites above ready, you can now construct your JSON configuration file for `vtgate` using the command line parameter `-mysql_ldap_auth_config_file`. The content of this file is a JSON format object with string key/value members as follows:

```
{
    "LdapServer": "ldapserver.example.org:389",
    "LdapCert": "path/to/ldap-client-cert.pem",
    "LdapKey": "path/to/ldap-client-key.pem",
    "LdapCA": "path/to/ldap-server-ca.pem",
    "User": "cn=admin,dc=example,dc=org",
    "Password": "adminpassword!",
    "GroupQuery": "ou=groups,ou=people,dc=example,dc=org",
    "UserDnPattern": "uid=%s,ou=users,ou=people,dc=example,dc=org",
    "RefreshSeconds": 300
}
```

Not all these options are necessary in all configurations. Here are what each key/value option represents:

- **LdapServer** : Hostname/IP and port to access the LDAP server via using STARTTLS. Note that as mentioned above, this needs to match the server TLS certificate presented by the LDAP server. This is required.
- **LdapCert** : Path to the local file that contains the PEM format TLS client certificate that you want to present to the LDAP server. This is optional unless you use client-certificates with the LDAP server. If you are using this option, `LdapKey` is also required.
- **LdapKey** : Path to the local file that contains the PEM format TLS private key for the client certificate you want to present to the LDAP server. This is optional unless you use client-certificates with the LDAP server. If you are using this option, `LdapCert` is also required.
- **LdapCA** : Path to the local file that contains the PEM format TLS CA certificate to verify against the TLS server certificate presented by the LDAP server. This is required.
- **User** : DN of the LDAP user you will be authenticating to the LDAP server to read information such as group membership. Required, unless you are using LDAP client certificates to authenticate to the LDAP server. If you are using this option, `Password` option is also required.
- **Password** : Cleartext password for the LDAP user specified above in `User`. This is required, unless you are using LDAP client certificates to authenticate to the LDAP server. If you are using this option, `User` option is also required.
- **GroupQuery** : LDAP base DN from which to start the group membership query to establish the group of which the `User` specified (or implied via the client certificate) is a member. The group membership query itself is hardcoded to the LDAP query filter of `(memberUid=%s)` where `%s` is the authenticating username. This is required.
- **UserDnPattern** : LDAP DN pattern to autofill with MySQL username passed during MySQL client authentication to `vtgate`. This DN is then used, along with the password provided to `vtgate`, to attempt to bind with the LDAP server. If the bind is successful, you know that the password provided to `vtgate` was valid. This is required.
- **RefreshSeconds** : Number of seconds that you should cache individual LDAP credentials for in-memory at the `vtgate`. This is used to reduce load on the LDAP for high traffic MySQL servers. As well as to avoid short LDAP server outages from causing Vitess/`vtgate` authentication outages. Default value is 0, which means **do not cache**. For production it is recommended to set this value to something reasonably high, for example at least a few minutes. This is optional.

Note that `vtgate` only does very basic validation of the values passed here and that incorrect configurations may just fail at runtime. If you are lucky, relevant errors may be logged by `vtgate`, but in many cases incorrect configuration will just result in a `vtgate` instance that you cannot log into via the MySQL protocol.

For debugging this, it is useful to have access to the logs from your LDAP server that you are pointing to. The logs would preferably be at trace or debug level, so that you can see each LDAP bind and search operation against the LDAP server as you are testing.

Once you have constructed the above file, you will need to remove any options that references static authentication from your `vtgate` command line such as:

- `-mysql_auth_server_static_file`
- `-mysql_auth_server_static_string`
- `-mysql_auth_static_reload_interval`
- `-mysql_auth_server_impl static`

and add the following new options:

```
-mysql_auth_server_impl ldap -mysql_ldap_auth_config_file /path/to/ldapconfig.json
```

Query Consolidation

Query consolidation is a VTTablet feature meant to protect your database from an overload caused by a spike in QPS for a specific query.

Without this feature enabled such spikes can completely overwhelm the database. With this feature enabled the following will occur: when a vttablet receives a query, if an identical query is already in the process of being executed, the query will then wait. As soon as the first query returns from the underlying database, the result is sent to all callers that have been waiting.

Note: an identical query is one that is exactly the same, including literals and bind variables.

Flags:

- `-enable_consolidator`: Defaults to true.
- `-enable_consolidator_replicas`: Only enable query consolidation on non-primary tablets.

Consistency

It is important to note that in some cases read-after-write consistency can be lost.

For example, if user1 issues a read query and user2 issues a write, that changes the result that the first read query would get, then user2 issues an identical read while user1's read is still executing.

In this case the consolidator will kick in and user2 will get the result of user1's query thereby losing read-after-write consistency.

If the application is sensitive to this behavior then you can specify that consolidation should be disabled on the primary using the following flags: `-enable_consolidator=false` and `-enable_consolidator_replicas=true`

Region-based Sharding

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have a local installation ready. You should also have already gone through the MoveTables and Resharding tutorials. {{< /info >}}

Preparation

Having gone through the Resharding tutorial, you should be familiar with VSchema and Vindexes. In this tutorial, we will perform resharding on an existing keyspace using a location-based vindex. We will create 4 shards (-40, 40-80, 80-c0, c0-). The location will be denoted by a `country` column.

Schema

We will create one table in the unsharded keyspace to start with.

```
CREATE TABLE customer (
  id int NOT NULL,
  fullname varbinary(256),
  nationalid varbinary(256),
  country varbinary(256),
  primary key(id)
  );
```

The customer table is the main table we want to shard using country.

Region Vindex

We will use a `region_json` vindex to compute the keyspace_id for a customer row using the (id, country) fields. Here's what the vindex definition looks like:

```
    "region_vdx": {
        "type": "region_json",
        "params": {
            "region_map": "/home/user/my-vitess/examples/region_sharding/countries.json",
            "region_bytes": "1"
        }
    },
```

And we use it thus:

```
    "customer": {
      "column_vindexes": [
        {
            "columns": ["id", "country"],
            "name": "region_vdx"
        },
```

This vindex uses a byte mapping of countries provided in a JSON file and combines that with the id column in the customer table to compute the keyspace_id. This is what the JSON file contains:

```
{
    "United States": 1,
    "Canada": 2,
    "France": 64,
    "Germany": 65,
    "China": 128,
    "Japan": 129,
    "India": 192,
    "Indonesia": 193
}
```

The values for the countries have been chosen such that 2 countries fall into each shard.

In this example, we are using 1 byte to represent a country code. You can use 1 or 2 bytes. With 2 bytes, 65536 distinct locations can be supported. The byte value of the country(or other location identifier) is prefixed to a hash value computed from the id to produce the keyspace_id. This will be primary vindex on the `customer` table. As such, it is sufficient for resharding, inserts and selects. However, we don't yet support updates and deletes using a multi-column vindex. In order for those to work, we need to create a lookup vindex that can used to find the correct rows by id. The lookup vindex also makes querying by id efficient. Without it, queries that provided id but not country will scatter to all shards.

To do this, we will use the new vreplication workflow `CreateLookupVindex`. This workflow will create the lookup table and a lookup vindex. It will also associate the lookup vindex with the `customer` table.

Start the Cluster

Start by copying the region_sharding example included with Vitess to your preferred location.

```
cp -r /usr/local/vitess/examples/region_sharding ~/my-vitess/examples/region_sharding
cd ~/my-vitess/examples/region_sharding
```

The VSchema for this tutorial uses a config file. You will need to edit the value of the `region_map` parameter in the vschema file `main_vschema_sharded.json`. For example:

```
"region_map": "/home/user/my-vitess/examples/region_sharding/countries.json",
```

Now start the cluster

```
./101_initial_cluster.sh
```

You should see output similar to the following:

```
~/my-vitess-example> ./101_initial_cluster.sh
add /vitess/global
add /vitess/zone1
add zone1 CellInfo
etcd start done...
Starting vtctld...
Starting MySQL for tablet zone1-0000000100...
Starting vttablet for zone1-0000000100...
HTTP/1.1 200 OK
```

```
Date: Mon, 17 Aug 2020 14:20:08 GMT
Content-Type: text/html; charset=utf-8

W0817 07:20:08.822742    7735 main.go:64] W0817 14:20:08.821985 reparent.go:185]
   master-elect tablet zone1-0000000100 is not the shard master, proceeding anyway as
   -force was used
W0817 07:20:08.823004    7735 main.go:64] W0817 14:20:08.822370 reparent.go:191]
   master-elect tablet zone1-0000000100 is not a master in the shard, proceeding anyway as
   -force was used
I0817 07:20:08.823239    7735 main.go:64] I0817 14:20:08.823075 reparent.go:222] resetting
   replication on tablet zone1-0000000100
I0817 07:20:08.833215    7735 main.go:64] I0817 14:20:08.833019 reparent.go:241]
   initializing master on zone1-0000000100
I0817 07:20:08.849955    7735 main.go:64] I0817 14:20:08.849736 reparent.go:274] populating
   reparent journal on new master zone1-0000000100
New VSchema object:
{
  "tables": {
    "customer": {

    }
  }
}
If this is not what you expected, check the input data (as JSON parsing will skip
   unexpected fields).
Waiting for vtgate to be up...
vtgate is up!
Access vtgate at http://localhost:15001/debug/status
```

You can also verify that the processes have started with `pgrep`:

```
~/my-vitess-example> pgrep -fl vtdataroot
9160 etcd
9222 vtctld
9280 mysqld_safe
9843 mysqld
9905 vttablet
10040 vtgate
10224 mysqld
```

The exact list of processes will vary. For example, you may not see `mysqld_safe` listed.

If you encounter any errors, such as ports already in use, you can kill the processes and start over:

```
pkill -9 -e -f '(vtdataroot|VTDATAROOT)' # kill Vitess processes
rm -rf vtdataroot
```

Aliases

For ease-of-use, Vitess provides aliases for `mysql` and `vtctlclient`. These are automatically created when you start the cluster.

```
source ./env.sh
```

Setting up aliases changes `mysql` to always connect to Vitess for your current session. To revert this, type `unalias mysql && unalias vtctlclient` or close your session.

Connect to your cluster

You should now be able to connect to the VTGate server that was started in `101_initial_cluster.sh`:

```
~/my-vitess-example> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.9-Vitess (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+-------------------+
| Tables_in_vt_main |
+-------------------+
| customer          |
+-------------------+
1 row in set (0.01 sec)
```

Insert some data into the cluster

```
~/my-vitess-example> mysql < insert_customers.sql
```

Examine the data we just inserted

```
~/my-vitess-example> mysql --table < show_initial_data.sql
```

```
+----+------------------+-------------+---------------+
| id | fullname         | nationalid  | country       |
+----+------------------+-------------+---------------+
|  1 | Philip Roth      | 123-456-789 | United States |
|  2 | Gary Shteyngart  | 234-567-891 | United States |
|  3 | Margaret Atwood  | 345-678-912 | Canada        |
|  4 | Alice Munro      | 456-789-123 | Canada        |
|  5 | Albert Camus     | 912-345-678 | France        |
|  6 | Colette          | 102-345-678 | France        |
|  7 | Hermann Hesse    | 304-567-891 | Germany       |
|  8 | Cornelia Funke   | 203-456-789 | Germany       |
|  9 | Cixin Liu        | 789-123-456 | China         |
| 10 | Jian Ma          | 891-234-567 | China         |
| 11 | Haruki Murakami  | 405-678-912 | Japan         |
| 12 | Banana Yoshimoto | 506-789-123 | Japan         |
| 13 | Arundhati Roy    | 567-891-234 | India         |
| 14 | Shashi Tharoor   | 678-912-345 | India         |
| 15 | Andrea Hirata    | 607-891-234 | Indonesia     |
| 16 | Ayu Utami        | 708-912-345 | Indonesia     |
+----+------------------+-------------+---------------+
```

Prepare for resharding

Now that we have some data in our unsharded cluster, let us go ahead and perform the setup needed for resharding. The initial vschema is unsharded and simply lists the customer table (see script output above). We are going to first apply the sharding vschema to the cluster from `main_vschema_sharded.json`

```
{
  "sharded": true,
  "vindexes": {
    "region_vdx": {
      "type": "region_json",
      "params": {
        "region_map": "/home/user/my-vitess/examples/region_sharding/countries.json",
        "region_bytes": "1"
      }
    }
  },
  "tables": {
    "customer": {
      "column_vindexes": [
        {
          "columns": ["id", "country"],
              "name": "region_vdx"
        }
      ]
    }
  }
}
```

Then we will create a lookup vindex (`CreateLookupVindex`) using the definition in `lookup_vindex.json`

Here is the lookup vindex definition. Here we both define the lookup vindex, and associate it with the customer table.

```
{
  "sharded": true,
  "vindexes": {
    "customer_region_lookup": {
      "type": "consistent_lookup_unique",
      "params": {
        "table": "main.customer_lookup",
        "from": "id",
        "to": "keyspace_id"
      },
      "owner": "customer"
    }
  },
  "tables": {
    "customer": {
      "column_vindexes": [
        {
          "column": "id",
          "name": "customer_region_lookup"
        }
      ]
    }
  }
}
```

Once the vindex is available, we have to `Externalize` it for it to be usable. Putting this all together, we run the script that combines the above steps.

```
./201_main_sharded.sh
```

Once this is complete, we can view the new vschema. Note that it now includes both region_vdx and a lookup vindex.

```
~/my-vitess-example> vtctlclient GetVSchema main
{
  "sharded": true,
  "vindexes": {
    "customer_region_lookup": {
      "type": "consistent_lookup_unique",
      "params": {
        "from": "id",
        "table": "main.customer_lookup",
        "to": "keyspace_id"
      },
      "owner": "customer"
    },
    "hash": {
      "type": "hash"
    },
    "region_vdx": {
      "type": "region_json",
      "params": {
        "region_bytes": "1",
        "region_map": "/home/user/my-vitess/examples/region_sharding/countries.json"
      }
    }
  },
  "tables": {
    "customer": {
      "columnVindexes": [
        {
          "name": "region_vdx",
          "columns": [
            "id",
            "country"
          ]
        },
        {
          "column": "id",
          "name": "customer_region_lookup"
        }
      ]
    },
    "customer_lookup": {
      "columnVindexes": [
        {
          "column": "id",
          "name": "hash"
        }
      ]
    }
  }
}
```

Notice that the vschema shows a hash vindex on the lookup table. This is automatically created by the workflow. Creating a lookup vindex via `CreateLookupVindex` also creates the backing table needed to hold the vindex, and populates it with the correct rows. We can see that by checking the database.

```
mysql> show tables;
+--------------------+
| Tables_in_vt_main |
+--------------------+
| customer          |
| customer_lookup   |
+--------------------+
2 rows in set (0.00 sec)

mysql> describe customer_lookup;
+-------------+----------------+------+-----+---------+-------+
| Field       | Type           | Null | Key | Default | Extra |
+-------------+----------------+------+-----+---------+-------+
| id          | int(11)        | NO   | PRI | NULL    |       |
| keyspace_id | varbinary(128) | YES  |     | NULL    |       |
+-------------+----------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> select id, hex(keyspace_id) from customer_lookup;
+----+--------------------+
| id | hex(keyspace_id)   |
+----+--------------------+
|  1 | 01166B40B44ABA4BD6 |
|  2 | 0106E7EA22CE92708F |
|  3 | 024EB190C9A2FA169C |
|  4 | 02D2FD8867D50D2DFE |
|  5 | 4070BB023C810CA87A |
|  6 | 40F098480AC4C4BE71 |
|  7 | 41FB8BAAAD918119B8 |
|  8 | 41CC083F1E6D9E85F6 |
|  9 | 80692BB9BF752B0F58 |
| 10 | 80594764E1A2B2D98E |
| 11 | 81AEFC44491CFE474C |
| 12 | 81D3748269B7058A0E |
| 13 | C062DCE203C602F358 |
| 14 | C0ACBFDA0D70613FC4 |
| 15 | C16A8B56ED414942B8 |
| 16 | C15B711BC4CEEBF2EE |
+----+--------------------+
16 rows in set (0.01 sec)
```

Once the sharding vschema and lookup vindex (+table) are ready, we can bring up the sharded cluster. Since we have 4 shards, we will bring up 4 sets of vttablets, 1 per shard. In this example, we are deploying only 1 tablet per shard and disabling semi-sync, but in general each shard will consist of at least 3 tablets.

```
./202_new_tablets.sh
```

```
Starting MySQL for tablet zone1-0000000200...
Starting vttablet for zone1-0000000200...
HTTP/1.1 200 OK
Date: Mon, 17 Aug 2020 15:07:41 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000300...
```

```
Starting vttablet for zone1-0000000300...
HTTP/1.1 200 OK
Date: Mon, 17 Aug 2020 15:07:46 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000400...
Starting vttablet for zone1-0000000400...
HTTP/1.1 200 OK
Date: Mon, 17 Aug 2020 15:07:50 GMT
Content-Type: text/html; charset=utf-8

Starting MySQL for tablet zone1-0000000500...
Starting vttablet for zone1-0000000500...
HTTP/1.1 200 OK
Date: Mon, 17 Aug 2020 15:07:55 GMT
Content-Type: text/html; charset=utf-8

W0817 08:07:55.217317   15230 main.go:64] W0817 15:07:55.215654 reparent.go:185]
   master-elect tablet zone1-0000000200 is not the shard master, proceeding anyway as
   -force was used
W0817 08:07:55.218083   15230 main.go:64] W0817 15:07:55.215771 reparent.go:191]
   master-elect tablet zone1-0000000200 is not a master in the shard, proceeding anyway as
   -force was used
I0817 08:07:55.218121   15230 main.go:64] I0817 15:07:55.215918 reparent.go:222] resetting
   replication on tablet zone1-0000000200
I0817 08:07:55.229794   15230 main.go:64] I0817 15:07:55.229416 reparent.go:241]
   initializing master on zone1-0000000200
I0817 08:07:55.249680   15230 main.go:64] I0817 15:07:55.249325 reparent.go:274] populating
   reparent journal on new master zone1-0000000200
W0817 08:07:55.286894   15247 main.go:64] W0817 15:07:55.286288 reparent.go:185]
   master-elect tablet zone1-0000000300 is not the shard master, proceeding anyway as
   -force was used
W0817 08:07:55.287392   15247 main.go:64] W0817 15:07:55.286354 reparent.go:191]
   master-elect tablet zone1-0000000300 is not a master in the shard, proceeding anyway as
   -force was used
I0817 08:07:55.287411   15247 main.go:64] I0817 15:07:55.286448 reparent.go:222] resetting
   replication on tablet zone1-0000000300
I0817 08:07:55.300499   15247 main.go:64] I0817 15:07:55.300276 reparent.go:241]
   initializing master on zone1-0000000300
I0817 08:07:55.324774   15247 main.go:64] I0817 15:07:55.324454 reparent.go:274] populating
   reparent journal on new master zone1-0000000300
W0817 08:07:55.363497   15264 main.go:64] W0817 15:07:55.362451 reparent.go:185]
   master-elect tablet zone1-0000000400 is not the shard master, proceeding anyway as
   -force was used
W0817 08:07:55.364061   15264 main.go:64] W0817 15:07:55.362569 reparent.go:191]
   master-elect tablet zone1-0000000400 is not a master in the shard, proceeding anyway as
   -force was used
I0817 08:07:55.364079   15264 main.go:64] I0817 15:07:55.362689 reparent.go:222] resetting
   replication on tablet zone1-0000000400
I0817 08:07:55.378370   15264 main.go:64] I0817 15:07:55.378201 reparent.go:241]
   initializing master on zone1-0000000400
I0817 08:07:55.401258   15264 main.go:64] I0817 15:07:55.400569 reparent.go:274] populating
   reparent journal on new master zone1-0000000400
W0817 08:07:55.437158   15280 main.go:64] W0817 15:07:55.435986 reparent.go:185]
   master-elect tablet zone1-0000000500 is not the shard master, proceeding anyway as
   -force was used
```

```
W0817 08:07:55.437953   15280 main.go:64] W0817 15:07:55.436038 reparent.go:191]
   master-elect tablet zone1-0000000500 is not a master in the shard, proceeding anyway as
   -force was used
I0817 08:07:55.437982   15280 main.go:64] I0817 15:07:55.436107 reparent.go:222] resetting
   replication on tablet zone1-0000000500
I0817 08:07:55.449958   15280 main.go:64] I0817 15:07:55.449725 reparent.go:241]
   initializing master on zone1-0000000500
I0817 08:07:55.467790   15280 main.go:64] I0817 15:07:55.466993 reparent.go:274] populating
   reparent journal on new master zone1-0000000500
```

Perform Resharding

Once the tablets are up, we can go ahead with the resharding.

```
./203_reshard.sh
```

This script has only one command: `Reshard`

```
vtctlclient Reshard -tablet_types=MASTER main.main2regions '0' '-40,40-80,80-c0,c0-'
```

Let us unpack this a bit. Since we are running only master tablets in this cluster, we have to tell the `Reshard` command to use them as the source for copying data into the target shards. The next argument is of the form `keyspace.workflow`. `keyspace` is the one we want to reshard. `workflow` is an identifier chosen by the user. It can be any arbitrary string and is used to tie the different steps of the resharding flow together. We will see it being used in subsequent steps. Then we have the source shard `0` and target shards `-40,40-80,80-c0,c0-`

This step copies all the data from source to target and sets up vreplication to keep the targets in sync with the source

We can check the correctness of the copy using `VDiff` and the `keyspace.workflow` we used for `Reshard`

```
vtctlclient VDiff main.main2regions
I0817 08:22:53.958578   16065 main.go:64] I0817 15:22:53.956743 traffic_switcher.go:389]
   Migration ID for workflow main2regions: 7369191857547657706
Summary for customer: {ProcessedRows:16 MatchingRows:16 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
```

Let's take a look at the vreplication streams

```
vtctlclient VReplicationExec zone1-0000000200 'select * from _vt.vreplication'
+----+--------------+---------------------------------+-----------------------------------------
| id |   workflow   |             source              |                          pos
                          | stop_pos |       max_tps       | max_replication_lag | cell |
   tablet_types | time_updated | transaction_timestamp |  state  | message | db_name |
+----+--------------+---------------------------------+-----------------------------------------
|  1 | main2regions | keyspace:"main" shard:"0"       |
   MySQL56/cd3b495a-e096-11ea-9088-34e12d1e6711:1-44 |          | 9223372036854775807 |
   9223372036854775807 |      | MASTER       |   1597676983 |                     0 |
   Running |         | vt_main |
|    |              | filter:<rules:<match:"/.*"      |
                                                      |          |                     |
                        |      |              |              |                          |
           |         |         |
|    |              | filter:"-40" > >                |
                                                      |          |                     |
                        |      |              |              |                          |
           |         |         |
+----+--------------+---------------------------------+-----------------------------------------
```

We have a running stream on tablet 200 (shard `-40`) that will keep it up-to-date with the source shard (`0`)

Cutover

Once the copy process is complete, we can start cutting-over traffic. This is done in 2 steps, `SwitchReads` and `SwitchWrites`. Note that the commands are named for the tablet_types and not user operations. `Reads` is used for replica/rdonly, and `Writes` for master. Read operations on master will not be affected by a `SwitchReads`.

```
./204_switch_reads.sh
./205_switch_writes.sh
```

Let us take a look at the sharded data

```
mysql> use main/-40;
Database changed

mysql> select * from customer;
+----+-----------------+-------------+---------------+
| id | fullname        | nationalid  | country       |
+----+-----------------+-------------+---------------+
|  1 | Philip Roth     | 123-456-789 | United States |
|  2 | Gary Shteyngart | 234-567-891 | United States |
|  3 | Margaret Atwood | 345-678-912 | Canada        |
|  4 | Alice Munro     | 456-789-123 | Canada        |
+----+-----------------+-------------+---------------+
4 rows in set (0.01 sec)

mysql> select id,hex(keyspace_id) from customer_lookup;
+----+--------------------+
| id | hex(keyspace_id)   |
+----+--------------------+
|  1 | 01166B40B44ABA4BD6 |
|  2 | 0106E7EA22CE92708F |
+----+--------------------+
2 rows in set (0.00 sec)
```

You can see that only data from US and Canada exists in the customer table in this shard. Repeat this for the other shards (40-80, 80-c0 and c0-) and see that each shard contains 4 rows in customer table.

The lookup table, however, has a different number of rows. This is because we are using a `hash` vindex to shard the lookup table which means that it is distributed differently from the customer table. If we look at the next shard 40-80:

```
mysql> use main/40-80;

Database changed
mysql> select id, hex(keyspace_id) from customer_lookup;
+----+--------------------+
| id | hex(keyspace_id)   |
+----+--------------------+
|  3 | 024EB190C9A2FA169C |
|  5 | 4070BB023C810CA87A |
|  9 | 80692BB9BF752B0F58 |
| 10 | 80594764E1A2B2D98E |
| 13 | C062DCE203C602F358 |
| 15 | C16A8B56ED414942B8 |
| 16 | C15B711BC4CEEBF2EE |
+----+--------------------+
7 rows in set (0.00 sec)
```

Drop source

Once resharding is complete, we can teardown the source shard

```
./206_down_shard_0.sh
./207_delete_shard_0.sh
```

What we have now is a sharded keyspace. The original unsharded keyspace no longer exists.

Teardown

Once you are done playing with the example, you can tear it down completely.

```
./301_teardown.sh
rm -rf vtdataroot
```

Reparenting

Reparenting is the process of changing a shard's master tablet from one host to another or changing a replica tablet to have a different master. Reparenting can be initiated manually or it can occur automatically in response to particular database conditions. As examples, you might reparent a shard or tablet during a maintenance exercise or automatically trigger reparenting when a master tablet dies.

This document explains the types of reparenting that Vitess supports:

- Active reparenting occurs when Vitess manages the entire reparenting process.
- External reparenting occurs when another tool handles the reparenting process, and Vitess just updates its topology service, replication graph, and serving graph to accurately reflect master-replica relationships.

Note: The `InitShardMaster` command defines the initial parenting relationships within a shard. That command makes the specified tablet the master and makes the other tablets in the shard replicas that replicate from that master.

MySQL requirements

GTIDs Vitess requires the use of global transaction identifiers (GTIDs) for its operations:

- During active reparenting, Vitess uses GTIDs to initialize the replication process and then depends on the GTID stream to be correct when reparenting. (During external reparenting, Vitess assumes the external tool manages the replication process.)
- During resharding, Vitess uses GTIDs for VReplication, the process by which source tablet data is transferred to the proper destination tablets.

Semisynchronous replication Vitess does not depend on semisynchronous replication but does work if it is implemented. Larger Vitess deployments typically do implement semisynchronous replication.

Active Reparenting You can use the following `vtctl` commands to perform reparenting operations:

- `PlannedReparentShard`
- `EmergencyReparentShard`

Both commands lock the Shard record in the global topology service. The two commands cannot run in parallel, nor can either command run in parallel with the `InitShardMaster` command.

Both commands are both dependent on the global topology service being available, and they both insert rows in the topology service's `_vt.reparent_journal` table. As such, you can review your database's reparenting history by inspecting that table.

PlannedReparentShard: Planned reparenting The `PlannedReparentShard` command reparents a healthy master tablet to a new master. The current and new master must both be up and running.

This command performs the following actions:

1. Puts the current master tablet in read-only mode.
2. Shuts down the current master's query service, which is the part of the system that handles user SQL queries. At this point, Vitess does not handle any user SQL queries until the new master is configured and can be used a few seconds later.
3. Retrieves the current master's replication position.
4. Instructs the master-elect tablet to wait for replication data and then begin functioning as the new master after that data is fully transferred.
5. Ensures replication is functioning properly via the following steps:
    - On the master-elect tablet, insert an entry in a test table and then update the global Shard object's MasterAlias record.
    - In parallel on each replica, including the old master, set the new master and wait for the test entry to replicate to the replica tablet. Replica tablets that had not been replicating before the command was called are left in their current state and do not start replication after the reparenting process.
    - Start replication on the old master tablet so it catches up to the new master.

In this scenario, the old master's tablet type transitions to `spare`. If health checking is enabled on the old master, it will likely rejoin the cluster as a replica on the next health check. To enable health checking, set the `target_tablet_type` parameter when starting a tablet. That parameter indicates what type of tablet that tablet tries to be when healthy. When it is not healthy, the tablet type changes to spare.

EmergencyReparentShard: Emergency reparenting The `EmergencyReparentShard` command is used to force a reparent to a new master when the current master is unavailable. The command assumes that data cannot be retrieved from the current master because it is dead or not working properly.

As such, this command does not rely on the current master at all to replicate data to the new master. Instead, it makes sure that the master-elect is the most advanced in replication within all of the available replicas.

Important: Before calling this command, you must first identify the replica with the most advanced replication position as that replica must be designated as the new master. You can use the `vtctl ShardReplicationPositions` command to determine the current replication positions of a shard's replicas.

This command performs the following actions:

1. Determines the current replication position on all of the replica tablets and confirms that the master-elect tablet has the most advanced replication position.
2. Promotes the master-elect tablet to be the new master. In addition to changing its tablet type to master, the master-elect performs any other changes that might be required for its new state.
3. Ensures replication is functioning properly via the following steps:
    - On the master-elect tablet, Vitess inserts an entry in a test table and then updates the `MasterAlias` record of the global Shard object.
    - In parallel on each replica, excluding the old master, Vitess sets the master and waits for the test entry to replicate to the replica tablet. Replica tablets that had not been replicating before the command was called are left in their current state and do not start replication after the reparenting process.

External Reparenting

External reparenting occurs when another tool handles the process of changing a shard's master tablet. After that occurs, the tool needs to call the `vtctl TabletExternallyReparented` command to ensure that the topology service, replication graph, and serving graph are updated accordingly.

That command performs the following operations:

1. Reads the Tablet from the local topology service.
2. Reads the Shard object from the global topology service.
3. If the Tablet type is not already `MASTER`, sets the tablet type to `MASTER`.
4. The Shard record is updated asynchronously (if needed) with the current master alias.
5. Any other tablets that still have their tablet type to `MASTER` will demote themselves to `REPLICA`.

The `TabletExternallyReparented` command fails in the following cases:

- The global topology service is not available for locking and modification. In that case, the operation fails completely.

Active reparenting might be a dangerous practice in any system that depends on external reparents. You can disable active reparents by starting `vtctld` with the `--disable_active_reparents` flag set to true. (You cannot set the flag after `vtctld` is started.)

Fixing Replication

A tablet can be orphaned after a reparenting if it is unavailable when the reparent operation is running but then recovers later on. In that case, you can manually reset the tablet's master to the current shard master using the `vtctl ReparentTablet` command. You can then restart replication on the tablet if it was stopped by calling the `vtctl StartReplication` command.

Resharding

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have an Operator, local or Helm installation ready. It also assumes that the MoveTables user guide has been followed. {{< /info >}}

Preparation

Resharding enables you to both *initially shard* and reshard tables so that your keyspace is partitioned across several underlying tablets. A sharded keyspace has some additional restrictions on both query syntax and features such as `auto_increment`, so it is helpful to plan out a reshard operation diligently. However, you can always *reshard again* if your sharding scheme turns out to be suboptimal.

Using our example commerce and customer keyspaces, lets work through the two most common issues.

Sequences The first issue to address is the fact that customer and corder have auto-increment columns. This scheme does not work well in a sharded setup. Instead, Vitess provides an equivalent feature through sequences.

The sequence table is an unsharded single row table that Vitess can use to generate monotonically increasing IDs. The syntax to generate an id is: `select next :n values from customer_seq`. The vttablet that exposes this table is capable of serving a very large number of such IDs because values are cached and served out of memory. The cache value is configurable.

The VSchema allows you to associate a column of a table with the sequence table. Once this is done, an insert on that table transparently fetches an id from the sequence table, fills in the value, and routes the row to the appropriate shard. This makes the construct backward compatible to how MySQL's `auto_increment` property works.

Since sequences are unsharded tables, they will be stored in the commerce database. Here is the schema:

```
CREATE TABLE customer_seq (id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
INSERT INTO customer_seq (id, next_id, cache) VALUES (0, 1000, 100);
CREATE TABLE order_seq (id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
INSERT INTO order_seq (id, next_id, cache) VALUES (0, 1000, 100);
```

Note the `vitess_sequence` comment in the create table statement. VTTablet will use this metadata to treat this table as a sequence.

- `id` is always 0
- `next_id` is set to `1000`: the value should be comfortably greater than the `auto_increment` max value used so far.
- `cache` specifies the number of values to cache before vttablet updates `next_id`.

Larger cache values perform better, but will exhaust the values quicker, since during reparent operations the new master will start off at the `next_id` value.

The VTGate servers also need to know about the sequence tables. This is done by updating the VSchema for commerce as follows:

```
{
  "tables": {
    "customer_seq": {
      "type": "sequence"
    },
    "order_seq": {
      "type": "sequence"
    },
    "product": {}
  }
}
```

Vindexes The next decision is about the sharding keys, or Primary Vindexes. This is a complex decision that involves the following considerations:

- What are the highest QPS queries, and what are the `WHERE` clauses for them?
- Cardinality of the column; it must be high.
- Do we want some rows to live together to support in-shard joins?
- Do we want certain rows that will be in the same transaction to live together?

Using the above considerations, in our use case, we can determine that:

- For the customer table, the most common `WHERE` clause uses `customer_id`. So, it shall have a Primary Vindex.
- Given that it has lots of users, its cardinality is also high.
- For the corder table, we have a choice between `customer_id` and `order_id`. Given that our app joins `customer` with `corder` quite often on the `customer_id` column, it will be beneficial to choose `customer_id` as the Primary Vindex for the `corder` table as well.
- Coincidentally, transactions also update `corder` tables with their corresponding `customer` rows. This further reinforces the decision to use `customer_id` as Primary Vindex.

There are a couple of other considerations out of scope for now, but worth mentioning:

- It may also be worth creating a secondary lookup Vindex on `corder.order_id`.
- Sometimes the `customer_id` is really a `tenant_id`. For example, your application is a SaaS, which serves tenants that themselves have customers. One key consideration here is that the sharding by the `tenant_id` can lead to unbalanced shards. You may also need to consider sharding by the tenant's `customer_id`.

Putting it all together, we have the following VSchema for `customer`:

```
{
  "sharded": true,
  "vindexes": {
    "hash": {
      "type": "hash"
    }
```

```
  },
  "tables": {
    "customer": {
      "column_vindexes": [
        {
          "column": "customer_id",
          "name": "hash"
        }
      ],
      "auto_increment": {
        "column": "customer_id",
        "sequence": "customer_seq"
      }
    },
    "corder": {
      "column_vindexes": [
        {
          "column": "customer_id",
          "name": "hash"
        }
      ],
      "auto_increment": {
        "column": "order_id",
        "sequence": "order_seq"
      }
    }
  }
}
```

Since the primary vindex columns are `BIGINT`, we choose `hash` as the primary vindex, which is a pseudo-random way of distributing rows into various shards. For other data types:

- For `VARCHAR` columns, use `unicode_loose_md5` or `unicode_loose_xxhash`.
- For `VARBINARY`, use `binary_md5` or `xxhash`.
- Vitess uses a plugin system to define vindexes. If none of the predefined vindexes suit your needs, you can develop your own custom vindex.

Apply VSchema

Applying the new VSchema instructs Vitess that the keyspace is sharded, which may prevent some complex queries. It is a good idea to validate this before proceeding with this step. If you do notice that certain queries start failing, you can always revert temporarily by restoring the old VSchema. Make sure you fix all of the queries before proceeding to the Reshard process.

```
helm upgrade vitess ../../helm/vitess/ -f 301_customer_sharded.yaml
```

```
vtctlclient ApplySchema -sql="$(cat create_commerce_seq.sql)" commerce
vtctlclient ApplyVSchema -vschema="$(cat vschema_commerce_seq.json)" commerce
vtctlclient ApplyVSchema -vschema="$(cat vschema_customer_sharded.json)" customer
vtctlclient ApplySchema -sql="$(cat create_customer_sharded.sql)" customer
```

```
vtctlclient ApplySchema -sql-file create_commerce_seq.sql commerce
vtctlclient ApplyVSchema -vschema_file vschema_commerce_seq.json commerce
vtctlclient ApplyVSchema -vschema_file vschema_customer_sharded.json customer
vtctlclient ApplySchema -sql-file create_customer_sharded.sql customer
```

Create new shards

At this point, you have finalized your sharded VSchema and vetted all the queries to make sure they still work. Now, it's time to reshard.

The resharding process works by splitting existing shards into smaller shards. This type of resharding is the most appropriate for Vitess. There are some use cases where you may want to bring up a new shard and add new rows in the most recently created shard. This can be achieved in Vitess by splitting a shard in such a way that no rows end up in the 'new' shard. However, it's not natural for Vitess. We have to create the new target shards:

```
helm upgrade vitess ../../helm/vitess/ -f 302_new_shards.yaml
```

```
kubectl apply -f 302_new_shards.yaml
```

Make sure that you restart the port-forward after you have verified with `kubectl get pods` that this operation has completed:

```
killall kubectl
./pf.sh &
```

```
for i in 300 301 302; do
 CELL=zone1 TABLET_UID=$i ./scripts/mysqlctl-up.sh
 SHARD=-80 CELL=zone1 KEYSPACE=customer TABLET_UID=$i ./scripts/vttablet-up.sh
done

for i in 400 401 402; do
 CELL=zone1 TABLET_UID=$i ./scripts/mysqlctl-up.sh
 SHARD=80- CELL=zone1 KEYSPACE=customer TABLET_UID=$i ./scripts/vttablet-up.sh
done

vtctlclient InitShardMaster -force customer/-80 zone1-300
vtctlclient InitShardMaster -force customer/80- zone1-400
```

Start the Reshard

This process starts the reshard operation. It occurs online, and will not block any read or write operations to your database:

```
# With Helm and Local Installation
vtctlclient Reshard -source_shards '0' -target_shards '-80,80-' Create customer.cust2cust
# With Operator
vtctlclient Reshard -source_shards '-' -target_shards '-80,80-' Create customer.cust2cust
```

All of the command options and parameters for `Reshard` are listed in our reference page for Reshard.

Validate Correctness

After the reshard is complete, we can use VDiff to check data integrity and ensure our source and target shards are consistent:

```
vtctlclient VDiff customer.cust2cust
```

You should see output similar to the following:

```
Summary for customer: {ProcessedRows:5 MatchingRows:5 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
Summary for corder: {ProcessedRows:5 MatchingRows:5 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
```

Switch Non-Primary Reads

After validating for correctness, the next step is to switch read operations to occur at the new location. By switching read operations first, we are able to verify that the new tablet servers are healthy and able to respond to requests:

```
vtctlclient Reshard -tablet_types=rdonly,replica SwitchTraffic customer.cust2cust
```

Switch Writes and Primary Reads

After the replica/rdonly reads have been switched, and the health of the system has been verified, it's time to switch writes. The usage is very similar to switching reads:

```
vtctlclient Reshard -tablet_types=master SwitchTraffic customer.cust2cust
```

Note

While we have switched reads and writes separately in this example, you can also switch all traffic, read and write, at the same time. If you don't specify the `-tablet_types` parameter `SwitchTraffic` will start serving traffic from the target for all tablet types.

You should now be able to see the data that has been copied over to the new shards:

```
mysql --table < ../common/select_customer-80_data.sql
Using customer/-80
Customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           5 | eve@domain.com     |
+-------------+--------------------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
```

```
|        3 |           3 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+

mysql --table < ../common/select_customer80-_data.sql
Using customer/80-
Customer
+-------------+----------------+
| customer_id | email          |
+-------------+----------------+
|           4 | dan@domain.com |
+-------------+----------------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        4 |           4 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Finalize and Cleanup

After celebrating your second successful resharding, you are now ready to clean up the leftover artifacts:

```
helm upgrade vitess ../../helm/vitess/ -f 306_down_shard_0.yaml
```

```
kubectl apply -f 306_down_shard_0.yaml
```

```
vtctlclient Reshard Complete customer.cust2cust

for i in 200 201 202; do
 CELL=zone1 TABLET_UID=$i ./scripts/vttablet-down.sh
 CELL=zone1 TABLET_UID=$i ./scripts/mysqlctl-down.sh
done
```

In this script, we just stopped all tablet instances for shard 0. This will cause all those vttablet and `mysqld` processes to be stopped. Beyond this, you will also need to manually delete the disk associated with this shard.

Tracing

Vitess tracing

Vitess allows you to generate Jaeger / OpenTracing compatible trace events from the Vitess major server components: `vtgate`, `vttablet`, and `vtctld`. To sync these trace events you need an OpenTracing compatible server (e.g. Jaeger). Vitess can send tracing events to this server in the Jaeger compact Thrift protocol wire format which is usually UDP on port 6381.

Configuring tracing

The first step of configuring tracing is to make sure you have tracing collectors properly setup. The tracing collectors must be located where they can be reached from the various Vitess components on which you want to configure tracing. We will not cover the entire setup process in this guide. The guide will cover the minimal config for testing/running locally, using the Jaeger docker container running on `localhost`. You can read more about Jaeger here.

Running Jaeger in docker You can follow the Jaeger getting started documentation here. In essence you need to run the Jaeger docker container:

```
$ docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.20
```

Note that you don't need to expose all these ports, Vitess only cares about port 6831 (the UDP compact Thrift Jaeger protocol listener). You will also need port 16686 for the Jaeger web UI to browse the spans reported.

Configuring tracing for vtgate, vttablet and vtctld Now that you have the Jaeger server running, you can add the necessary startup options to `vtgate`, `vttablet` and `vtctld`. This will enable you to send trace spans to the server. The command line options for doing this are the same across `vtgate`, `vttablet` and `vtctld`. Add the following options for a tracing agent running on the `localhost`:

```
-tracer opentracing-jaeger -jaeger-agent-host 127.0.0.1:6831 -tracing-sampling-rate 0.0
```

There are a few things to note:

- There are other tracing plugins and the `-tracer` option allows you to select them. Currently we have `opentracing-jaeger` and `opentracing-datadog`. Only `opentracing-jaeger` is covered in this document.
- `-jaeger-agent-host` should point to the `hostname:port` or `ip:port` of the tracing collector running the Jaeger compact Thrift protocol.
- The tracing sample rate (`-tracing-sampling-rate`) is expressed as a fraction from 0.0 (no sampling) to 1.0 (100% of all events are sent to the server). In the example, this option is set to zero, because we will be passing custom span contexts to the queries we want to trace. In this way, we only instrument the queries we want. This is recommended for large installations because it is typically very hard to organize and consume the volume of tracing events generated by even a small fraction of events from a non-trivial production Vitess system. However, if you just want events to flow automatically without you having to instrument queries, you can set this to a value other than `0.0` and skip the following section on instrumenting queries.

After adding these options, you must restart the Vitess components in question.

Instrumenting queries Now that you have the Vitess components setup, you can start instrumenting your queries to choose which queries (or application actions) for which you want to generate trace events. This is obviously an application-specific process, but there are a few things to note:

- The `SpanContext` id you have to instrument your Vitess queries with, in order for them to generate trace events, has a very specific format. It is recommended to use one of the Jaeger / OpenTracing client libraries to generate these for you. They take the format of a base64 string of a JSON object that, at it simplest, looks something like this:

  ```
  {"uber-trace-id":"{trace-id}:{span-id}:{parent-span-id}:{flags}"}
  ```

  Note the very specific format requirements in the documentation. Because of these requirements, it can be tiresome to generate them yourself, and it is more convenient to use the client libraries instead.

- Once you have the `SpanContext` string in its encoded base64 format, you can then generate your SQL query/queries related to this span to send them to Vitess. To inform Vitess of the `SpanContext`, you need to use a special SQL comment style, e.g.:

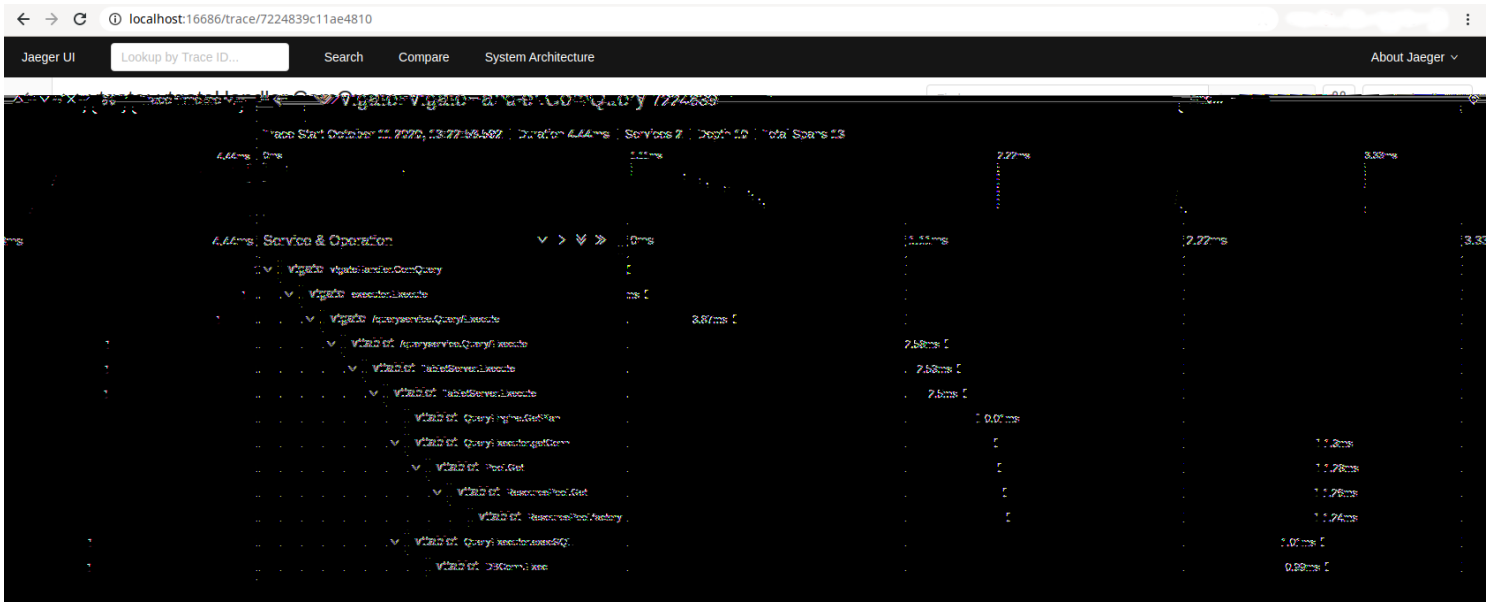
  ```
  /*VT_SPAN_CONTEXT=<base64 value>*/ SELECT * from product;
  ```

  There are additional notes here:

  – The underlying tracing libraries are very particular about the base64 value, so if you have any formatting problems (including trailing spaces between the base64 value and the closing of the comment); you will get many warnings in your `vtgate` logs.
  – When testing with, for example, the `mysql` CLI tool, make sure you are using the `-c` (or `--comments` flag), since the default is `--skip-comments`, which will never send your comments to the server (`vtgate`).

Inspecting trace spans in the Jaeger web UI This is beyond the scope of this guide. However, in general, if you have set everything above up correctly and you have instrumented and executed some queries appropriately, you can now access the Jager web UI to look at the spans recorded. If you are using the local docker container version of Jaeger, you can access the web UI in your browser at http://localhost:16686/.

You should be able to search for and find spans based on the `trace-id` or `span-id` with which your query/queries were instrumented. Once you find a query, you will be able to see the trace events emitted by different parts of the code as the query moves through `vtgate` and the `vttablet(s)` involved in the query. An example would look something like this:



Unmanaged Tablet

{{< info >}} This guide follows on from the local installation guide. {{< /info >}}

This guide uses the Vitess components vtctld, Topology Service and VTGate which have already been started in the local installation guide. It assumes that you have an existing MySQL Server setup that you would like to add to Vitess as a new keyspace, which we will call `legacy`. The same set of steps can be used to create a tablet that uses Amazon RDS, AWS Aurora, or Google CloudSQL.

Ensure all components are up

You should have previously executed `./101_initial_cluster.sh` in the get-started guide. This will ensure that you have a Topology Service, vtgate, vtctld. For the unmanaged MySQL instance, I will be using an instance running on `127.0.0.1:5726`:

```
source env.sh

# verify vtgate/vitess is up and running
mysql commerce -e 'show tables'

# verify my unmanaged mysql is running
mysql -h 127.0.0.1 -P 5726 -umsandbox -pmsandbox legacy -e 'show tables'
```

Output:

```
~/vitess/examples/local$ source env.sh
~/vitess/examples/local$
~/vitess/examples/local$ # verify vtgate/vitess is up and running
~/vitess/examples/local$ mysql commerce -e 'show tables'
+------------------------+
| Tables_in_vt_commerce |
+------------------------+
| corder                |
| customer              |
| product               |
+------------------------+
~/vitess/examples/local$ # verify my unmanaged mysql is running
~/vitess/examples/local$ mysql -h 127.0.0.1 -P 5726 -umsandbox -pmsandbox legacy -e 'show
   tables'
mysql: [Warning] Using a password on the command line interface can be insecure.
+------------------+
| Tables_in_legacy |
+------------------+
| legacytable      |
+------------------+
```

Start a tablet to correspond to legacy

The variables `TOPOLOGY_FLAGS` and `VTDATAROOT` should already be in the environment from sourcing env.sh earlier. We will call the new tablet UID 401.

```
mkdir -p $VTDATAROOT/vt_0000000401
vttablet \
 $TOPOLOGY_FLAGS \
 -logtostderr \
 -log_queries_to_file $VTDATAROOT/tmp/vttablet_0000000401_querylog.txt \
 -tablet-path "zone1-0000000401" \
 -init_keyspace legacy \
```

```
-init_shard 0 \
-init_tablet_type replica \
-port 15401 \
-grpc_port 16401 \
-service_map grpc-queryservice,grpc-tabletmanager,grpc-updatestream \
-pid_file $VTDATAROOT/vt_0000000401/vttablet.pid \
-vtctld_addr http://localhost:15000/ \
-db_host 127.0.0.1 \
-db_port 5726 \
-db_app_user msandbox \
-db_app_password msandbox \
-db_dba_user msandbox \
-db_dba_password msandbox \
-db_repl_user msandbox \
-db_repl_password msandbox \
-db_filtered_user msandbox \
-db_filtered_password msandbox \
-db_allprivs_user msandbox \
-db_allprivs_password msandbox \
-init_db_name_override legacy \
-init_populate_metadata &
```

Note that if your tablet is using a MySQL instance type where you do not have `SUPER` privileges to the database (e.g. AWS RDS, AWS Aurora or Google CloudSQL), you should omit the `-init_populate_metadata` flag. The `-init_populate_metadata` flag should only be enabled if the cluster is being managed through Vitess.

You should be able to see debug information written to screen confirming Vitess can reach the unmanaged server. A common problem is that you may need to change the authentication plugin to `mysql_native_password` (MySQL 8.0).

Assuming that there are no errors, after a few seconds you can mark the server as externally promoted to master:

```
vtctlclient TabletExternallyReparented zone1-401
```

Connect via VTGate

VTGate should now be able to route queries to your unmanaged MySQL server:

```
~/vitess/examples/local$ mysql legacy -e 'show tables'
+------------------+
| Tables_in_legacy |
+------------------+
| legacytable      |
+------------------+
```

You can even join between the unmanaged tablet and the managed tablets. Vitess will execute the query as a scatter-gather:

```
mysql> use commerce;
Database changed

mysql> select corder.order_id from corder inner join legacy.legacytable on
   corder.order_id=legacy.legacytable.id;
Empty set (0.01 sec)
```

Move legacytable to the commerce keyspace

Move the table:

```
vtctlclient MoveTables -tablet_types=master -workflow=legacy2commerce legacy commerce
   '{"legacytable": {}}'
```

Switch reads:

```
vtctlclient SwitchReads -tablet_type=rdonly commerce.legacy2commerce
vtctlclient SwitchReads -tablet_type=replica commerce.legacy2commerce
```

Switch writes:

```
vtctlclient SwitchWrites commerce.legacy2commerce
```

Drop source table:

```
vtctlclient DropSources commerce.legacy2commerce
```

Verify that the table was moved:

```
source env.sh

# verify vtgate/vitess is up and running
mysql commerce -e 'show tables'

# verify my unmanaged mysql is running
mysql -h 127.0.0.1 -P 5726 -umsandbox -pmsandbox legacy -e 'show tables'
```

Output:

```
~/vitess/examples/local$ source env.sh
~/vitess/examples/local$
~/vitess/examples/local$ # verify vtgate/vitess is up and running
~/vitess/examples/local$ mysql commerce -e 'show tables'
+------------------------+
| Tables_in_vt_commerce |
+------------------------+
| corder                |
| customer              |
| legacytable           |
| product               |
+------------------------+
~/vitess/examples/local$ # verify my unmanaged mysql is running
~/vitess/examples/local$ mysql -h 127.0.0.1 -P 5726 -umsandbox -pmsandbox legacy -e 'show
   tables'
mysql: [Warning] Using a password on the command line interface can be insecure.
```

User Management and Authentication

Vitess uses its own mechanism for managing users and their permissions through VTGate. As a result, the `CREATE USER...` and `GRANT...` statements will not work if sent through VTGate.

Authentication

The Vitess VTGate component takes care of authentication for requests so we will need to add any users that should have access to the Keyspaces via the command-line options to VTGate.

The simplest way to configure users is using a `static` auth method and we can define the users in a JSON formatted file or string.

```
$ cat > users.json << EOF
{
  "vitess": [
    {
      "UserData": "vitess",
      "Password": "supersecretpassword"
    }
  ],
  "myuser1": [
    {
      "UserData": "myuser1",
      "Password": "password1"
    }
  ],
  "myuser2": [
    {
      "UserData": "myuser2",
      "Password": "password2"
    }
  ]
}
EOF
```

Then we can load this into VTGate with the additional commandline parameters:

```
vtgate $(cat <<END_OF_COMMAND
    -mysql_auth_server_impl=static
    -mysql_auth_server_static_file=users.json
    ...
    ...
    ...
END_OF_COMMAND
)
```

Now we can test our new users:

```
$ mysql -h 127.0.0.1 -u myuser1 -ppassword1 -e "select 1"
+---+
| 1 |
+---+
| 1 |
+---+

$ mysql -h 127.0.0.1 -u myuser1 -pincorrect_password -e "select 1"
ERROR 1045 (28000): Access denied for user 'myuser1'
```

Password format

In the above example we used plaintext passwords. Vitess supports the MySQL mysql_native_password hash format, and you should always specify your passwords using this in a non-test or external environment. Vitess does not yet support the caching_sha2_password format that became the default for MySQL in 8.0.

To use a `mysql_native_password` hash, your user section in your static JSON authentication file would look something like this instead:

```
{
  "vitess": [
    {
```

```
      "UserData": "vitess",
      "MysqlNativePassword": "*9E128DA0C64A6FCCCDCFBDD0FC0A2C967C6DB36F"
    }
  ]
}
```

You can extract a `mysql_native_password` hash from an existing MySQL install by looking at the `authentication_string` column of the relevant user's row in the `mysql.user` table. An alternate way to generate this hash is to SHA1 the cleartext password string twice, e.g. doing it in MySQL for the cleartext password `password`:

```
mysql> SELECT UPPER(SHA1(UNHEX(SHA1("password")))) as hash;
+-------------------------------------------+
| hash                                      |
+-------------------------------------------+
| 2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
+-------------------------------------------+
1 row in set (0.01 sec)
```

So, you would use `*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19` as the `MysqlNativePassword` hash value for the cleartext password `password`.

UserData

In the static authentication JSON file, the `UserData` string is **not** the username; the username is the string key for the list. The `UserData` string does **not** need to correspond to the username, and is used by the authorization mechanism when referring to a user. It is usually however simpler if you make the `UserData` string and the username the same.

The `UserData` feature can be leveraged to create multiple users that are equivalent to the authorization layer (i.e. multiple users having the same `UserData` strings), but are different in the authentication layer (i.e. have different usernames and passwords).

Multiple passwords

A very convenient feature of the VTGate authorization is that, as can be seen in the example JSON authentication files, you have a **list** of `UserData` and `Password`/`MysqlNativePassword` pairs associated with a user. You can optionally leverage this to assign multiple different passwords to a single user, and VTGate will allow a user to authenticate with any of the defined passwords. This makes password rotation much easier; and less likely to require or cause downtime.

An example could be:

```
{
  "vitess": [
    {
      "UserData": "vitess_old",
      "MysqlNativePassword": "*9E128DA0C64A6FCCCDCFBDD0FC0A2C967C6DB36F"
    },
    {
      "UserData": "vitess_new",
      "MysqlNativePassword": "*B3AD996B12F211BEA47A7C666CC136FB26DC96AF"
    }
  ]
}
```

This feature also allows different `UserData` strings to be associated with a user depending on the password used. This can be used in concert with the authorization mechanism to migrate an application gracefully from one set of ACLs (or no ACLs) to another set of ACLs, by just changing the password used by the application.

In the example above, the username `vitess` has **two different** passwords that would be allowed, each resulting in different `UserData` strings (`vitess_old` or `vitess_new`) being passed to the VTTablet layer that can be used for authorization/ACL enforcement.

Other authentication methods

Other than the static authentication file method above, other authentication mechanisms are also provided:

- LDAP-based authentication
- TLS client certificate-based authentication

Running in Production

description: Instructions and guidelines for running Vitess in a production environment

Add or Delete a Cell

Add

To add a cell after a cluster is up and running, you start off by creating one using the same steps previously performed to create the first cell:

```
vtctlclient AddCellInfo \
  -root /vitess/cell2 \
  -server_address <cell2_topo_address> \
  cell2
```

Additionally, you will need to make the keyspace info visible in the cell. For every keyspace, issue the following:

```
vtctlclient RebuildKeyspaceGraph -cells=cell2 <keyspace>
```

And finally, deploy the VSchema with

```
vtctlclient RebuildVSchemaGraph -cells=cell2
```

{{< info >}} If the `cells` option is not specified, the rebuild deploys to all cells. {{< /info >}}

Once these steps are done, you can bring up the necessary mysqls, vttablets and vtgates under that cell.

Delete

To delete a cell, bring down all servers in that cell, and then remove its entry from the global topo with:

```
vtctlclient DeleteCellInfo -force cell2
```

If `-force` is not used the command will error out if any keyspace was deployed to that cell. There is currently no clean way to undeploy a keyspace from a cell. So, `-force` will need to be used for most use cases.

VTGates and vtctlds do not refresh themselves after a cell is deleted or updated. It is recommended that you restart them.

Once the Vitess components are restarted, the final step will be to bring down the cell-specific topo server.

If you had deployed a cell-specific toposerver, that can now be brought down. The deployed info under the cell's root (`/vitess/cell2`) will not be automatically deleted. You will have to manually delete that directory.

Backups and Restores

The default settings created by `mysqlctl` set the binlog `expire_logs_days` to 3. Once the binlogs expire, you will not be able to bring up new empty replicas that can catch up to the current state.

So, you need to take regular backups, probably through a cron job. Assuming that you've configured the shared storage and provided the correct parameters to the Vitess components, you should be able to create a backup as follows:

```
vtctlclient Backup cell1-101
```

{{< warning >}} If you are not using an online backup method like xtrabackup, the `Backup` command will shut down the MySQL instance to perform the operation. The instance will be unavailable until the backup is finished, the restarted MySQL instance is pointed back at the primary and caught up on replication. {{< /warning >}}

{{< info >}} It is recommended that you also periodically backup your binlogs. Vitess does not natively support this ability. You will need to set this up yourself. {{< /info >}}

Once a backup is taken, bringing up a subsequent vttablet-MySQL pair will cause it to restore from the backup instead of starting with a fresh MySQL instance. This will make it catch up from the beginning of time. You should see the following messages in the vttablet logs:

```
I0102 13:06:01.379759   30842 backup.go:227] Restore: looking for a suitable backup to
   restore
I0102 13:06:01.379820   30842 shard_sync.go:70] Change to tablet state
I0102 13:06:01.380757   30842 backupengine.go:221] Restore: found backup commerce/0
   2021-01-02.205158.cell1-0000000101 to restore
```

If a MySQL with existing data is restarted either manually or due to a failure, an automatic restore will not be performed. Instead, a recovery will be performed from the existing data files.

Please refer to the Backup and Restore documentation for more information.

Creating a cell

A Vitess cell is a logical grouping of servers that typically maps to an availability zone, region, or data center. The purpose of a cell is to provide isolation. The loss of one cell should not disrupt other cells. To fulfil this, Vitess allows you to configure separate cell-specific topo servers. There is no need to distribute the servers of a cell-specific toposerver to other cells. However, it is recommended that you bring up more than one instance in order to survive individual server failures.

Even if you do not want a multi-cell deployment, you still need to create at least one cell before bringing up the rest of the Vitess servers. If you do not plan to deploy multiple cells, you can reuse the global toposerver as the cell-specific one also.

You can use the `vtctlclient` alias to create one:

```
vtctlclient AddCellInfo \
  -root /vitess/cell1 \
  -server_address <cell_topo_address> \
  cell1
```

Note that the cell topo has its own root path. If reusing the same toposerver, you must ensure that they don't overlap.

The cell information is saved in the global toposerver. Vitess takes care of deploying the necessary information from the global topo to the cell-specific topos. Vitess binaries fetch the cell information from the global topo before switching to use the cell topo.

{{< info >}} You will only need to specify the topo global root for launching the Vitess servers. The cell-specific information including its root path will be automatically loaded from the cell info. {{< /info >}}

Mapping cells to zones and regions

Most public clouds offer a hierarchy of failure boundaries. Regions are data centers that are far apart. Depending on the distance, the latency between two regions can be in the 10s to 100s of milliseconds. Zones are partitions within a region where the machines are in different buildings. Latency between zones is typically in the range of sub-millisecond to 1-2ms.

There is also a cost to transferring data between zones and regions, and these costs can come into play when making decisions about how to layout the topology.

The general recommendation for Vitess is to map each cell to a zone. The main advantage of this approach is that it minimizes cross-zone data transfers to the extent possible, thereby minimizing cost.

If an application must be deployed across regions, then you can create more cells in the newer region, one for each zone.

Within a region, you could use a single topo cluster to serve all the cells, as long as their root paths are distinct, or you could use one topo cluster per cell. The decision depends on the kind of failure tolerance you want to build into the system.

For example, let us say that you plan to deploy in three zones and decided to use a shared topo. In this case, losing two zones will result in a full outage because the topo server in the third zone would become unavailable due to loss of quorum. On the other hand, deploying a separate topo cluster for each cell would allow the third zone to survive the loss of the other two zones.

If you intend to use more than one region for the sake of survivability, then it is recommended that you use at least three regions. This will allow you to deploy a balanced quorum of servers for the global topo.

If you have deployed in multiple regions and would like the flexibility of queries that go cross-cell within a region, you can create cell aliases. These aliases will indicate to the vtgates that they can send requests to the vttablets of a different cell if a no local vttablet is available.

Checklist

- Ensure that vtctlds come up successfully. If there is a failure, check the log files for any errors.
- Ensure that you can browse to the http port of vtctld. The dashboard should appear, after which you should be able to browse under the `Topology` tab and verify that the cell information is as you created it.
- If you configured a separate cell-specific topo, ensure that you can connect to it using the parameters in the cell information.
- Ensure that the cell-specific topos are reachable from other cells.

Browsing to the cell information should look like this screenshot:

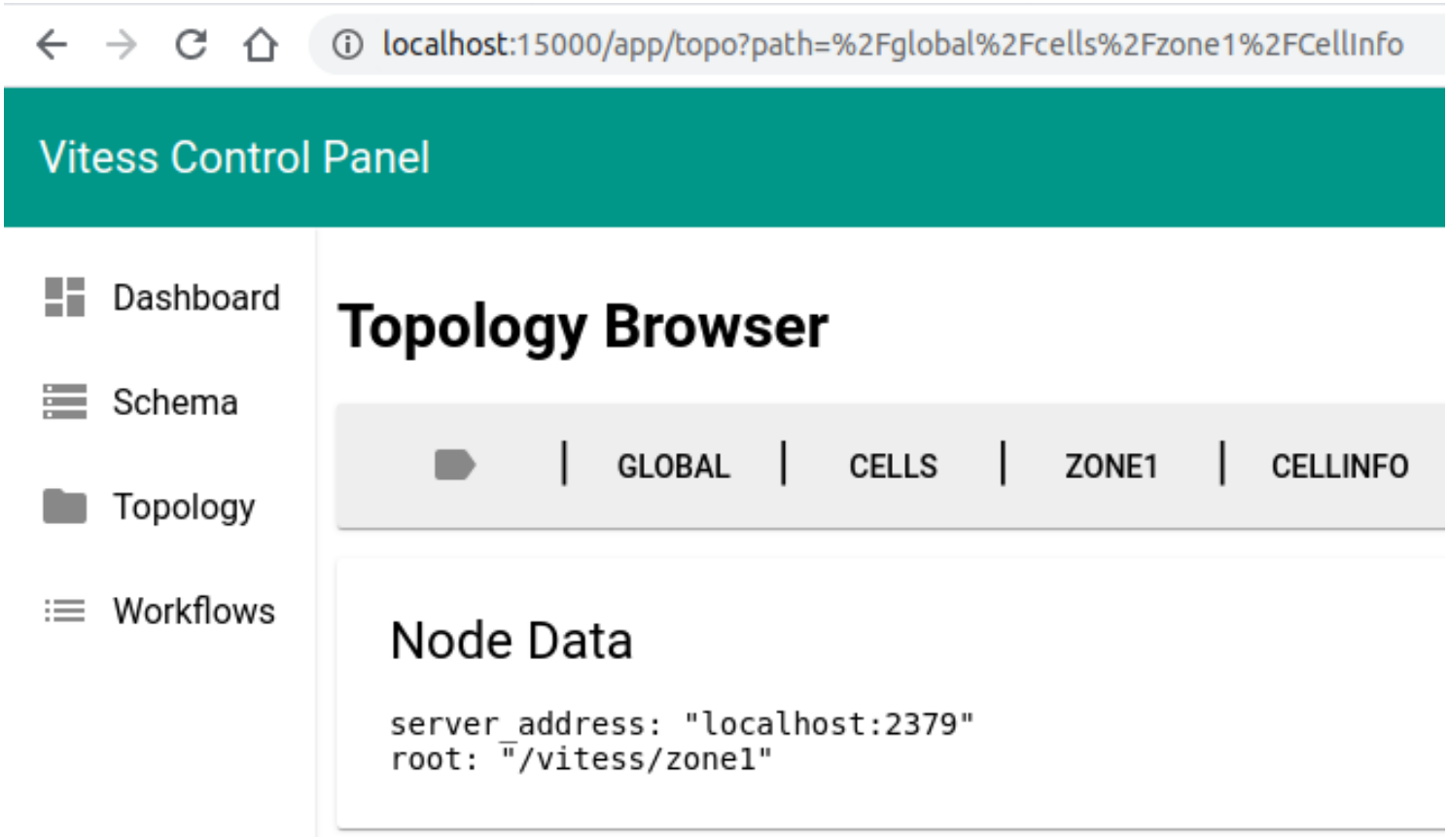


Figure 6: cell-in-topo

{{< info >}} Clicking on the `cell1` link will fail because metadata under that path will be created only when we bring up cell-specific Vitess components. {{< /info >}}

Delete a Keyspace

Although adding a keyspace is implicit, deleting one is not. In order to delete a keyspace, you must first bring down all tablets on that keyspace.

Following this, you should delete all shards of a keyspace. In the current example, `commerce` has only one shard (`0`). You can use the following command to delete it:

```
vtctlclient DeleteShard -recursive commerce/0
```

The `recursive` flag is required to ensure that the shard is deleted from all cells. If there are tablets still running against the shard, the command will fail.

You can add an `-even_if_serving` flag to ignore running tablets, but it is not recommended. Otherwise, this would cause existing vttablets to get confused about their tablet records being deleted.

If a keyspace has more than one shard, each one must be deleted individually.

{{< warning>}} Deleting a shard also deletes the metadata for all the backups, but it does not delete the backups themselves. This is something you have to do manually. {{< /warning >}}

Once all shards are deleted, you can delete the keyspace with:

```
vtctlclient DeleteKeyspace commerce
```

`DeleteKeyspace` also supports a `-recursive` flag that loops through all shards and deletes them recursively, but also ignores any tablets that are running. Note that restarting a vttablet for the deleted keyspace will cause the keyspace to be recreated. This is yet another reason for ensuring that all vttablets are shutdown upfront.

Exporting data from Vitess

Since VTGate supports the MySQL protocol, in many cases it is possible to use existing client utilities when connecting to Vitess. This includes using logical dump tools such as `mysqldump`, in certain cases.

This guide provides instructions on the required options when using these tools against a VTGate server for the purposes of exporting data from Vitess. It is recommended to follow the Backup and Restore guide for regular backups, since this method is performed directly on the tablet servers and is more efficient and safer for databases of any significant size. The dump methods that follow are typically not suitable for production backups, because Vitess does not implement all the locking constructs across a sharded database that are necessary to do a consistent logical backup while writing to the database. As a result, you will only be guaranteed to get a 100% consistent dump using these tools if you are sure that you are not writing to the database while running the dump.

mysqldump The default invocation of `mysqldump` attempts to execute statements which are not supported by Vitess, such as attempting to lock tables and dump GTID coordinates. The following options are required when using the `mysqldump` binary from MySQL 5.7 to export data from the `commerce` keyspace:

- `--lock-tables=off`: VTGate currently prohibits the syntax `LOCK TABLES` and `UNLOCK TABLES`.
- `--set-gtid-purged=OFF`: `mysqldump` attemps to dump GTID coordinates of a server, but in the case of VTGate this does not make sense since it could be routing to multiple servers.
- `--no-tablespaces`: This option disables dumping InnoDB tables by tablespace. This functionality is not yet supported by Vitess.
- `--skip-network-timeout`: This option is required when using `mysqldump` from MySQL 8.0 (#5401) with Vitess versions before 7.0.

For example to export the `commerce` keyspace using the `mysqldump` binary from MySQL 5.7:

```
$ mysqldump  --set-gtid-purged=OFF --no-tablespaces commerce > commerce.sql
```

{{< info >}} Vitess' support for LOCK and UNLOCK statements is currently syntax-only. As a result, Vitess will simply ignore LOCK and UNLOCK statements without taking any underlying action. It is therefore *unsafe* to perform a locking mysqldump against a database that is actively being written to, and you should pause writes completely while performing the dump; or be willing to deal with any data inconsistencies that result. {{< /info >}}

NOTE: You will be limited by the Vitess row limits in the size of the tables that you can dump using this method. The default Vitess row limit is determined by the vttablet option `-queryserver-config-max-result-size` and defaults to 10000 rows. So for an unsharded database, you will not be able to dump tables with more than 10000 rows, or N x 10000 rows if the table is fully sharded across N shards. Note that you should not blindly raise your row limits just because of this, it is an important Vitess operability and reliability feature. If you have large tables to dump, look into using go-mydumper instead.

To restore dump files created by `mysqldump`, replay it against a Vitess server or other MySQL server using the `mysql` command line client.

go-mydumper Alternatively, you can use a slight modification of the `go-mydumper` tool to export logical dumps of a Vitess keyspace. `go-mydumper` has the advantage of being multi-threaded, and so can run faster on a database that has many tables. For a database with just one or a handful of large tables, `go-mydumper` may not be that much faster than `mysqldump`.

For information on the Vitess-compatible fork of `go-mydumper`, see https://github.com/aquarapid/go-mydumper . Examples and instructions are available in the README.md in that repo. You will need to be able to compile golang binaries to use this tool.

`go-mydumper` creates multiple files for each backup. To restore a backup, you can use the `mysql` commandline client, but using the `myloader` tool as described in the `go-mydumper` repo above is easier and can be faster, since the loader is also multithreaded.

Global TopoServer

The first component to bring up is the Global TopoServer. As mentioned before, this can be a zookeeper or etcd cluster. The servers will likely have to be distributed across data centers for resilience. Please refer to the corresponding documentation for instructions on how to configure and launch these servers.

{{< info >}} Vitess previously supported Consul as a TopoServer. However, it is now deprecated because Consul does not natively support the ability to connect to multiple servers from a single program. This is a prerequisite for multi-region deployments. {{< /info >}}

Vitess will store global metadata like keyspaces, shards and cells in the topserver.

Choosing a TopoRoot

Vitess allows you to share the same global toposerver for multiple clusters. For example, you may want to run separate testing, staging and production clusters.

To support this separation, we allow you to assign a root directory in the Topo for each cluster. This directory is known as the `topo_global_root` and will need to be provided as a command line flag to every Vitess component.

{{< info >}} In the case of ZooKeeper, you will also need to create this directory on the server. For etcd, the path will be automatically created on first write. {{< /info >}}

The following command line options are required for every Vitess component:

```
-topo_implementation=etcd2 -topo_global_server_address=<comma_separated_addresses>
   -topo_global_root=/vitess/global
```

To avoid repetition we will use `<topo_flags>` in our examples to signify the above flags.

Note that the topo implementation for etcd is `etcd2`. This is because Vitess uses the v2 API of etcd.

{{< info >}} To be safe, you may want to bring up etcd with `--enable-v2=true`, even though it is the default value. Also, you will need to set the `ETCDCTL_API=2` environment variable before bringing up etcd. {{< /info >}}

Moving to a different TopoServer

It is generally not recommended that you migrate from one type of toposerver to another. However, if absolutely necessary, you can use the topo2topo command line tool to perform this migration.

Backups

It is important to periodically backup the data in the TopoServer. Although most of the information in the TopoServer can be manually reconstructed, it is still a painful task. This can be avoided if a recent backup was readily available.

Checklist

- Ensure toposerver is up, and that you can set and get values using their provided client tools.
- Ensure you have the mechanism to include the correct topo flags for all the components: `-topo_implementation`, `-topo_global_server_address` and `-topo_global_root`.
- If using zookeeper, ensure the global root path is created. It may be beneficial to do the same for etcd also.
- Ensure that the servers are reachable from other parts of the system where Vitess components will be launched.

InitShardMaster

The InitShardMaster step initializes the quorum by electing a primary and setting up replication for the replicas. Additionally, a database is created to store the data for the keyspace-shard. The default name for the database will be `vt_` followed by the keyspace. In our use case, it would be `vt_commerce`. You can override this default by providing an `init_db_name_override` flag to vttablet. However, all future invocations must continue to supply this parameter.

The command for `InitShardMaster` is as follows:

```
vtctlclient \
  InitShardMaster \
  -force \
  commerce/0 \
  cell1-100
```

Until this command is run, you may also see errors like this in the vttablet logs: `Cannot start query service: Unknown database 'vt_xxx'`. This is because the database will be created only after a primary is elected.

If you have semi-sync enabled and did not set up at least three replicas, InitiShardMaster could hang indefinitely. Even if it succeeds, future operations that perform failovers could cause this shard to go into a deadlocked state.

After this step, visiting the `/debug/status` page on the vttablets should show all the tablets as healthy:

{{< warning >}} `InitShardMaster` is a destructive command that resets all servers by deleting their binlog metadata. It should only be used for initializing a brand new cluster. {{< /warning >}}

{{< info >}} `InitShardMaster` will soon be deprecated. This action will be performed automatically by `vtorc` once it is released as production-ready. {{< /info >}}

Keyspaces and Shards

You can create keyspaces and shards using vtctlclient commands. However, they are not necessary because these are implicitly created as you bring up the vttablets.

The canonical information for keyspaces and shards is initially created in the global topo. This information is then deployed to the cell-specific topos through rebuild commands like `RebuildKeyspaceGraph` and `RebuildVSchemaGraph`. These commands are implicitly issued on your behalf whenever applicable. But there are situations where you will have to issue them manually. For example, if you create a new cell, you will have to issue these commands to copy the data into the new cell.

Health

Current status: healthy

Health Details

Current State	MASTER: Serving, Jan 2, 2021 at 12:33:33 (PST)

Health History

Time	Status	Tablet Type
Jan 2, 2021 at 12:33:33 (PST)	healthy	master
Jan 2, 2021 at 12:31:04 (PST)	not serving: no replication status	replica

healthy
: serving traffic.

unhappy
: will serve traffic only if there are no fully healthy tablets.

unhealthy
: will not serve traffic.

Figure 7: healthy-tablet

There are use cases where you may want to experimentally deploy changes to only some cells. Separating information from the global topo and local cells makes those experiments possible without affecting the entire deployment.

Tools like vtgate and vttablet consume information from the local copy of the topo.

An unsharded keyspace typically has a single shard named `0` or`-`. A sharded keyspace has shards named after the keyranges assigned to it, like `-80` and `80-`. Please refer to the section on shard naming for more info on how shards are named.

Monitoring

This section describes how to monitor Vitess components. Additionally, we recommend that you also add the necessary monitoring and alerting for the TopoServers as well as the MySQL instances running with each vttablet.

Tools

Vitess provides integrations with a variety of popular monitoring tools: Prometheus, InfluxDB and Datadog. The core infrastructure uses go's `expvar` package to export real-time variables visible as a JSON object served by the `/debug/vars` URL. The exported variables are CamelCase names. These names are algorithmically converted to the appropriate naming standards for each monitoring tool. For example, Prometheus uses a snake case conversion algorithm. In this case, the Prometheus exporter would convert the `Queries.Histograms.Select.500000` variable to `vttablet_queries_bucket{plan_type="Select",le="0.0005"}`.

In the sections below, we will be describing the variables as seen in the `/debug/vars` page.

The two critical Vitess processes to monitor are vttablet and vtgate. Additionally, we recommend that you setup monitoring for the underlying MySQL instances as commonly recommended in the MySQL community.

Beyond what the tools export, it is important to also monitor system resource usage: CPU, memory, network and disk usage.

There is a popular mixin contributed by Slack that is based on how they monitor Vitess internally. It can be used as a starting point if you intend to use Prometheus and Grafana.

Beyond the monitoring variables, the Vitess processes export additional information about their status on other URL paths. Some of those pages are for human consumption, and others are machine-readable meant for building automation.

VTTablet

/debug/status This page has a variety of human-readable information about the current vttablet and contains links to all the other URLs. You can look at this page to get a general overview of what is going on.

/debug/vars The following sections describe the various sub-objects of the JSON object exported by `/debug/vars`. These variables can be found at the top level of the JSON object:

Queries Vitess has a structured way of exporting certain performance stats. The most common one is the Histogram structure, which is used by Queries:

```
"Queries": {
  "Histograms": {
    "Select": {
      "1000000": 1138196,
      "10000000": 1138313,
      "100000000": 1138342,
      "1000000000": 1138342,
      "10000000000": 1138342,
      "500000": 1133195,
      "5000000": 1138277,
      "50000000": 1138342,
```

```
        "500000000": 1138342,
        "5000000000": 1138342,
        "Count": 1138342,
        "Time": 387710449887,
        "inf": 1138342
      }
    },
    "TotalCount": 1138342,
    "TotalTime": 387710449887
  },
```

The histograms are broken out into query categories. In the above case, `Select` is the only category, which measures all SELECT statements. An entry like `"500000": 1133195` means that `1133195` queries took under `500000` nanoseconds to execute.

Here is the full list of categories expanded from the source:

```
Select
SelectLock
Nextval
SelectImpossible
Insert
InsertMessage
Update
UpdateLimit
Delete
DeleteLimit
DDL
Set
OtherRead
OtherAdmin
SelectStream
MessageStream
Savepoint
Release
RollbackSavepoint
ShowTables
Load
Flush
LockTables
UnlockTables
```

The numbers are cumulative. For example, if you wish to count how many queries took between `500000ns` and `1000000ns`, the answer would be `1138196-1133195`, which is `1`.

Later below, we will be covering variables that break out these queries into further sub-categories like per-table, etc. However, we do not generate histograms for them because the number of values generated would be too big.

The thresholds are hard-coded. However, if you are integrating with an extermal tool like Prometheus, it will have its own thresholds.

The counters increment for the lifetime of the vttablet, and all values are updated in real-time.

`Queries.Histograms.Select.Count` is the total count in the `Select` category.

`Queries.Histograms.Select.Time` is the total time in the `Select` category.

`Queries.TotalCount` is the total count across all categories.

`Queries.TotalTime` is the total time across all categories.

There are other Histogram variables described below, and they will always have the same structure.

Use this variable to track:

- QPS
- Latency
- Per-category QPS. For replicas, the only category will be `Select`, but there will be more for masters.
- Per-category latency
- Per-category tail latency
- Per-category cost: This value is calculated as QPS*Latency. If the latency of a high QPS query goes up, it is likely causing more harm than the latency increase of an occasional query.


```
"Results": {
  "0": 0,
  "1": 0,
  "10": 1138326,
  "100": 1138326,
  "1000": 1138342,
  "10000": 1138342,
  "5": 1138326,
  "50": 1138326,
  "500": 1138342,
  "5000": 1138342,
  "Count": 1138342,
  "Total": 1140438,
  "inf": 1138342
}
```


Results is a simple histogram with no timing info. It gives you a histogram view of the number of rows returned per query. This does not include rows affected from write queries.

`Count` is expected to be the same as the `TotalCount` of the `Queries` histogram.

`Total` is the total number of rows returned so far.

Mysql Mysql is a histogram variable like Queries, except that it reports MySQL execution times. The categories are "Exec" and "ExecStream".

The vttablet queries exec time is roughly equal to the sum of the MySQL exec time, `ConnPoolWaitTime` and `Waits.Consolidations`.

Transactions `Transactions` is a histogram variable that tracks transactions. The categories are "Completed" and "Aborted". Since these are histograms, they include count as well as timings.

Waits Waits is a histogram variable that tracks various waits in the system. Right now, the only category is "Consolidations". A consolidation happens when one query waits for the results of an identical query already executing, thereby saving the database from performing duplicate work.

This variable used to report connection pool waits, but a refactor moved those variables out into the pool related vars.


```
"Errors": {
  "Deadlock": 0,
  "Fail": 1,
  "NotInTx": 0,
  "TxPoolFull": 0
},
```


Errors are reported under different categories. It's beneficial to track each category separately as it will be more helpful for troubleshooting. Right now, there are four categories. The category list may vary as Vitess evolves.

Errors/Query is a useful stat to track.

VTTablet also exports an InfoErrors variable that tracks inconsequential errors that don't signify any kind of problem with the system. For example, a dup key on insert is considered normal because apps tend to use that error to instead update an existing row. So, no monitoring is needed for that variable.


```
"InternalErrors": {
  "HungQuery": 0,
  "Invalidation": 0,
  "MemcacheStats": 0,
  "Mismatch": 0,
  "Panic": 0,
  "Schema": 0,
  "StrayTransactions": 0,
  "Task": 0
},
```


An internal error is an unexpected situation in code that may possibly point to a bug. Such errors may not cause outages, but even a single error needs to be escalated for root cause analysis.


```
"Kills": {
  "Queries": 2,
  "Transactions": 0,
  "ReservedConnection": 0
},
```


Kills reports the queries, transactions and reserved connections killed by vttablet due to timeout. It is a very important variable to look at during incidents.

TransactionPool* and FoundRowsPool* There are a few variables with the above prefixes that report the status of the two transaction pools:


```
"TransactionPoolAvailable": 300,
"TransactionPoolCapacity": 300,
"TransactionPoolIdleTimeout": 600000000000,
"TransactionPoolMaxCap": 300,
"TransactionPoolTimeout": 30000000000,
"TransactionPoolWaitCount": 0,
"TransactionPoolWaitTime": 0,
"FoundRowsPoolAvailable": 300,
"FoundRowsPoolCapacity": 300,
"FoundRowsPoolIdleTimeout": 600000000000,
"FoundRowsPoolMaxCap": 300,
"FoundRowsPoolTimeout": 30000000000,
"FoundRowsPoolWaitCount": 0,
"FoundRowsPoolWaitTime": 0,
```


The choice of which pool gets used depends on whether the application connected with the `CLIENT_FOUND_ROWS` flag or not.

- `WaitCount` will give you how often the transaction pool gets full, which causes new transactions to wait.
- `WaitTime/WaitCount` will tell you the average wait time.
- `Available` is a gauge that tells you the number of available connections in the pool in real-time. `Capacity-Available` is the number of connections in use. Note that this number could be misleading if the traffic is spiky.

Other Pool variables Just like `TransactionPool`, there are variables for other pools:

- `ConnPool`: This is the pool used for read traffic.
- `StreamConnPool`: This is the pool used for streaming queries.

There are other internal pools used by vttablet that are not very consequential.

`TableACLAllowed, TableACLDenied, TableACLPseudoDenied` The above three variables are table acl stats. They are broken out by table, plan and user.

`QueryPlanCacheSize` If the application does not make good use of bind variables, this value would reach the QueryCacheCapacity. If so, inspecting the current query cache will give you a clue about where the misuse is happening.

`QueryCounts, QueryErrorCounts, QueryRowCounts, QueryTimesNs` These variables are another multi-dimensional view of Queries. They have a lot more data than Queries because they're broken out into tables as well as plan. This is a priceless source of information when it comes to troubleshooting. If an outage is related to rogue queries, the graphs plotted from these vars will immediately show the table on which such queries are run. After that, a quick look at the detailed query stats will most likely identify the culprit.

`UserTableQueryCount, UserTableQueryTimesNs, UserTransactionCount, UserTransactionTimesNs` These variables are yet another view of Queries, but broken out by user, table and plan. If you have well-compartmentalized app users, this is another priceless way of identifying a rogue "user app" that could be misbehaving.

/debug/health This URL prints out a simple "ok" or "not ok" string that can be used to check if the server is healthy. The health check makes sure mysqld connections work, and replication is configured (though not necessarily running) if not on master.

/debug/status_details This URL prints out a JSON object that lists the state of all the variables that contribute to the current healthy or unhealthy state of a vttablet. If healthy, you should see something like this:

```
[
  {
    "Key": "Current State",
    "Class": "healthy",
    "Value": "MASTER: Serving, Jan 13, 2021 at 20:52:13 (PST)"
  }
]
```

/queryz, /debug/query_stats, /debug/tablet_plans, /livequeryz

- /queryz is a human-readable version of /debug/query_stats. If a graph shows a table as a possible source of problems, this is the next place to look to see if a specific query is the root cause. This is list is sorted in descending order of query latency. If the value is greater than 100 milliseconds, it's color-coded red. If it is greater than 10 milliseconds, it is color coded yellow. Otherwise, it is color coded gray.
- /debug/query_stats is a JSON view of the per-query stats. This information is pulled in real-time from the query cache. The per-table stats in /debug/vars are a roll-up of this information.
- /debug/table_plans is a more static view of the query cache. It just shows how vttablet will process or rewrite the input query.
- /livequeryz lists the currently running queries. You have the option to kill any of them from this page.

/querylogz, /debug/querylog, /txlogz, /debug/txlog

- /debug/querylog is a continuous stream of verbose execution info as each query is executed. This URL can generate a lot of data because it streams every query processed by vttablet. The details are as per this function: https://github.com/vitessio/vitess/blob/master/go/vt/vttablet/tabletserver/tabletenv/logstats.go#L202
- /querylogz is a limited human readable version of /debug/querylog. It prints the next 300 queries by default. The limit can be specified with a limit=N parameter on the URL.
- /txlogz is like /querylogz, but for transactions.
- /debug/txlog is the JSON counterpart to /txlogz.

/debug/consolidations This URL has an MRU list of consolidations. This is a way of identifying if multiple clients are spamming the same query to a server.

/schemaz, /debug/schema

- /schemaz shows the schema info loaded by vttablet.
- /debug/schema is the JSON version of /schemaz.

/debug/query_rules This URL displays the currently active query blacklist rules.

Alerting Alerting is built on top of the variables you monitor. Before setting up alerts, you should get some baseline stats and variance, and then you can build meaningful alerting rules. You can use the following list as a guideline to build your own:

- Query latency among all vttablets
- Per keyspace latency
- Errors/query
- Memory usage
- Unhealthy for too long
- Too many vttablets down
- Health has been flapping
- Transaction pool full error rate
- Any internal error
- Traffic out of balance among replicas
- Qps/core too high
- High replication lag
- Errant transactions
- Primary is in read-only mode

VTGate

/debug/status This is the landing page for a vtgate, which gives you the status of how a particular server is doing. Of particular interest there is the list of tablets this vtgate process is connected to, as this is the list of tablets that can potentially serve queries.

/debug/vars

VTGateApi This is the main histogram variable to track for vtgates. It gives you a break up of all queries by command, keyspace, and type.

HealthcheckConnections It shows the number of tablet connections for query/healthcheck per keyspace, shard, and tablet type.

TopologyWatcherErrors and TopologyWatcherOperations These two variables track events related to how vtgate watches the topology. It is particularly important to monitor the error count. This can act as an early warning sign if a vtgate is not able to refresh the list of tablets from the topo.

/debug/health This URL prints out a simple "ok" or "not ok" string that can be used to check if the server is healthy.

/debug/querylogz, /debug/querylog, /debug/queryz, /debug/query_plans These URLs are similar to the ones exported by vttablet, but apply to the current vtgate instance.

/debug/vschema This URL shows the vschema as loaded by vtgate.

Alerting For vtgate, here's a list of possible variables to alert on:

- Error rate
- Error/query rate
- Error/query/tablet-type rate
- vtgate serving graph is stale by x minutes (topology service is down)
- Qps/core
- Latency

Planning

This guide explains how to bring up and manage a Vitess cluster. We cover every individual component of Vitess and how they interact with each other. If you are deploying on Kubernetes, a lot of the wire-up is automatically handled by the operator. However, it is still important to know how the components work in order to be able to troubleshoot problems if they occur in production.

We assume that you are familiar with the setup of your production environment. If operating in Kubernetes, you should be able to access all the logs, and be able to reach any ports of the pods that are getting launched. In addition, you should be familiar with provisioning storage.

In self-hosted environments, you are expected to have the ability to troubleshoot network issues, firewalls and hostnames. You will also have to configure and setup certificates for components to talk to each other securely. This topic will not be covered in this guide. In addition, you are expected to perform all other sysadmin work related to provisioning, resource allocation, etc.

Vitess is capable of running on a variety of platforms. They may be self-hosted, in the public cloud, or in a cloud orchestration environment like Kubernetes.

In this guide, we will assume that we are deploying in a self-hosted environment that has multiple data centers. This setup allows us to better understand the interaction between the components.

Before starting, we assume that you have downloaded Vitess and finished the Get Started tutorial.

External tools

Vitess relies on two external components, and we recommend that you choose them upfront:

1. TopoServer: This is the server in which Vitess stores its metadata. We currently support zookeeper or etcd.
2. MySQL: Vitess supports MySQL/Percona Server 5.7 to 8.0, and MariaDB 10.0 to 10.3. MariaDB 10.4 is currently known to have installation issues (#5362). The most common deployments use MySQL 5.7.

In this guide, we will be covering the case where the MySQL instances are managed by Vitess. A different section covers the details of running against externally managed databases.

Provisioning

Some high level decisions have to be made about the number of cells you plan to deploy on. This will loosely tie into how many replicas you intend to run per MySQL primary. You are likely to deploy at least one replica per cell.

Vitess resource consumption is mostly driven by QPS, but there may be variations depending on your use case. As a starting point, you can use a rule of thumb of provisioning about 1 CPU for every 1000QPS. This CPU will be divided between MySQL, vttablets and vtgates, about 1/3 each. As for memory, you can start with approximately 1GB per CPU provisioned for Vitess components. MySQL memory will be largely guided by the buffer pool size, which may take some trial and error or prior experience to tune.

Resources for other servers like the toposerver, vtctld and vtorc are minimal. They are likely not going to exceed one CPU per server instance.

Environment variables

Setting up a few environment variables upfront will improve the manageability of the system:

- `VTDATAROOT`: Setting up this value will make Vitess create the MySQL data files under this directory. Other Vitess binaries will also use this variable to locate such files as needed. If not specified, the default value is `/vt`. Typically, no other files get stored under this directory. However, many idiomatic deployments tend to reuse this as root directory for other purposes like log files, etc.
- `VT_MYSQL_ROOT`: Informs Vitess about where to find the `mysqld` binary. If this is not specified, Vitess will try to find `mysqld` in the current `PATH`.

Vitess will automatically detect the flavor of MySQL and will adjust its behavior accordingly. You can override this behavior by specifying an explicit flavor with the `-db_flavor` command line argument to the various components.

Backups

Backups will need to be shared across vttablet instances and multiple cells. You need to plan and allocate shared storage that must be accessible from all cells. Depending on the choice made, you will need to prepare a group of command line arguments to include with the Vitess components to launch. Here is an example:

```
-backup_storage_implementation file -file_backup_storage_root <mounted_path_dir>
```

Please refer to the Backup and Restore guide for instructions on how to configure other storage options.

To avoid repetition we will use `<backup_flags>` in our examples to signify the above flags.

Logging

Vitess servers write to log files, and they are rotated when they reach a maximum size. It's recommended that you run at INFO level logging. The information printed in the log files can come in handy for troubleshooting. You can limit the disk usage by running cron jobs that periodically purge or archive them.

All Vitess servers accept a `-log_dir` as argument and will create the log files in that specified directory. For example:

```
-log_dir=${VTDATAROOT}/tmp
```

Ports

Ports and Network interactions in Vitess

Many/most of these ports are fully configurable, but we are listing their defaults or the defaults we use in examples here. Your environment may differ considerably, depending on your configuration options for the various components:

- Data path - e.g.:~/vitess/vtdataroot
  - Main query path:
    - application -> vtgate
      - port 3306 or 15306 (MySQL)
      - port 15999 (gRPC)
    - vtgate -> vttablet
      - port 16000 + vttablet UID (gRPC)
    - vttablet -> MySQL
      - local socket (if MySQL is local)
      - port 3306 (if MySQL is remote)
    - vttablet -> vttablet: vreplication within or across shards
      - port 16000 + vttablet UID (gRPC)
    - MySQL -> MySQL: within-shard replication
      - port 3306 (MySQL protocol)
- Control or meta-data paths - e.g.: ~/vitess/global
  - vtctld -> vttablet
    - port 16000 + vttablet UID (gRPC)
  - vtctlclient -> vtctld
    - port 15999 (gRPC)
  - vtgate -> topology server
    - Depends on topology server, e.g. for etcd typically port 2379
  - vttablet -> topology server
    - Depends on topology server, e.g. for consul typically port 8502
  - vtctld -> topology server
    - Depends on topology server, e.g. for zookeper typically port 2888
  - administrator using web browser -> vtgate web UI
    - port 15001 (HTTP)
  - administrator using web browser -> vttablet web UI
    - port 15000 + vttablet UID (HTTP)
  - administrator using web browser -> vtctld web UI
    - port 15000 (HTTP)
  - Metrics scraper (e.g. Prometheus) -> vtgate web port
    - port 15001 (HTTP)
  - Metrics scraper (e.g. Prometheus) -> vttablet web port
    - port 15000 + vttablet UID (HTTP)
  - Metrics scraper (e.g. Prometheus) -> vtctld web port
    - port 15000 (HTTP)

Reparenting

Once you have the cluster up and running, you should perform a trial failover using `PlannedReparentShard` to make sure it works as expected.

A typical use case for `PlannedReparentShard` is to use it during software updates. The command has a convenient `avoid_master` flag that allows you to specify the current vttablet you are going to perform maintenance on. If that is a primary, then it performs a failover to another eligible replica. Otherwise, it is a no-op.

You can also perform a "Planned Reparent" through the browser from the `vtctld` Dashboard.

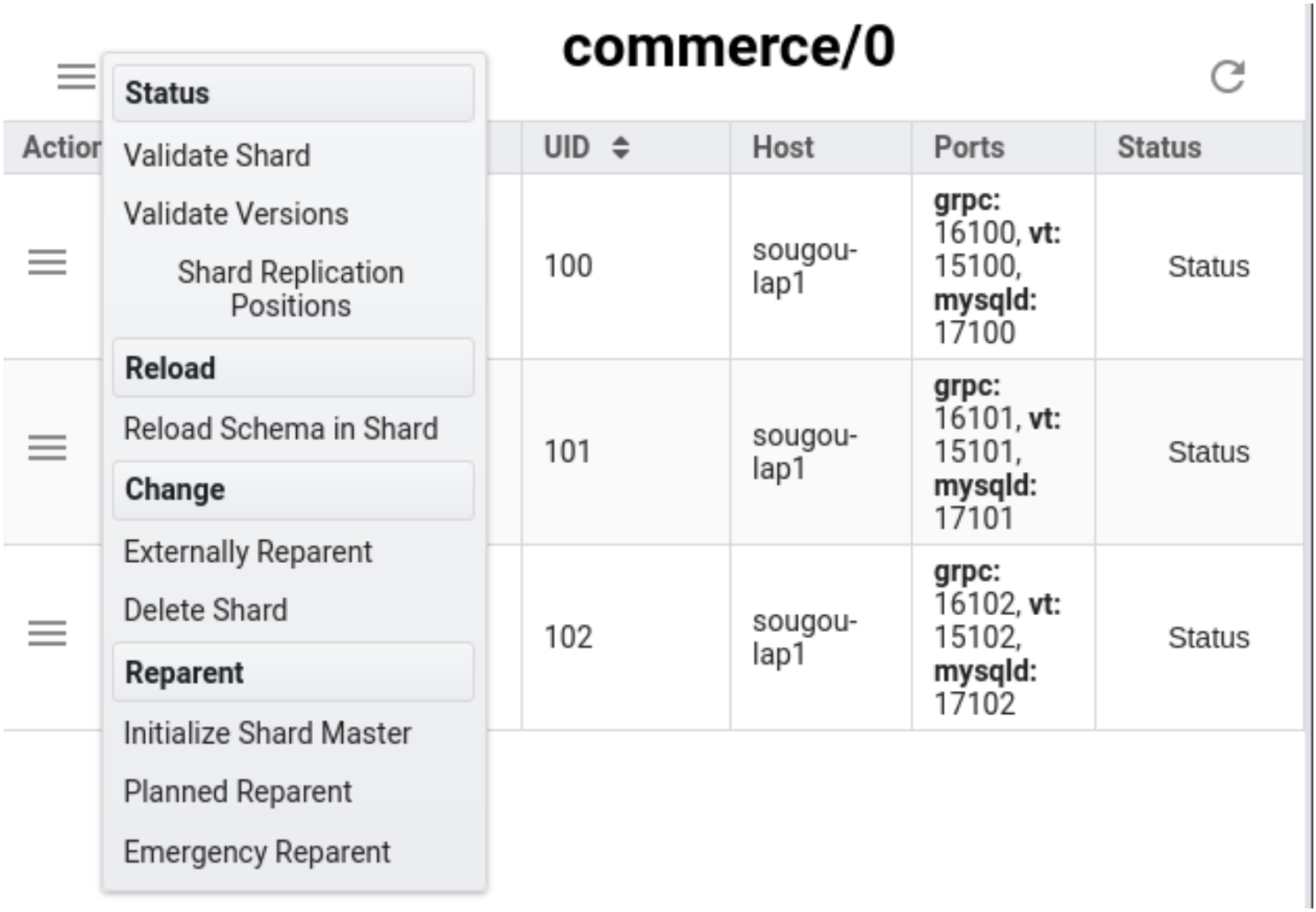


Figure 8: vtctld-reparenting

{{< info >}} The Vitess operator performs this step automatically when a container is gracefully brought down by Kubernetes, which also takes care of the use case of a software rollout. If a container or pod is brought down abruptly or crashes, then the primary will be unavailable until Kubernetes restarts it. However, if `vtorc` is also deployed, it will detect this and failover to another eligible replica as the primary. `vtorc` will not intervene during a graceful shut down. {{< /info >}}

If you are running `vtorc`, you can perform the same function from the dashboard by doing a drag-and-drop operation of the nodes.

For more information, please refer to the Reparenting section.

{{< info >}} A reparenting operation can fail in the middle. If so, it is possible for the system to be in a situation where two vttablets report themselves as primary. If this happens, the one with the newer timestamp wins. The vtgates will automatically treat the newer primary as authoritative. The system will eventually heal itself because the vttablets use a registration protocol via the global topo and the older tablet will demote itself to a replica when it notices that it is no longer the primary. {{< /info >}}

Troubleshooting

Understanding the Components

The secret to troubleshooting a Vitess cluster well comes from knowing how all the components are wired together.

Of these connections, the most important one is the query serving path:

- The application sends a request to a vtgate.
- The vtgate forwards that request to one or more vttablets.
- The vttablets in turn send the request to MySQL.

If there is any kind of problem with serving queries, these are the components to drill into.

VTGates and vttablets connect to the global and cell-specific toposerver. They use these toposervers to broadcast their state as well as to discover configuration changes. Additionally, vtgates receive health information from the vttablets they are connected to, and use this information to direct traffic in real-time. If there is any kind of problem with configuration, these are the areas to focus on.

The first step to troubleshooting is to have an established baseline. It is recommended that monitoring is set up for all the components as directed in the monitoring section. When there is an incident, the change in the graphs will likely help with identifying the root cause.

Query Serving

Most of the focus of Vitess monitoring centers around the cost of a query. Although not always true, we use the following rules of thumb to get a rough estimate:

- In MySQL, the time taken by a query roughly translates to its actual cost. More often than not, the limiting factor is the number of disk IOPS.
- In Vitess components, the payload contributes to the cost. The limiting factor is most often the CPU, followed by memory.
- The cost of parsing and analyzing the input query has started to become significant. This is typically driven by the size and complexity of the query.

Let us now go through some fire drills.

Too many connections If you see an error that contains the string `Too many connections (errno 1040)(sqlstate 08004)`, then it means that the maximum number of connections allowed by MySQL has been exceeded.

The remedy is to further increase the `max_connections` settings in MySQL. The other alternative is to reduce the pool sizes. As recommended before, the `max_connections` value should be set about double the number of connections allocated for pools.

Elevated Query Latency This is one of the most common problems. It can show up as just an elevated latency with no external impact. Sometimes, the problem can be more acute where queries return errors due to timeouts. There can be multiple root causes.

Approach 1: Validate resource usage and remove bottlenecks Check vtgate CPU usage: If the vtgate CPU is too high or is getting throttled, it could be the root cause. Bringing up another vtgate to take on additional load or increasing the CPU quota of the VTGate should resolve the problem.

Check vttablet CPU usage: If vttablet CPU usage is maxed out, the immediate solution is to increase the quota. If the vttablet is a replica, then bringing up more replicas should help distribute the load. If it is a master, then you need to plan on splitting the shard into smaller parts, or reshard in the case of an unsharded keyspace.

Check MySQL resource usage: If MySQL is resource constrained, for example IOPS, then you need to provision more. Also, you will then need to add more replicas or plan a reshard or split just like in the case of a vttablet. More often than not, MySQL will run into resource constraints before vttablet does.

Check Network packet limits. Some cloud providers limit network capacity by limiting the number of packets per second. Check to see if this limit has been reached. If so, you will need to request a quota increase from the cloud provider.

If none of the processes are resource constrained, the next step is to check connection pool waits in vttablet. If there are long waits, then we know that the connection pool is under-provisioned. Restarting the vttablets with increased pool sizes should fix the problem.

If latency continues to be elevated after increasing the pool size, it most likely means that the contention has shifted from vttablet to MySQL. In that case, it is likely that MySQL is the one resource constrained. You may have to roll back the pool size increase and instead look at increasing resources for MySQL.

Approach 2: Find the problematic query Typically, resource constraints are encountered during the early stages of a rollout where things are not tuned well. Once the system is well tuned and running smoothly, the root cause is likely to be related to how the application sends queries.

We first need to determine if this is an overall increase in application traffic or a change in specific query patterns.

Inspect the per-tablet and per-user graphs to see if the increase is uniform across all tables or is to only a specific table.

If the increase is across the board, then it is likely a general overload. The mitigation is to increase provisioning or address the root cause of the overload if it was not expected.

If the increase is on a specific table, then this data itself is sufficient to determine the root cause. If not, we may need to drill down further to identify if a specific query is causing the problem. There are two approaches:

- Inspect the `/queryz` page and look at the stats for all queries of that table. It is very likely that the problematic ones have already risen to the top of the page and may be color-coded red. If not, talking a few snapshots of the page and comparing the stats should help identify the problematic query.
- Real-time stream from `/debug/querylog`, filtering out unwanted tables and observing the results to identify the problematic query.

Once the query is identified, the remedy depends on the situation. It could be a rewrite of the query to be more efficient, the creation of an additional index, or it could be the shutdown of an abusive batch process.

The above guidelines have not required you to inspect MySQL. Over time, Vitess has evolved by improving its observability every time there was an incident. However, there may still be situations where the above approach is insufficient. If so, you will need to resort to looking inside MySQL to find out the root cause.

The actual identification of the root cause may not be as straightforward as described above. Sometimes, an incident is caused by multiple factors. In such cases, using first principles of troubleshooting and understanding how the components communicate with each other may be the only way to get to the bottom of a problem. If you have exhausted all the recommendations given so far and still have not found the root cause, you may have to directly troubleshoot the problem at the MySQL level.

Elevated Error Rates The analysis for elevated error rates for read queries follows steps similar to elevated latency. You should essentially use the same drill down approach to identify the root cause.

Transaction timeouts Transaction timeouts manifest as the following errors:

```
ERROR 1317 (HY000): vtgate: http://sougou-lap1:15001/: vttablet: rpc error: code = Aborted
   desc = transaction 1610909864463057369: ended at 2021-01-17 10:58:49.155 PST (exceeded
   timeout: 30s) (CallerID: userData1)
```

If you see such errors, you may have to do one of the following:

- Increase the transaction timeout in vttablet by setting a higher value for `queryserver-config-transaction-timeout`.
- Refactor the application code to finish the transaction sooner.

It is recommended to minimize long running transactions in MySQL. This is because the efficiency of MySQL drastically drops as the number of concurrent transactions increases.

Transaction connection limit errors If you encounter errors that contain the following text: `transaction pool connection limit exceeded`, it means that your connection pool for transactions is full and Vitess timed out waiting for a connection. This issue can have multiple root causes.

If your transaction load is just spiky, then you may just have to increase the pool timeout to make the transaction wait longer for a connection. This can be increased by setting the `queryserver-config-txpool-timeout` flag in vttablet. The default value is one second.

It is also possible that you have underprovisioned the transaction pool size. If so, you can increase the size by changing the value for `queryserver-config-transaction-cap`. Note that it is risky to increase the pool size beyond the low hundreds because MySQL performance can drastically deteriorate if too many concurrent transactions are opened.

Another possibility is that the application is unnecessarily keeping transactions open for too long thereby causing the pool to get full. To identify this, you can look at the vttablet logs. Every time the pool gets full, the list of transactions that were occupying those connections are printed out. Looking at those transactions should help you identify the root cause. This logging is throttled to prevent log spam.

Frequent application crashes can also leave transactions open in the pool until timeout. Vitess tries to detect this situation and proactively rolls back such transactions, but it is not always reliable. If this is the case, the log file will contain many unfinished transactions that were rolled back.

If you are starting to see these errors due to steady organic growth, it may be time to split the database or reshard.

Errant GTIDs An errant GTID incident is one where the MySQL instances of a shard do not agree on the events in the binlogs. This means that the data on those instances has potentially diverged. Errant GTIDs are often introduced due to operator errors. For example, someone could use DBA privileges to write data to a replica.

In Vitess, you can get into an errant GTID situation if a primary is network partitioned and you make a decision to proceed forward with an `EmergencyReparentShard`. This will essentially cause the old primary to get indefinitely stuck due to transactions waiting for a semi-sync ack. The natural instinct would be to restart that server. However, such a restart will cause MySQL to go into recovery mode and complete those pending transactions thereby causing divergence.

`vtorc` can be used to detect errant GTIDs. You can also set up your own monitoring to detect this situation. This can be performed by ensuring that the GTIDs of the replicas are always a subset of the primary.

If an errant GTID is detected, the first task is to identify the GTIDs that are diverged. If the divergence did not cause any data skew, you could choose to create dummy transactions with those extra GTIDs on instances that do not contain them. If you are running `vtorc`, you can use it to perform this fix.

If the data has diverged, you have to make a decision about which one is authoritative. Once that is decided, it is recommended that you first take a backup of the instance that you have determined to be authoritative. Following this, you can bring down the instances that were non-authoritative and restart them with empty directories. This will trigger the restore workflow that will start with the authoritative backup. Once restored, the MySQL will point itself to the current primary and catch up to a consistent state.

Replica vttablet not receiving queries A number of reasons can cause a vtgate to not send queries to a replica vttablet. The first step is to visit the `/debug/status` page of vtgate and look at the `Health Check Cache` section.

If the vttablet entry is present, but is color coded red and displays an error message, it means that vtgate is seeing the vttablet as unhealthy. Once you fix the error that causes the problem, traffic should resume to the vttablet. There can be many reasons for the unhealthiness:

- vttablet may not be reachable: troubleshoot connectivity from the vtgate machine to the vttablet, make sure firewall rules allow access, ports are reachable, etc.
- vttablet is reporting itself as unhealthy: Fix the root cause. For example, the MySQL may be lagging too much, or vttablet may have trouble connecting to the MySQL instance, etc.

If the vttablet entry is absent, then it means that vtgate has not discovered the vttablet yet. Check to see if vtgate is having trouble connecting to the topo server. If there was a problem, there should be errors in the log file like `cannot get tablets`. If such errors are found, fix the root cause and verify again.

It may be worth proactively monitoring `TopologyWatcherErrors` and `TopologyWatcherOperations`. Alerting on errors can help identify these problems early.

If there are no topo errors in vtgate, check to see if the tablet record has been created by vttablet using the `vtctlclient ListAllTablets` command. If the tablet record is absent or does not contain the correct host and port info, check the vttablet logs to see if it has trouble connecting to the topo and is unable to publish its existence. If there are errors, fixing the issue should resolve the problem.

Read-only errors If you see the following error string `The MySQL server is running with the --read-only option so it cannot execute this statement (errno 1290)(sqlstate HY000)` while trying to write to the primary, then it likely means that a previous `PlannedReparentShard` operation failed in the middle.

Re-executing `PlannedReparentShard` against that master should fix the problem. If this operation fails with an error saying that there is no current master, you may have to issue an `EmergencyReparentShard` to safely elect a master.

If `vtorc` is running, no action is needed because `vtorc` will notice this state and fix it in a safe manner.

This error can also be encountered if a new primary has been elected, but the older vttablet continues to think that it is still the primary. If this is the situation, then it is transient and will heal itself as long as components are able to communicate with each other. In this situation, the older vttablet will be in read-only mode. VTGates that are trying to send the writes to it will fail.

Eventually, the new primary will inform the vtgates of its existence, and they will start sending traffic to the new primary instead of the old one. The old primary will also eventually notice that a new primary was elected. When it notices it, it will demote itself to a replica.

Local TopoServer of a Cell is getting overloaded This situation can happen if a large number of vtgates continuously spam the local toposerver to check for changes in the list of tablet servers. If this is the case, you may have to reduce the polling frequency of the vtgates by reducing the `-tablet_refresh_interval` value.

Global Topo or Cell Topo is down Vitess servers are built to survive brief topo outages in the order of many minutes. All Vitess servers cache the necessary information to serve traffic. If there is an outage, they use the cached information to continue serving traffic.

However, during such an outage, you may not be able to perform cluster maintenance operations like a reparent, resharding, or bringing up new servers.

Topo Complete Data Loss In the unforeseen circumstance of a total data loss of the topo servers, a Vitess cluster can be restored to an operational state by performing the following actions:

- Bring up a brand new set of empty topo servers.

- Recreate the cell info as before.
- Restart all the vttablets.
- Upload the VSchema for the recreated keyspaces.

If you are in the middle of a reshard, make sure you restart the source vttablets first. This order will ensure that they get marked as serving. Any shards that get added later that overlap with existing keyranges will be marked as non-serving.

vtctld

vtctld is not required to be highly available because it is not in the serving path of a query. Nevertheless, it may be wise to bring up more than a single instance. Typically, users bring up one instance per cell.

Even if brought up within each cell, vtctld itself is not tied to that cell. It will attempt to access all servers of all cells. You can bring up vtctld with the following invocation:

```
vtctld <topo_flags> <backup_flags> \
  -log_dir=${VTDATAROOT}/tmp \
  -port=15000 \
  -grpc_port=15999 \
  -service_map='grpc-vtctl'
```

If the TopoServer is unreachable, or if the topo flags are incorrectly configured, vtctld will fail to start. You may see an error message like the following in the logs:

```
F0426 11:11:40.363545   14833 server.go:223] Failed to open topo server
   (etcd2,localhost:2379,/vitess/global): dial tcp 127.0.0.1:2379: connect: connection
   refused
```

The `service_map` flag allows you to configure the grpc APIs that a Vitess server exposes as grpc. If grpc-vtctl is not specified as a service_map for vtctld, you will not be able to access it using `vtctlclient`.

vtctld is usually not very resource intensive. But you may need to provision more if you plan to run the `VDiff` command. This functionality will soon be moved to vttablet.

vtctlclient

Since we will be using `vtctlclient` often, it will be convenient to configure an alias for it:

```
alias vtctlclient="command vtctlclient -server <vtctld_grpc_address> -log_dir
   ${VTDATAROOT}/tmp -alsologtostderr"
```

{{< info >}} We intend to move these arguments into an init file. Once that is done, there will be no need to set up the alias any more. {{< /info >}}

The next step will be to create a cell.

vtgate

VTGates are the primary interface to the application. They do not have any persistent state. You can bring up as many vtgates as necessary. An application can connect to any vtgate to get access to all the servers within a cell. We recommend configuring a load balancer to distribute traffic between them. A typical rule of thumb would be to bring up as many vtgates as there are vttablets.

It is better to bring up multiple vtgates within a single machine rather than trying to run a single instance that tries to use all the resources. This approach enables better amortization of Go's garbage collector.

You can bring up the vtgates before creating any keyspaces or bringing up vttablets. The advantage of bringing up the vtgates last is that it allows them to immediately discover the existing vttablets. Otherwise, it will take up to one polling cycle before

they are discovered. The default value for this flag (`-tablet_refresh_interval`) is one minute. If you intend to reduce this value, ensure that the increased polling frequency will not overwhelm the toposerver.

VTGate requires a cell to operate in. A vtgate's main job is to forward requests to the vttablets in the local cell. However, vtgates can go cross-cell in two situations:

- vtgate receives queries to the primary, and the primary is not in the current cell.
- vtgate was configured to go to other cells in case no local vttablets were available.

Here is a sample vtgate invocation:

```
vtgate <topo_flags> \
  -log_dir=${VTDATAROOT}/tmp \
  -cell=cell1 \
  -cells_to_watch=cell1 \
  -port=15001 \
  -mysql_server_port=15306 \
  -mysql_auth_server_impl=static \
  -mysql_auth_server_static_file=mysql_creds.json \
  -grpc_port=15991 \
  -service_map='grpc-vtgateservice' \
  -vschema_ddl_authorized_users'=dba'%
```

VTGate uses the global topo to get the topo addresses of the cells it has to watch. For this reason, you do not need to specify the topo addresses for the current cell.

VTGate does not require `<backup_flags>`.

For sending primary queries across cells, you must specify an additional `cells_to_watch` flag. This will make the vtgates watch those additional cells, and will allow them to keep track of the primaries in those cells.

The `cells_to_watch` flag is a required parameter and must at least include the current cell. This is an issue we will fix soon.

Going cross-cell for non-primary requests is an advanced use case that requires setting up cell aliases. This topic will not be covered in this user guide.

For those who wish to use the MySQL protocol, you must specify a `mysql_server_port` and a `mysql_auth_server_impl` for configuring authentication. Predefined auth servers are `clientcert`, `static`, `ldap` and `none`. The most commonly used authentication is `static` that allows you to specify the credentials through a `mysql_auth_server_static_file` parameter.

The `vschema_ddl_authorized_users` specifies which users can alter the vschema by issuing "vschema ddls" directly to vtgate. VSchema DDL is an experimental feature that will be documented soon.

Here are the contents of an example file that shows the ability to specify MySQL native passwords as well as plain text:

```
{
  "mysql_user": [
    {
      "MysqlNativePassword": "*9E128DA0C64A6FCCCDCFBDD0FC0A2C967C6DB36F",
      "Password": "mysql_password",
      "UserData": "mysql_user"
    }
  ],
  "mysql_user2": [
    {
      "Password": "mysql_password",
      "UserData": "mysql_user"
    }
  ],
  "mysql_user3": [
    {
```

```
      "MysqlNativePassword": "*9E128DA0C64A6FCCCDCFBDD0FC0A2C967C6DB36F",
      "UserData": "mysql_user"
    }
  ]
}
```

For those who wish to use the Java or Go grpc clients to vtgate, you must also configure `grpc_port` and specify the service map as `service_map='grpc-vtgateservice'`. Note that the `VStream` feature is only available via grpc.

You can also set the following flags to control load-balancing for replicas:

- `discovery_high_replication_lag_minimum_serving`: If the replication lag of a vttablet exceeds this value, vtgate will treat it as unhealthy and will not send queries to it. This value is meant to match vttablet's `unhealthy_threshold` value.
- `discovery_low_replication_lag`: If a single vttablet lags beyond this value, vtgate will not send it any queries. However, if too many replicas exceed this threshold, then vtgate will send queries to the ones that have the least lag. A weighted average algorithm is used to exclude the outliers. This value is meant to match vttablet's `degraded_threshold` value.

A vtgate that comes up successfully will show all the vttablets it has discovered in its `/debug/status` page under the `Health Check Cache` section.

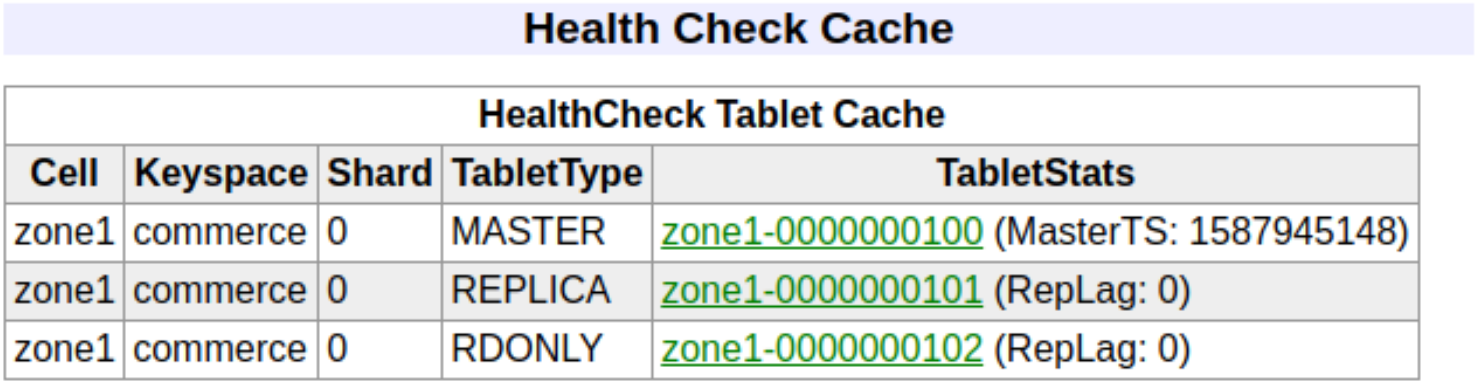


Figure 9: vtgate-healthy-tablets

If vtgates cannot connect to one of the vttablets it discovered from the topo, or if the vttablet is unhealthy, it will be shown in red in the `Health Check Cache`, and a corresponding error message will be displayed next to it:

You can verify that the vtgates came up successfully by using the MySQL client:

```
~/...vitess/examples/local> mysql -h 127.0.0.1 -P 15306 -u=mysql_user -p=mysql_password
[snip]
mysql> show databases;
+-----------+
| Database |
+-----------+
| commerce |
+-----------+
1 row in set (0.00 sec)
```

The `show databases` command presents the `commerce` keyspace as a database. Under the covers, the MySQL database backing it is actually `vt_commerce`.

Congratulations! You have successfully brought up a Vitess cluster.

vtorc (experimental)

`vtorc` is a fork of the Orchestrator customized to run as a Vitess component. It is available as an experimental feature. It still needs thorough testing, and more features need to be added.

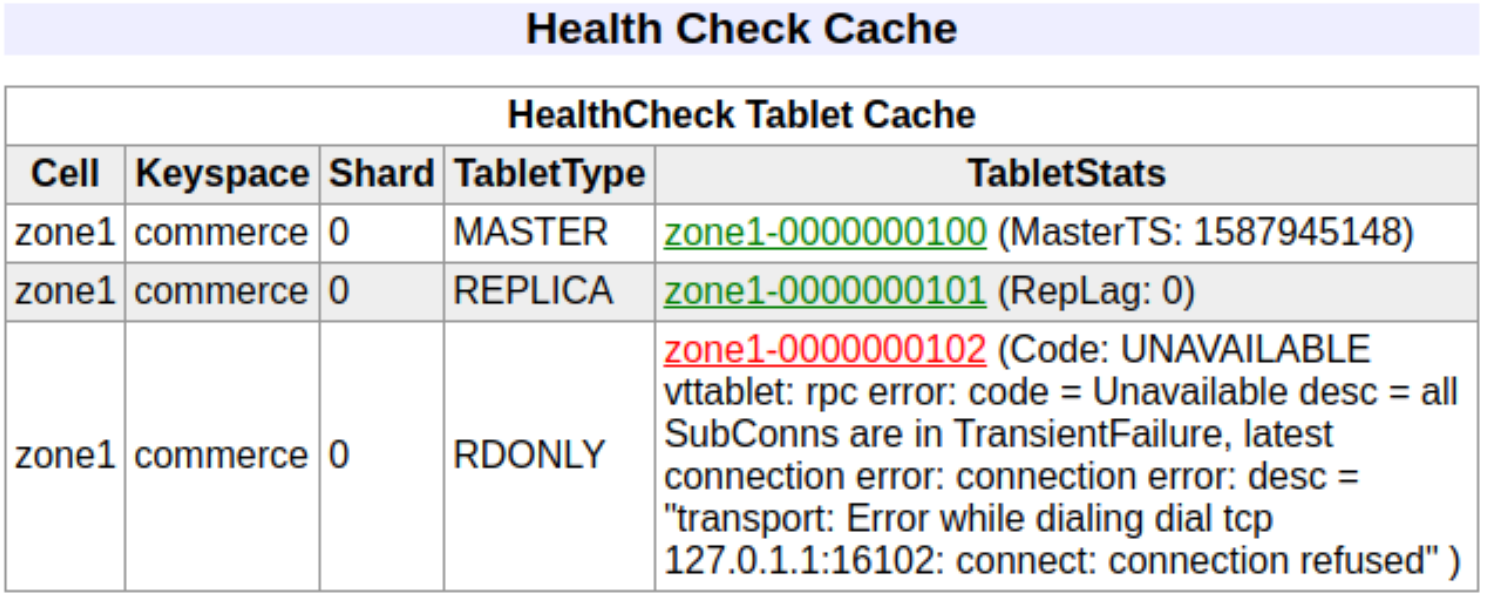


Figure 10: vtgate-partially-healthy-tablets

In order to configure `vtorc`, you have to make the following changes to `vttablet`:

- Remove `-enable_semi_sync=true`: This part will be managed by vtorc instead.
- Add `-disable_active_reparents=true`: This part will prevent vttablet from fixing replication, and will rely on vtorc instead.

As mentioned before, bringing up `vtorc` also lets you avoid performing the `InitShardMaster` step.

`vtorc` requires a config file to be launched. A sample is available here:

```
{
  "Debug": true,
  "MySQLTopologyUser": "orc_client_user",
  "MySQLTopologyPassword": "orc_client_user_password",
  "MySQLReplicaUser": "vt_repl",
  "MySQLReplicaPassword": "",
  "RecoveryPeriodBlockSeconds": 5
}
```

The `orc_client_user` and its password are in the `init_db.sql` file. It is highly recommended that you change the password to a more secure value. The same recommendation holds for the `vt_repl` user.

In production, you may also want to set "Debug" to false, and use a higher value for `RecoveryPeriodBlockSeconds` (default 3600).

You can bring `vtorc` using the following invocation:

```
vtorc <topo_flags> \
  -log_dir=${VTDATAROOT}/tmp \
  -config orc_config.json \
  -orc_web_dir ${VTROOT}/web/orchestrator
```

`orc_web_dir` must point at the contents of the orchestrator web files. The source can be found here.

Bringing up `vtorc` should immediately cause a primary to be elected among the vttablets that have come up.

The `vtorc` config supports a new `Durability` setting that can currently be set to `none`, `semi_sync` or `cross_cell`. The `semi_sync` setting is the equivalent to setting the vttablet's `enable_semi_sync` flag, whereas `cross_cell` will ensure that a primary will acknowledge a commit only if a `replica` that is not in the current cell has received the binary logs.

You can optionally add a `clusters_to_watch` flag that contains a comma separated list of keyspaces or `keyspace/shard` values. If specified, `vtorc` will manage only those clusters.

You can perform planned failovers using `vtorc`. Additionally, `vtorc` will also perform failure detection with automatic failovers while honoring the `Durability` rules.

Other Orchestrator settings may also be carefully added to the config. However, some of them may not be compatible with Vitess. These will be documented soon.

VTTablet and MySQL

Let us assume that we want to bring up a single unsharded keyspace. The first step is to identify the number of replicas (including the primary) we would like to deploy. We should also make a decision about how to distribute them across the cells.

Vitess requires you to assign a globally unique id (tablet UID) to every vttablet. This has to be an unsigned 32-bit integer. This is a legacy requirement derived from the fact that the MySQL server id (also an unsigned 32-bit integer) used to be the same as the tablet uid. This is not the case any more.

In terms of mapping these components to machines, Vitess allows you to run multiple of these on the same machine. If this is the case, you will need to assign non-conflicting ports for these servers to listen on.

VTTablet and MySQL are meant to be brought up as a pair within the same machine. By default, vttablet will connect to its MySQL over a unix socket.

Let us look at the steps to bring up the first pair for an unsharded keyspace `commerce` in cell1 and a tablet uid of 100.

Starting MySQL

`mysqlctl` is a convenience wrapper that can bring up and initialize a fresh MySQL server, and isolate all associated files within directories that are tied to the unique UID. This makes it easy to bring up multiple MySQL instances on the same machine.

The necessary arguments to a `mysqlctl` are the `tablet_uid` and `mysql_port`. Here is a sample invocation:

```
mysqlctl \
  -log_dir=${VTDATAROOT}/tmp \
  -tablet_uid=100 \
  -mysql_port=17100 \
  init
```

my.cnf `mysqlctl` **will not** read configuration files from common locations such as `/etc/my.cnf` or `/etc/mysql/my.cnf`. Instead, it will create a separate `my.cnf` config file using builtin defaults. The source files can be found here. To add your own settings, you can set the `EXTRA_MY_CNF` environment variable to a list of colon-separated files. Alternatively, you can override the default behavior by specifying your own template file using the `-mysqlctl_mycnf_template` command line argument.

For example, to override the default innodb buffer pool size, you would create a file named `/path/to/common.cnf` as follows:

```
innodb_buffer_pool_size=1G
```

And then launch mysqlctl with as follows:

```
EXTRA_MY_CNF"=/path/to/common."cnf mysqlctl \
  -log_dir=${VTDATAROOT}/tmp \
  -tablet_uid=100 \
  -mysql_port=17100 \
  init
```

When specifying additional configuration changes to Vitess, please keep in mind that changing the following settings is unsupported:

Setting	Reason
auto_commit	MySQL autocommit needs to be turned on. VTTablet uses connection pools to MySQL. If autocommit is turned off, MySQL will start an implicit transaction (with a point in time snapshot) for each connection and will work very hard at keeping the current view unchanged, which would be counter-productive.
log-bin	Several Vitess features rely on the binary log being enabled.
binlog-format	Vitess only supports row-based replication. Do not change this setting from the included configuration files.
binlog-row-image	Vitess only supports the default value (FULL)
log-slave-updates	Vitess requires this setting enabled, as it is in the included configuration files.
character-set*	Vitess only supports utf8 (and variants such as utf8mb4)
gtid-mode	Vitess relies on GTIDs to track changes to topology.
gtid-strict-mode/enforce-gtid-consistency	Vitess requires this setting to be unchanged.

Support was recently added to override sql_mode. However, we recommend keeping STRICT_TRANS_TABLES or replacing it withSTRICT_ALL_TABLES. Without one of these settings, MySQL could truncate values at the time of writing, and this can mismatch with decisions made by the sharding logic and lead to data corruption. VTTablet ensures that one of these values is set. If absolutely necessary, you can override this check by setting -enforce_strict_trans_tables=false while invoking vttablet.

init_db.sql After mysqld comes up, mysqlctl will initialize the server using an internal script, the contents of which can be found here. You can override this behavior by providing your own script using the -init_db_sql_file command line argument.

Disable AppArmor There is a common pitfall to watch out for: If you see an error like this in the mysqlctl logs, you may need to disable AppArmor:

```
I0429 01:16:25.648506       1 mysqld.go:454] Waiting for mysqld socket file
   (/vtdataroot/tabletdata/mysql.sock) to be ready...
I0429 01:16:25.656153       1 mysqld.go:399] Mysqld.Start(1588122985) stderr: mysqld:
   [ERROR] Could not open required defaults file: /vtdataroot/tabletdata/my.cnf
I0429 01:16:25.656180       1 mysqld.go:399] Mysqld.Start(1588122985) stderr: mysqld:
   [ERROR] Fatal error in defaults handling. Program aborted!
I0429 01:16:25.657249       1 mysqld.go:418] Mysqld.Start(1588122985) exit: exit status 1
```

Disabling AppArmor once may not be enough. Many software installs or upgrades automatically install it back. You may have to disable it again if this happens.

Verify MySQL You can verify that MySQL came up successfully by connecting to it from the command line client:

```
$ mysql -S ${VTDATAROOT}/vt_0000000100/mysql.sock -u vt_dba
[snip]
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| _vt                |
```

```
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

The MySQL instance that was brought up has no identity related to keyspace or shard at this moment. These will be assigned in subsequent steps.

mysqlctld `mysqlctld` is the server version of `mysqlctl`. If the target directories are empty when it is invoked, it automatically performs an `init`. The process can subsequently receive commands from vttablet to perform housekeeping operations like shutting down and restarting MySQL as needed.

To enable communication with vttablet, the server must be configured to receive grpc messages on a unix domain socket:

```
mysqlctld \
  -log_dir=${VTDATAROOT}/tmp \
  -tablet_uid=100 \
  -mysql_port=17100 \
  -socket_file=/path/to/socket_file
```

When starting vttablet, the following additional flag must be specified:

```
-mysqlctl_socket=/path/to/socket_file
```

Starting vttablet

VTTablet should be brought up on the same machine as the MySQL instance. It needs the following flags:

- and .
- `cell`: the cell in which vttablet is being brought up. Example: `cell1`.
- `tablet-path`: This should be the cell name followed by a `-` and the tablet UID used for `mysqlctl`. Example: `cell1-100`.
- `init_keyspace`: The keyspace that the tablet is going to serve. This will cause a keyspace to be created if one is not present.
- `init_shard`: The shard that the tablet is going to serve. This will cause a shard to be created if one is not present.
- `init_tablet_type`: This will typically be REPLICA. You may use other tablet types like "RDONLY". Those tablet types will be deprecated in favor of newer ways to achieve their functionality. Note that you are not allowed to start a tablet as a MASTER.
- `port`, `grpc_port`, and '-service_map' 'grpc-queryservice,grpc-tabletmanager"

There are some additional parameters that we recommend setting:

- `enable_semi_sync`: The recommended value for this is TRUE. You will need to bring up at least three vttablets for this setting to work correctly. This flag will be deprecated once `vtorc` takes over the management of MySQL instances.
- `enable_replication_reporter`: Enabling this flag will make vttablet send its replication lag information to the vtgates, and they will use this information to avoid sending queries to replicas that are lagged beyond a threshold.
- `unhealthy_threshold`: If `enable_replication_reporter` is enabled, and the replication lag exceeds this threshold, then vttablet stops serving queries. This value is meant to match the vtgate `discovery_high_replication_lag_minimum_serving` flag.
- `degraded_threshold`: This flag does not change vttablet's behavior. This threshold is used to report a warning in the status page if the replication lag exceeds this threshold. This value is meant to match the vtgate `discovery_low_replication_lag` flag.
- `restore_from_backup`: This flag informs vttablet to automatically restore data from the latest backup. Once this task completes, vttablet will point itself at the current primary to catch up on replication. When that falls below the specified threshold, vtgate will automatically start sending queries to the tablet.

- `queryserver-config-pool-size`:This value should be set to the max number of simultaneous queries you want MySQL to run. This should typically be around 2-3x the number of allocated CPUs. Around 4-16. There is not much harm in going higher with this value, but you may see no additional benefits. This pool gets used if the workload is set to `oltp`, which is the default.
- `queryserver-config-transaction-cap`: This value should be set to how many concurrent transactions you wish to allow. This should be a function of transaction rate and transaction length. Typical values are in the low 100s.
- `queryserver-config-stream-pool-size`: This value is relevant only if you plan to run streaming queries using the 'workload='olap"`setting. This value depends on how many simultaneous streaming queries you plan to run. Typical values are similar to`queryserver-config-pool-size'.
- `queryserver-config-query-timeout`: This value should be set to the upper limit you're willing to allow an OLTP query to run before it's deemed too expensive or detrimental to the rest of the system. VTTablet will kill any query that exceeds this timeout. This value is usually around 15-30s.
- `queryserver-config-transaction-timeout`: This value is meant to protect the situation where a client has crashed without completing a transaction. Typical value for this timeout is 30s.
- `queryserver-config-max-result-size`: This parameter prevents the OLTP application from accidentally requesting too many rows. If the result exceeds the specified number of rows, VTTablet returns an error. The default value is 10,000.

Here is a typical vttablet invocation:

```
vttablet <topo_flags> <backup_flags> \
  -log_dir=${VTDATAROOT}/tmp \
  -cell=cell1 \
  -tablet-path=cell1-100 \
  -init_keyspace=commerce \
  -init_shard=0 \
  -init_tablet_type=replica \
  -port=15100 \
  -grpc_port=16100 \
  -service_map grpc-queryservice,grpc-tabletmanager \
  -enable_semi_sync=true \
  -enable_replication_reporter=true \
  -restore_from_backup=true \
  -queryserver-config-pool-size=16 \
  -queryserver-config-transaction-cap=300 \
  -queryserver-config-stream-pool-size= 16
```

{{< info >}} It is important to set MySQL's `max_connections` property to be 50%-100% higher than the total number of connections in the various pools. This is because Vitess may have to kill connections and open new ones. MySQL accounting has a delay in how it counts closed connections, which may cause its view of the number of connections to exceed the ones currently opened by Vitess. For example, in the above example, the `max_connections` settings should be around 800. {{< /info >}}

It is normal to see errors like these in the log file until MySQL instances have been initialized and a vttablet has been elected as primary:

```
2020-04-27T00:38:02.040081Z 2 [Note] Aborted connection 2 to db: 'unconnected' user: 'root'
   host: 'localhost' (Got an error reading communication packets)
```

Starting the first vttablet against a keyspace and shard performs the following actions:

- Create a keyspace and shard in the global topo if these did not exist before.
- Perform a RebuildKeyspaceGraph to deploy the global topo to the current cell (cell1).
- Create a tablet record, which will allow vtgates to discover it.
- No restore action will be performed because this is the first time vttablet is coming up and no backups exist yet.

The vttablet will be unhealthy because the database for the keyspace has not been created. Visiting the `/debug/status` page on its port should show the following information:

The next step is to bring up the rest of the vttablet-MySQL pairs on other machines or different ports of the same machine.

Health

Current status: not serving: no replication status

Health Details

Current State	REPLICA: Not connected to mysql
Transition Error	Unknown database 'vt_commerce' (errno 1049) (sqlstate 42000)
Replication Lag	0s
Replication Error	no replication status

Health History

Time	Status	Tablet Type
Jan 2, 2021 at 12:31:04 (PST)	not serving: no replication status	replica

healthy
: serving traffic.

unhappy
: will serve traffic only if there are no fully healthy tablets.

unhealthy
: will not serve traffic.

Figure 11: unhealthy-tablet

Tablet Records

You can find out the current state of all vttablets with the following command:

```
$ vtctlclient ListAllTablets
cell1-0000000100 commerce 0 master sougou-lap1:15100 sougou-lap1:17100 []
   2021-01-02T22:27:11Z
cell1-0000000101 commerce 0 replica sougou-lap1:15101 sougou-lap1:17101 [] <null>
cell1-0000000102 commerce 0 rdonly sougou-lap1:15102 sougou-lap1:17102 [] <null>
```

This information is extracted from the "tablet record" in the cell specific topo. You can also browse to this information in vtctld either from the Dashboard or from the Topology tab.

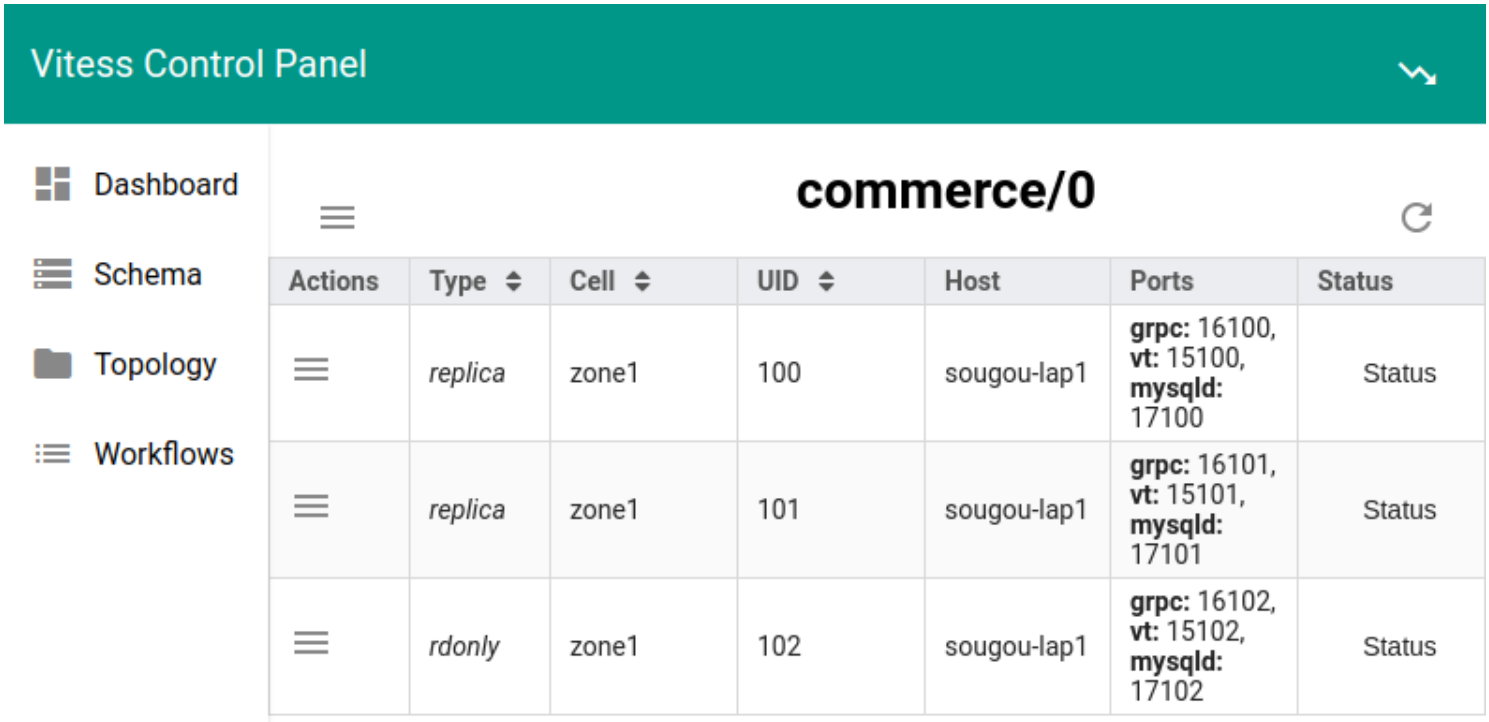


Figure 12: vtctld-tablet-list

You can move a vttablet-MySQL pair to a new host after shutting them down on the current host. Bringing up the new pair with the same UID will update the tablet record with the new address and ports. This will be noticed by the vtgates and they will adjust their traffic accordingly. However, you must not move a tablet to another cell.

Gracefully bringing down a vttablet will remove the address information from the tablet record thereby informing the vtgates that they should not attempt to send any more traffic to the tablets.

If a vttablet crashes, the address info will remain in the topo. However, vtgates will notice that the tablet is not reachable and will remember it as unhealthy. They will keep attempting to contact the tablet until it comes back up as healthy.

It is recommended that you delete the tablet record if you intend to bring down a vttablet permanently. The command to delete a tablet is:

```
vtctlclient DeleteTablet cell1-100
```

Legacy

description: User guides for features in older version of Vitess

Horizontal Sharding

{{< warning >}} In Vitess 6, Horizontal Sharding became obsolete with the introduction of Resharding! It is recommended to skip this guide, and continue on with the resharding user guide instead. {{< /warning >}}

{{< info >}} This guide follows on from Vertical Split and Get Started with a Local deployment. It assumes that several scripts have been executed, and that you have a running Vitess cluster. {{< /info >}}

The DBAs you hired with massive troves of hipster cash are pinging you on Slack and are freaking out. With the amount of data that you're loading up in your keyspaces, MySQL performance is starting to tank - it's okay, you're prepared for this! Although the query guardrails and connection pooling are cool features that Vitess can offer to a single unsharded keyspace, the real value comes into play with horizontal sharding.

Preparation

Before starting the resharding process, you need to make some decisions and prepare the system for horizontal resharding. Important note, this is something that should have been done before starting the vertical split. However, this is a good time to explain what normally would have been decided upon earlier the process.

Sequences The first issue to address is the fact that customer and corder have auto-increment columns. This scheme does not work well in a sharded setup. Instead, Vitess provides an equivalent feature through sequences.

The sequence table is an unsharded single row table that Vitess can use to generate monotonically increasing ids. The syntax to generate an id is: `select next :n values from customer_seq`. The vttablet that exposes this table is capable of serving a very large number of such ids because values are cached and served out of memory. The cache value is configurable.

The VSchema allows you to associate a column of a table with the sequence table. Once this is done, an insert on that table transparently fetches an id from the sequence table, fills in the value, and routes the row to the appropriate shard. This makes the construct backward compatible to how MySQL's `auto_increment` property works.

Since sequences are unsharded tables, they will be stored in the commerce database. The schema:

```
CREATE TABLE customer_seq (id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
INSERT INTO customer_seq (id, next_id, cache) VALUES (0, 1000, 100);
CREATE TABLE order_seq (id int, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
INSERT INTO order_seq (id, next_id, cache) VALUES (0, 1000, 100);
```

Note the `vitess_sequence` comment in the create table statement. VTTablet will use this metadata to treat this table as a sequence.

- `id` is always 0
- `next_id` is set to `1000`: the value should be comfortably greater than the `auto_increment` max value used so far.
- `cache` specifies the number of values to cache before vttablet updates `next_id`.

Larger cache values perform better, but will exhaust the values quicker since during reparent operations the new master will start off at the `next_id` value.

The VTGate servers also need to know about the sequence tables. This is done by updating the VSchema for commerce as follows:

```
{
  "tables": {
    "customer_seq": {
      "type": "sequence"
    },
    "order_seq": {
      "type": "sequence"
    },
    "product": {}
  }
}
```

Vindexes The next decision is about the sharding keys, aka Primary Vindexes. This is a complex decision that involves the following considerations:

- What are the highest QPS queries, and what are the where clauses for them?
- Cardinality of the column; it must be high.
- Do we want some rows to live together to support in-shard joins?
- Do we want certain rows that will be in the same transaction to live together?

Using the above considerations, in our use case, we can determine that:

- For the customer table, the most common where clause uses `customer_id`. So, it shall have a Primary Vindex.
- Given that it has lots of users, its cardinality is also high.
- For the corder table, we have a choice between `customer_id` and `order_id`. Given that our app joins `customer` with `corder` quite often on the `customer_id` column, it will be beneficial to choose `customer_id` as the Primary Vindex for the `corder` table as well.
- Coincidentally, transactions also update `corder` tables with their corresponding `customer` rows. This further reinforces the decision to use `customer_id` as Primary Vindex.

NOTE: It may be worth creating a secondary lookup Vindex on `corder.order_id`. This is not part of the example. We will discuss this in the advanced section.

NOTE: For some use cases, `customer_id` may actually map to a tenant_id. In such cases, the cardinality of a tenant id may be too low. It's also common that such systems have queries that use other high cardinality columns in their where clauses. Those should then be taken into consideration when deciding on a good Primary Vindex.

Putting it all together, we have the following VSchema for `customer`:

```
{
  "sharded": true,
  "vindexes": {
    "hash": {
      "type": "hash"
    }
  },
  "tables": {
    "customer": {
      "column_vindexes": [
        {
          "column": "customer_id",
          "name": "hash"
        }
      ],
      "auto_increment": {
        "column": "customer_id",
```

```
        "sequence": "customer_seq"
      }
    },
    "corder": {
      "column_vindexes": [
        {
          "column": "customer_id",
          "name": "hash"
        }
      ],
      "auto_increment": {
        "column": "order_id",
        "sequence": "order_seq"
      }
    }
  }
}
```

Note that we have now marked the keyspace as sharded. Making this change will also change how Vitess treats this keyspace. Some complex queries that previously worked may not work anymore. This is a good time to conduct thorough testing to ensure that all the queries work. If any queries fail, you can temporarily revert the keyspace as unsharded. You can go back and forth until you have got all the queries working again.

Since the primary vindex columns are `BIGINT`, we choose `hash` as the primary vindex, which is a pseudo-random way of distributing rows into various shards.

NOTE: For `VARCHAR` columns, use `unicode_loose_md5` or `unicode_loose_xxhash`. For `VARBINARY`, use `binary_md5` or `xxhash`.

NOTE: All vindexes in Vitess are plugins. If none of the predefined vindexes suit your needs, you can develop your own custom vindex.

Now that we have made all the important decisions, it's time to apply these changes:

```
./301_customer_sharded.sh
```

Create new shards

At this point, you have finalized your sharded VSchema and vetted all the queries to make sure they still work. Now, it's time to reshard.

The resharding process works by splitting existing shards into smaller shards. This type of resharding is the most appropriate for Vitess. There are some use cases where you may want to spin up a new shard and add new rows in the most recently created shard. This can be achieved in Vitess by splitting a shard in such a way that no rows end up in the 'new' shard. However, it's not natural for Vitess.

We have to create the new target shards:

```
./302_new_shards.sh
```

Shard 0 was already there. We have now added shards `-80` and `80-`. We've also added the `CopySchema` directive which requests that the schema from shard 0 be copied into the new shards.

Shard naming What is the meaning of `-80` and `80-`? The shard names have the following characteristics:

- They represent a range, where the left number is included, but the right is not.
- Their notation is hexadecimal.
- They are left justified.
- A `-` prefix means: anything less than the RHS value.
- A `-` postfix means: anything greater than or equal to the LHS value.

- A plain `-` denotes the full keyrange.

What does this mean: `-80` == `00-80` == `0000-8000` == `000000-800000`

`80-` is not the same as `80-FF`. This is why:

`80-FF` == `8000-FF00`. Therefore `FFFF` will be out of the `80-FF` range.

`80-` means: 'anything greater than or equal to `0x80`

A `hash` vindex produces an 8-byte number. This means that all numbers less than `0x8000000000000000` will fall in shard `-80`. Any number with the highest bit set will be >= `0x8000000000000000`, and will therefore belong to shard `80-`.

This left-justified approach allows you to have keyspace ids of arbitrary length. However, the most significant bits are the ones on the left.

For example an `md5` hash produces 16 bytes. That can also be used as a keyspace id.

A `varbinary` of arbitrary length can also be mapped as is to a keyspace id. This is what the `binary` vindex does.

In the above case, we are essentially creating two shards: any keyspace id that does not have its leftmost bit set will go to `-80`. All others will go to `80-`.

Applying the above change should result in the creation of six more vttablet instances.

At this point, the tables have been created in the new shards but have no data yet.

```
mysql --table < ../common/select_customer-80_data.sql
Using customer/-80
Customer
COrder
mysql --table < ../common/select_customer80-_data.sql
Using customer/80-
Customer
COrder
```

SplitClone

The process for SplitClone is similar to VerticalSplitClone. It starts the horizontal resharding process:

```
./303_horizontal_split.sh
```

This starts the following job "SplitClone -min_healthy_rdonly_tablets=1 customer/0":

For large tables, this job could potentially run for many days, and can be restarted if failed. This job performs the following tasks:

- Dirty copy data from customer/0 into the two new shards. But rows are split based on their target shards.
- Stop replication on customer/0 rdonly tablet and perform a final sync.
- Start a filtered replication process from customer/0 into the two shards by sending changes to one or the other shard depending on which shard the rows belong to.

Once `SplitClone` has completed, you should see this:

The horizontal counterpart to `VerticalSplitDiff` is `SplitDiff`. It can be used to validate the data integrity of the resharding process "SplitDiff -min_healthy_rdonly_tablets=1 customer/-80":

NOTE: This example does not actually run this command.

Note that the last argument of SplitDiff is the target (smaller) shard. You will need to run one job for each target shard. Also, you cannot run them in parallel because they need to take an `rdonly` instance offline to perform the comparison.

NOTE: SplitDiff can be used to split shards as well as to merge them.

Cut over

Now that you have verified that the tables are being continuously updated from the source shard, you can cutover the traffic. This is typically performed in three steps: `rdonly`, `replica` and `master`:

For rdonly and replica:

```
./304_migrate_replicas.sh
```

For master:

```
./305_migrate_master.sh
```

During the *master* migration, the original shard master will first stop accepting updates. Then the process will wait for the new shard masters to fully catch up on filtered replication before allowing them to begin serving. Since filtered replication has been following along with live updates, there should only be a few seconds of master unavailability.

The replica and rdonly cutovers are freely reversible. Unlike the Vertical Split, a horizontal split is also reversible. You just have to add a `-reverse_replication` flag while cutting over the master. This flag causes the entire resharding process to run in the opposite direction, allowing you to Migrate in the other direction if the need arises.

You should now be able to see the data that has been copied over to the new shards.

```
mysql --table < ../common/select_customer-80_data.sql
Using customer/-80
Customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           5 | eve@domain.com     |
+-------------+--------------------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+

mysql --table < ../common/select_customer80-_data.sql
Using customer/80-
Customer
+-------------+----------------+
| customer_id | email          |
+-------------+----------------+
|           4 | dan@domain.com |
+-------------+----------------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        4 |           4 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Clean up

After celebrating your second successful resharding, you are now ready to clean up the leftover artifacts:

```
./306_down_shard_0.sh
```

In this script, we just stopped all tablet instances for shard 0. This will cause all those vttablet and `mysqld` processes to be stopped. But the shard metadata is still present. We can clean that up with this command (after all vttablets have been brought down):

```
./307_delete_shard_0.sh
```

This command runs the following "`DeleteShard -recursive customer/0`".

Beyond this, you will also need to manually delete the disk associated with this shard.

Next Steps

Feel free to experiment with your Vitess cluster! Execute the following when you are ready to teardown your example:

```
./401_teardown.sh
```

Vertical Split

{{< warning >}} In Vitess 6, Vertical Split became obsolete with the introduction of MoveTables! It is recommended to skip this guide, and continue on with the MoveTables user guide instead. If you continue please note that all scripts referenced are now contained in a different repo called 'legacy_local'. {{< /warning >}}

{{< info >}} This guide follows on from get started with a local deployment. It assumes that the `./101_initial_cluster.sh` script has been executed, and that you have a running Vitess cluster. {{< /info >}}

Vertical Split enables you to move a subset of tables to their own keyspace. Continuing on from the ecommerce example started in the get started guide, as your database continues to grow, you may decide to separate the `customer` and `corder` tables from the `product` table. Let us add some data into our tables to illustrate how the vertical split works. Paste the following:

```
mysql < ../common/insert_commerce_data.sql
```

We can look at what we just inserted:

```
mysql --table < ../common/select_commerce_data.sql
Using commerce/0
Customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           4 | dan@domain.com     |
|           5 | eve@domain.com     |
+-------------+--------------------+
Product
+----------+-------------+-------+
| sku      | description | price |
+----------+-------------+-------+
| SKU-1001 | Monitor     |   100 |
| SKU-1002 | Keyboard    |    30 |
+----------+-------------+-------+
COrder
```

```
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        4 |           4 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Notice that we are using keyspace `commerce/0` to select data from our tables.

Create Keyspace

For a vertical split, we first need to create a special `served_from` keyspace. This keyspace starts off as an alias for the `commerce` keyspace. Any queries sent to this keyspace will be redirected to `commerce`. Once this is created, we can vertically split tables into the new keyspace without having to make the app aware of this change:

```
./201_customer_keyspace.sh
```

This creates an entry into the topology indicating that any requests to master, replica, or rdonly sent to `customer` must be redirected to (served from) `commerce`. These tablet type specific redirects will be used to control how we transition the cutover from `commerce` to `customer`.

Customer Tablets

Now you have to create vttablet instances to back this new keyspace onto which you'll move the necessary tables:

```
./202_customer_tablets.sh
```

The most significant change, this script makes is the instantiation of vttablets for the new keyspace. Additionally:

- You moved customer and corder from the commerce's VSchema to customer's VSchema. Note that the physical tables are still in commerce.
- You requested that the schema for customer and corder be copied to customer using the `copySchema` directive.

The move in the VSchema should not make a difference yet because any queries sent to customer are still redirected to commerce, where all the data is still present.

VerticalSplitClone

The next step:

```
./203_vertical_split.sh
```

starts the process of migrating the data from commerce to customer.

For large tables, this job could potentially run for many days, and may be restarted if failed. This job performs the following tasks:

- Dirty copy data from commerce's customer and corder tables to customer's tables.
- Stop replication on commerce's rdonly tablet and perform a final sync.
- Start a filtered replication process from commerce->customer that keeps the customer's tables in sync with those in commerce.

NOTE: In production, you would want to run multiple sanity checks on the replication by running `SplitDiff` jobs multiple times before starting the cutover.

We can look at the results of VerticalSplitClone by examining the data in the customer keyspace. Notice that all data in the `customer` and `corder` tables has been copied over.

```
mysql --table < ../common/select_customer0_data.sql
Using customer/0
Customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           4 | dan@domain.com     |
|           5 | eve@domain.com     |
+-------------+--------------------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        4 |           4 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Cut over

Once you have verified that the `customer` and `corder` tables are being continuously updated from commerce, you can cutover the traffic. This is typically performed in three steps: `rdonly`, `replica` and `master`:

For rdonly and replica:

```
./204_vertical_migrate_replicas.sh
```

For master:

```
./205_vertical_migrate_master.sh
```

Once this is done, the `customer` and `corder` tables are no longer accessible in the `commerce` keyspace. You can verify this by trying to read from them.

```
mysql --table < ../common/select_commerce_data.sql
Using commerce/0
Customer
ERROR 1105 (HY000) at line 4: vtgate: http://vtgate-zone1-5ff9c47db6-7rmld:15001/: target:
   commerce.0.master, used tablet: zone1-1564760600 (zone1-commerce-0-replica-0.vttablet),
   vttablet: rpc error: code = FailedPrecondition desc = disallowed due to rule: enforce
   blacklisted tables (CallerID: userData1)
```

The replica and rdonly cutovers are freely reversible. However, the master cutover is one-way and cannot be reversed. This is a limitation of vertical resharding, which will be resolved in the near future. For now, care should be taken so that no loss of data or availability occurs after the cutover completes.

Clean up

After celebrating your first successful 'vertical resharding', you will need to clean up the leftover artifacts:

```
./206_clean_commerce.sh
```

Those tables are now being served from customer. So, they can be dropped from commerce.

The 'control' records were added by the `MigrateServedFrom` command during the cutover to prevent the commerce tables from accidentally accepting writes. They can now be removed.

After this step, the `customer` and `corder` tables no longer exist in the `commerce` keyspace.

```
mysql --table < ../common/select_commerce_data.sql
Using commerce/0
Customer
ERROR 1105 (HY000) at line 4: vtgate: http://vtgate-zone1-5ff9c47db6-7rmld:15001/: target:
   commerce.0.master, used tablet: zone1-1564760600 (zone1-commerce-0-replica-0.vttablet),
   vttablet: rpc error: code = InvalidArgument desc = table customer not found in schema
   (CallerID: userData1)
```

Next Steps

You can now proceed with Horizontal Sharding.

Or alternatively, if you would like to teardown your example:

```
./401_teardown.sh
```

Migration

description: User guides covering migration to Vitess

Materialize

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have an Operator, local or Helm installation ready. Make sure you have only run the "101" step of the examples, for example `101_initial_cluster.sh` in the local example. The commands in this guide also assumes you have setup the shell aliases from the example, e.g. `env.sh` in the local example. {{< /info >}}

Materialize is a new VReplication workflow in Vitess 6. It can be used as a more general way to achieve something similar to MoveTables, or as a way to generate materialized views of a table (or set of tables) in the same or different keyspace from the source table (or set of tables). In general, it can be used to create and maintain continually updated materialized views in Vitess, without having to resort to manual or trigger-based population of the view content.

Since `Materialize` uses VReplication, the view can be kept up-to-date very close to real-time, which enables use-cases like creating copies of the same table sharded different ways for the purposes of certain types of queries that would otherwise be prohibitively expensive on the original table. `Materialize` is also flexible enough to allow for you to pre-create the schema and vschema for the copied table, allowing you to, for example, maintain a copy of a table without some of the source table's MySQL indexes. Alternatively, you could use `Materialize` to do certain schema changes (e.g. change the type of a table column) without having to use other tools like gh-ost.

All of the command options and parameters are listed in our reference page for Materialize.

In our example, we will be using `Materialize` to perform something similar to the MoveTables user guide, which will cover just the basics of what is possible using `Materialize`.

Let's start by simulating this situation by loading sample data:

```
# On helm and local installs:
mysql < /usr/local/vitess/examples/common/insert_commerce_data.sql
# With operator:
mysql --table < insert_commerce_data.sql
```

We can look at what we just inserted:

```
# On helm and local installs:
mysql --table < /usr/local/vitess/examples/common/select_commerce_data.sql
# With operator:
mysql --table < select_commerce_data.sql

Using commerce
Customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           4 | dan@domain.com     |
|           5 | eve@domain.com     |
+-------------+--------------------+
Product
+----------+-------------+-------+
| sku      | description | price |
+----------+-------------+-------+
| SKU-1001 | Monitor     |   100 |
| SKU-1002 | Keyboard    |    30 |
+----------+-------------+-------+
COrder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        4 |           4 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Note that we are using keyspace `commerce` to select data from our tables.

Planning to use Materialize

In this scenario, we are going to make two copies of the `corder` table **in the same keyspace** using a different tablenames of `corder_view` and `corder_view_redacted`. The first copy will be identical to the source table, but for the `corder_view_redacted` copy, we will use the opportunity to drop the `price` column from the copy. Since we are doing the `Materialize` to the same keyspace, we do not need to create a new keyspace or tablets as we did for the MoveTables user guide.

Create the destination tables

In the case where we using `Materialize` to copy tables between keyspaces, we can use the `"create_ddl": "copy"` option in the `Materialize json_spec table_settings` to create the target table for us (similar to what `MoveTables` does). However, in our case where we are using `Materialize` with a target table name different from the source table name, we need to manually create the target tables. Let's go ahead and do that:

```
$ mysql -A
Welcome to the MySQL monitor.  Commands end with ; or \g.
.
.

mysql> CREATE TABLE `corder_view` (
  `order_id` bigint NOT NULL,
  `customer_id` bigint DEFAULT NULL,
  `sku` varbinary(128) DEFAULT NULL,
  `price` bigint DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE `corder_view_redacted` (
  `order_id` bigint NOT NULL,
  `customer_id` bigint DEFAULT NULL,
  `sku` varbinary(128) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB;
Query OK, 0 rows affected (0.09 sec)
```

Now we need to make sure Vitess' view of our schema is up-to-date:

```
$ vtctlclient ReloadSchemaKeyspace commerce
```

And now we can proceed to the `Materialize` step(s).

Start the Materialize (first copy)

We will run two `Materialize` operations, one for each copy/view of the `corder` table we will be creating. We could combine these two operations into a single `Materialize` operation, but we will keep them separate for clarity.

```
$ vtctlclient Materialize '{"workflow": "copy_corder_1", "source_keyspace": "commerce",
   "target_keyspace": "commerce", "table_settings": [{"target_table": "corder_view",
   "source_expression": "select * from corder"}]}'
```

Now, we should see the materialized view table `corder_view`:

```
$ echo "select * from corder_view;" | mysql --table commerce
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        4 |           4 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

And if we insert a row into the source table, it will be replicated to the materialized view:

```
$ echo "insert into corder (order_id, customer_id, sku, price) values (6, 6, 'SKU-1002',
   30);" | mysql commerce
$ echo "select * from corder_view;" | mysql --table commerce
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
```

```
|         1 |           1 | SKU-1001 |   100 |
|         2 |           2 | SKU-1002 |    30 |
|         3 |           3 | SKU-1002 |    30 |
|         4 |           4 | SKU-1002 |    30 |
|         5 |           5 | SKU-1002 |    30 |
|         6 |           6 | SKU-1002 |    30 |
+-----------+-------------+----------+-------+
```

Note that the target table is just a normal table, there is nothing that prevents you from writing to it directly. While you might not want to do that in this "view" use-case, in certain other use-cases, it might be completely acceptable to write to the table, as long as you don't end up altering or removing rows in a fashion that would break the "replication" part of VReplication (e.g. removing a row in the target table directly that is later updated in the source table).

Viewing the workflow while in progress

While we can also see and manipulate the underlying VReplication streams created by `Materialize`; there are commands to show, stop, start and delete the operations associated with a Materialize workflow. For example, once we have started the `Materialize` command above, we can observe the status of the VReplication stream doing the materialization via the `vtctlclient Workflow` command:

```
$ vtctlclient Workflow commerce.copy_corder_1 show
{
    "Workflow": "copy_corder_1",
    "SourceLocation": {
        "Keyspace": "commerce",
        "Shards": [
            "0"
        ]
    },
    "TargetLocation": {
        "Keyspace": "commerce",
        "Shards": [
            "0"
        ]
    },
    "MaxVReplicationLag": 1599019410,
    "ShardStatuses": {
        "0/zone1-0000000100": {
            "MasterReplicationStatuses": [
                {
                    "Shard": "0",
                    "Tablet": "zone1-0000000100",
                    "ID": 4,
                    "Bls": {
                        "keyspace": "commerce",
                        "shard": "0",
                        "filter": {
                            "rules": [
                                {
                                    "match": "corder_view",
                                    "filter": "select * from corder"
                                }
                            ]
                        }
                    },
                    "Pos": "MySQL56/c0d82d27-ecd0-11ea-9840-001e677affd5:1-97978",
                    "StopPos": "",
```

```
                    "State": "Copying",
                    "MaxReplicationLag": 9223372036854775807,
                    "DBName": "vt_commerce",
                    "TransactionTimestamp": 0,
                    "TimeUpdated": 1599019408,
                    "Message": "",
                    "CopyState": [
                        {
                            "Table": "corder_view",
                            "LastPK": "fields:<name:\"order_id\" type:INT64 >
                               rows:<lengths:5 values:\"37014\" >"
                        }
                    ]
                }
            ],
            "TabletControls": null,
            "MasterIsServing": true
        }
    }
}
```

Note the state of `Copying`, this will transition to `Running` when the bulk copying of rows is complete.

We can now also use the stop/start commands to temporarily stop the materialization workflow. E.g. `stop`:

```
$ vtctlclient Workflow commerce.copy_corder_1 stop
+------------------+--------------+
|      Tablet      | RowsAffected |
+------------------+--------------+
| zone1-0000000100 |            1 |
+------------------+--------------+
```

And `start` to start the workflow again and continue the materialization:

```
$ vtctlclient Workflow commerce.copy_corder_1 start
+------------------+--------------+
|      Tablet      | RowsAffected |
+------------------+--------------+
| zone1-0000000100 |            1 |
+------------------+--------------+
```

Eventually, when the copy is done, or we have materialized the data, and do not want to continue the copy of new source rows, we can delete the workflow via:

```
$ vtctlclient Workflow commerce.copy_corder_1 delete
+------------------+--------------+
|      Tablet      | RowsAffected |
+------------------+--------------+
| zone1-0000000100 |            1 |
+------------------+--------------+
```

Note that deleting the workflow will not drop the target table for the `Materialize` workflow, or any of the data already copied. The data in the target table will remain as it was at the moment the workflow was deleted (or previously stopped).

Start the Materialize (redacted copy)

Now, we can perform the copy to the `corder_view_redacted` table we created earlier. Note that we created this table without a price column; we will not be copying that column.

```
$ vtctlclient Materialize '{"workflow": "copy_corder_2", "source_keyspace": "commerce",
   "target_keyspace": "commerce", "table_settings": [{"target_table":
   "corder_view_redacted", "source_expression": "select order_id, customer_id, sku from
   corder"}]}'
```

Again, looking the target table will show all the source table rows, this time without the `sku` column:

```
$ echo "select * from corder_view_redacted;" | mysql --table commerce
+----------+-------------+----------+
| order_id | customer_id | sku      |
+----------+-------------+----------+
|        1 |           1 | SKU-1001 |
|        2 |           2 | SKU-1002 |
|        3 |           3 | SKU-1002 |
|        4 |           4 | SKU-1002 |
|        5 |           5 | SKU-1002 |
|        6 |           6 | SKU-1002 |
+----------+-------------+----------+
```

Again, we can add a row to the source table, and see it replicated into the target table:

```
$ echo "insert into corder (order_id, customer_id, sku, price) values (7, 7, 'SKU-1002',
   30);" | mysql commerce
$ echo "select * from corder_view_redacted;" | mysql --table commerce
+----------+-------------+----------+
| order_id | customer_id | sku      |
+----------+-------------+----------+
|        1 |           1 | SKU-1001 |
|        2 |           2 | SKU-1002 |
|        3 |           3 | SKU-1002 |
|        4 |           4 | SKU-1002 |
|        5 |           5 | SKU-1002 |
|        6 |           6 | SKU-1002 |
|        7 |           7 | SKU-1002 |
+----------+-------------+----------+
```

What happened under the covers

As with `MoveTables`, a VReplication stream was formed for each of the `Materialize` workflows we executed. We can see these by inspecting the VReplication table on the target keyspace master tablet, e.g. in this case:

```
$ vtctlclient VReplicationExec zone1-0000000100 'select * from _vt.vreplication'
+----+---------------+-------------------------------------------------+-----------------------------
| id |   workflow    |                  source                         |
   pos                          | stop_pos |       max_tps       | max_replication_lag |
   cell | tablet_types | time_updated | transaction_timestamp |  state  | message |
   db_name   |
+----+---------------+-------------------------------------------------+-----------------------------
|  1 | copy_corder_1 | keyspace:"commerce" shard:"0"                   |
   MySQL56/00a04e3a-e74d-11ea-a8c9-001e677affd5:1-926 |          | 9223372036854775807 |
   9223372036854775807 |      |              |   1598416592 |            1598416591 |
   Running |         | vt_commerce |
|    |               | filter:<rules:<match:"corder_view"              |
                                                  |          |                       |
                        |      |              |              |                       |
           |         |             |
```

```
|    |               | filter:"select * from corder" > >           |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
|  2 | copy_corder_2 | keyspace:"commerce" shard:"0"               |
   MySQL56/00a04e3a-e74d-11ea-a8c9-001e677affd5:1-926 |          | 9223372036854775807 |
   9223372036854775807 |      |              |   1598416592 |            1598416591 |
   Running |         | vt_commerce |
|    |               | filter:<rules:<match:"corder_view_redacted" |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
|    |               | filter:"select order_id, customer_id, sku   |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
|    |               | from corder" > >                            |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
+----+---------------+---------------------------------------------+-----------------------------
```

It is important to use the `vtctlclient VReplicationExec` command to inspect this table, since some of the fields are binary and might not render properly in a MySQL client (at least with default options). In the above output, you can see a summary of the VReplication streams that were setup (and are still `Running`) to copy and then do continuous replication of the source table (`corder`) to the two different target tables.

Cleanup

As seen earlier, you can easily use the `vtctlclient Workflow ... stop` and `vtctlclient Workflow ... delete` commands to clean up a materialize operation. If you like, you can instead use the `VReplicationExec` command to temporarily stop the VReplication streams that make up the `Materialize` process. For example, to stop both streams, you can do:

```
$ vtctlclient VReplicationExec zone1-0000000100 'update _vt.vreplication set state =
   "Stopped" where id in (1,2)'
+
+
$ vtctlclient VReplicationExec zone1-0000000100 'select * from _vt.vreplication'
+----+---------------+---------------------------------------------+-----------------------------
| id |   workflow    |                  source                     |
                           pos                          | stop_pos |       max_tps       |
   max_replication_lag | cell | tablet_types | time_updated | transaction_timestamp |
   state  | message |   db_name   |
+----+---------------+---------------------------------------------+-----------------------------
|  1 | copy_corder_1 | keyspace:"commerce" shard:"0"               |
   MySQL56/00a04e3a-e74d-11ea-a8c9-001e677affd5:1-1218 |          | 9223372036854775807 |
   9223372036854775807 |      |              |   1598416861 |            1598416859 |
   Stopped |         | vt_commerce |
|    |               | filter:<rules:<match:"corder_view"          |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
|    |               | filter:"select * from corder" > >           |
                                                 |          |                      |
                       |      |              |              |                      |
           |         |            |
```

```
|  2 | copy_corder_2 | keyspace:"commerce" shard:"0"                |
   MySQL56/00a04e3a-e74d-11ea-a8c9-001e677affd5:1-1218 |          | 9223372036854775807 |
   9223372036854775807 |      |              |   1598416861 |            1598416859 |
   Stopped |          | vt_commerce |
|    |               | filter:<rules:<match:"corder_view_redacted" |
                                                  |          |                      |
                        |      |              |              |                      |
           |          |             |
|    |               | filter:"select order_id, customer_id, sku   |
                                                  |          |                      |
                        |      |              |              |                      |
           |          |             |
|    |               | from corder" > >                             |
                                                  |          |                      |
                        |      |              |              |                      |
           |          |             |
+----+---------------+-------------------------------------------------+-----------------------------
```

Any changes to the source tables will now not be applied to the target tables until you update the `state` column back to `Running`.

Lastly, you can clean up the `Materialize` process by just using `VReplicationExec` to delete the rows in the `_vt.vreplication` table. This will do the necessary runtime cleanup as well. E.g.:

```
$ vtctlclient VReplicationExec zone1-0000000100 'delete from  _vt.vreplication where id in
   (1,2)'
+
+
$ vtctlclient VReplicationExec zone1-0000000100 'select * from _vt.vreplication'
+----+----------+--------+-----+----------+---------+---------------------+------+--------------+-
| id | workflow | source | pos | stop_pos | max_tps | max_replication_lag | cell |
   tablet_types | time_updated | transaction_timestamp | state | message | db_name |
+----+----------+--------+-----+----------+---------+---------------------+------+--------------+-
+----+----------+--------+-----+----------+---------+---------------------+------+--------------+-
```

Note that this just cleans up the VReplication streams; the actual source and target tables are left untouched and in the same state they were at the moment the VReplication streams were stopped or deleted.

Recap

As mentioned at the beginning, `Materialize` gives you finer control over the VReplication process without having to form VReplication rules by hand. For the ultimate flexibility, that is still possible, but you should be able to use `Materialize` together with other Vitess features like routing rules to cover a large set of potential migration and data maintenance use-cases without resorting to creating VReplication rules directly.

Migrating data into Vitess

Introduction

There are two main parts to migrating your data to Vitess: migrating the actual data and repointing the application. The answer here will focus primarily on the methods that can be used to migrate your data into Vitess.

Overview

There are three different methods to migrate your data into Vitess. Choosing the appropriate option depends on several factors. 1. The nature of the application accessing the MySQL database 1. The size of the MySQL database to be migrated 1. The load,

especially the write load, on the MySQL database 1. Your tolerance for downtime during the migration of data 1. Whether you require the ability to reverse the migration if needed 1. The network level configuration of your components

The three different methods are:

- 'Stop-the-world'
- VReplication from Vitess setup in front of the existing external MySQL database
- Application-level migration

Method 1: "Stop-the-world":

The simplest method to migrate data is to do a 'dump and restore' or 'stop-the-world'. We recommend using 'go-mydumper'. To execute this method you would follow these steps: 1. Stop writing to the source MySQL database 1. Take a logical dump of the database using go-mydumper or possibly mysqldump 1. Apply some simple transformations on the output 1. Import the data into Vitess via the frontend 1. Repoint your application to the new database
1. Resume writing to the new database in Vitess

This method is only suitable for migrating small or non-critical databases that can tolerate downtime. The database will be unavailable for writes between the time the dump is started and the time the restore of the dump is completed. For databases of 10's of GB and up this process could take hours or even days. The amount of downtime scales with the amount of data being migrated.

Please note the 'dump and restore' method likely isn't viable for most production applications, unless the applicable downtime can be handled.

Method 2: VReplication from Vitess setup in front of the existing external MySQL database

A set of Vitess components will be created, on a temporary basis, to run in front of the source MySQL database in order to migrate the data. These components should reference at least one of the replicas, in addition to the master, of the MySQL database. The Vitess components can be run on bare metal, in a VM, or potentially even in Kubernetes.

It is important to note that the Vitess components must be reachable over a network by Vitess's backend systems. Your topology must be set up such that the source database is reachable from your vitess cluster. Similarly, all the VTTablets being configured for migration must be set up to run against your database within the same Vitess cluster.

Vitess offers a choice of two VReplication commands to perform the data migration process described above: *MoveTables* or *Materialize*.

Both methods use a combination of transactional SELECTs and filtered MySQL replication to copy each of the tables in the source database to Vitess. Once all the data is copied, the two databases are kept in sync using the replication stream from the source database. While in this synchronized state, you can verify the source and destination are in sync, and testing on the copy of the data in Vitess can commence.

Once the testing has completed, application traffic can be removed from the source MySQL database and switched to the Vitess database. For this switch, a small amount of downtime will be necessary. This downtime could be seconds or minutes, depending on the application and application automation.

There are some differences between *MoveTables* and *Materialize* that you will need to evaluate to determine which process to use:

Materialize: This process works well if you want to get data out as purely a copy or you want to transform the data during the copying process

- Has more flexibility because you can transform the data while you are migrating it. E.g. you can choose not to migrate specific columns from a table
- It isn't directly reversible. E.g. changes to the downstream Vitess copy of the data after the application cutover will not flow back to the original source MySQL database
- Switching over application traffic is not integrated. You have to manually configure the commands in order to do the switch over

MoveTables: This process works well if you want to have minimum downtime during the switch over and to be able to reverse the switch over

- Switch reads and switch writes are integrated
- Allows the switch over to be reversible due to reverse replication. Writes to Vitess can be propagated back to the source MySQL database after the copy
- Cannot transform the data during the migration. The assumption is that the entire dataset is being copied as is

Choosing the Right Method The first and most important point to consider when choosing the right method is whether you can or cannot interconnect between components on your network. If you cannot, or do not wish to, perform extra steps to ensure interconnectivity then you will need to use the 'Stop-the-world' method.

If you can ensure interconnectivity and that the target VTTablets are in the same Vitess cluster, then for cases when larger amounts of downtime are not an option you will want to use VReplication with either *MoveTables* or *Materialize*.

Method 3: Application-level migration

In some cases it might be necessary to perform the data migration on an application level. Reasons for this might be things like:

- The source data is spread across a large set of MySQL databases, and is being consolidated as part of the migration process. Thus it's not possible to migrate data using only normal MySQL replication
- The source database systems are not running MySQL Row-Based Replication and it's not possible, feasible, or practical to convert them
- The source database system might not be MySQL, in which case a custom application-level migration will be necessary

In these cases custom tools must first be written on the application side to start writing data to both the legacy database and Vitess. Secondly, the source data must be moved over in bulk to the Vitess database and then the switch over can be performed.

There are multiple options to do those steps, however we won't go into detail as each situation for these cases is unique. A summary of some potential options are:

"Stop the world":

- Write application-level tools to export, import, and verify data between the source and destination systems.

Dual writes:

- Modify the application to start doing dual writes between the source and destination databases, while the application is still pointing to the source database as the primary datastore.
- Create custom tools to backfill old data from the source to destination system. VReplication could be used to form a part of this solution.
- Cut-over by having the application start to read, as well as write, from the destination Vitess database as the primary data source. This option can be reversible, assuming the dual writes continue after the read cutover.

MoveTables

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have an Operator, local or Helm installation ready after the `101_initial_cluster` step, and making sure you have setup aliases and port-forwarding (if necessary). {{< /info >}}

MoveTables is a new VReplication workflow in Vitess 6 and later, and obsoletes Vertical Split from earlier releases.

This feature enables you to move a subset of tables between keyspaces without downtime. For example, after Initially deploying Vitess, your single commerce schema may grow so large that it needs to be split into multiple keyspaces.

All of the command options and parameters are listed in our reference page for MoveTables.

As a stepping stone towards splitting a single table across multiple servers (sharding), it usually makes sense to first split from having a single monolithic keyspace (`commerce`) to having multiple keyspaces (`commerce` and `customer`). For example, in our hypothetical ecommerce system we may know that `customer` and `corder` tables are closely related and both growing quickly.

Let's start by simulating this situation by loading sample data:

```
# On helm, local and operator installs:
mysql --table < ../common/insert_commerce_data.sql
```

We can look at what we just inserted:

```
# On helm, local and operator installs:
mysql --table < ../common/select_commerce_data.sql

Using commerce/0
customer
+-------------+--------------------+
| customer_id | email              |
+-------------+--------------------+
|           1 | alice@domain.com   |
|           2 | bob@domain.com     |
|           3 | charlie@domain.com |
|           4 | dan@domain.com     |
|           5 | eve@domain.com     |
+-------------+--------------------+
product
+----------+-------------+-------+
| sku      | description | price |
+----------+-------------+-------+
| SKU-1001 | Monitor     |   100 |
| SKU-1002 | Keyboard    |    30 |
+----------+-------------+-------+
corder
+----------+-------------+----------+-------+
| order_id | customer_id | sku      | price |
+----------+-------------+----------+-------+
|        1 |           1 | SKU-1001 |   100 |
|        2 |           2 | SKU-1002 |    30 |
|        3 |           3 | SKU-1002 |    30 |
|        4 |           4 | SKU-1002 |    30 |
|        5 |           5 | SKU-1002 |    30 |
+----------+-------------+----------+-------+
```

Notice that we are using keyspace `commerce/0` to select data from our tables.

Planning to Move Tables

In this scenario, we are going to add the `customer` keyspace to the `commerce` keyspace we already have. This new keyspace will be backed by its own set of mysqld instances. We will then move the tables `customer` and `corder` from the `commerce` keyspace into the newly created `customer`. The `product` table will remain in the `commerce` keyspace. This operation happens online, which means that it does not block either read or write operations to the tables, **except** for a small window during the final cut-over.

Show our current tablets

```
$ mysql --table --execute="show vitess_tablets"
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| Cell  | Keyspace | Shard | TabletType | State   | Alias            | Hostname  |
   MasterTermStartTime  |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| zone1 | commerce | 0     | MASTER     | SERVING | zone1-0000000100 | localhost |
   2020-08-26T00:37:21Z |
| zone1 | commerce | 0     | REPLICA    | SERVING | zone1-0000000101 | localhost |
                        |
| zone1 | commerce | 0     | RDONLY     | SERVING | zone1-0000000102 | localhost |
                        |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
```

As can be seen, we have 3 tablets running, with tablet ids 100, 101 and 102; which we use in the examples to form the tablet alias/names like `zone1-0000000100`, etc.

Create new tablets

The first step in our MoveTables operation is to deploy new tablets for our `customer` keyspace. By the convention used in our examples, we are going to use the tablet ids 200-202 as the `commerce` keyspace previously used `100-102`. Once the tablets have started, we can force the first tablet to be the master using the `InitShardMaster -force` flag:

```
helm upgrade vitess ../../helm/vitess/ -f 201_customer_tablets.yaml
```

After a few minutes the pods should appear running:

```
$ kubectl get pods,jobs
NAME                                              READY   STATUS      RESTARTS   AGE
pod/vtctld-58bd955948-pgz7k                       1/1     Running     0          5m36s
pod/vtgate-zone1-c7444bbf6-t5xc6                  1/1     Running     3          5m36s
pod/zone1-commerce-0-init-shard-master-gshz9      0/1     Completed   0          5m35s
pod/zone1-commerce-0-replica-0                    2/2     Running     0          5m35s
pod/zone1-commerce-0-replica-1                    2/2     Running     0          5m35s
pod/zone1-commerce-0-replica-2                    2/2     Running     0          5m35s
pod/zone1-customer-0-init-shard-master-7w7rm      0/1     Completed   0          84s
pod/zone1-customer-0-replica-0                    2/2     Running     0          84s
pod/zone1-customer-0-replica-1                    2/2     Running     0          84s
pod/zone1-customer-0-replica-2                    2/2     Running     0          84s

NAME                                              COMPLETIONS   DURATION   AGE
job.batch/zone1-commerce-0-init-shard-master      1/1           90s        5m36s
job.batch/zone1-customer-0-init-shard-master      1/1           23s        84s
```

`InitShardMaster` is performed implicitly by Helm for you.

```
kubectl apply -f 201_customer_tablets.yaml
```

After a few minutes the pods should appear running:

```
$ kubectl get pods
NAME                                            READY   STATUS    RESTARTS   AGE
example-etcd-faf13de3-1                         1/1     Running   0          8m11s
example-etcd-faf13de3-2                         1/1     Running   0          8m11s
example-etcd-faf13de3-3                         1/1     Running   0          8m11s
```

```
example-vttablet-zone1-1250593518-17c58396       3/3     Running   1          2m20s
example-vttablet-zone1-2469782763-bfadd780       3/3     Running   1          7m57s
example-vttablet-zone1-2548885007-46a852d0       3/3     Running   1          7m47s
example-vttablet-zone1-3778123133-6f4ed5fc       3/3     Running   1          2m20s
example-zone1-vtctld-1d4dcad0-59d8498459-kdml8   1/1     Running   1          8m11s
example-zone1-vtgate-bc6cde92-6bd99c6888-csnkj   1/1     Running   2          8m11s
vitess-operator-8454d86687-4wfnc                 1/1     Running   0          22m
```

Again, the operator will perform `InitShardMaster` implicitly for you.

Make sure that you restart the port-forward after launching the pods has completed:

```
killall kubectl
./pf.sh &
```

```
for i in 200 201 202; do
 CELL=zone1 TABLET_UID=$i ./scripts/mysqlctl-up.sh
 CELL=zone1 KEYSPACE=customer TABLET_UID=$i ./scripts/vttablet-up.sh
done

vtctlclient InitShardMaster -force customer/0 zone1-200
vtctlclient ReloadSchemaKeyspace customer
```

Show our old and new tablets

```
$ mysql --table --execute="show vitess_tablets"
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| Cell  | Keyspace | Shard | TabletType | State   | Alias            | Hostname  |
   MasterTermStartTime  |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
| zone1 | commerce | 0     | MASTER     | SERVING | zone1-0000000100 | localhost |
   2020-08-26T00:37:21Z |
| zone1 | commerce | 0     | REPLICA    | SERVING | zone1-0000000101 | localhost |
                        |
| zone1 | commerce | 0     | RDONLY     | SERVING | zone1-0000000102 | localhost |
                        |
| zone1 | customer | 0     | MASTER     | SERVING | zone1-0000000200 | localhost |
   2020-08-26T00:52:39Z |
| zone1 | customer | 0     | REPLICA    | SERVING | zone1-0000000201 | localhost |
                        |
| zone1 | customer | 0     | RDONLY     | SERVING | zone1-0000000202 | localhost |
                        |
+-------+----------+-------+------------+---------+------------------+-----------+----------------
```

Note: The following change does not change actual routing yet. We will use a *switch* directive to achieve that shortly.

Start the Move

In this step we will initiate the MoveTables, which copies tables from the commerce keyspace into customer. This operation does not block any database activity; the MoveTables operation is performed online:

```
$ vtctlclient MoveTables -source commerce -tables 'customer,corder' Create
    customer.commerce2customer
```

You can read this command as: "Start copying the tables called **customer** and **corder** from the **commerce** keyspace to the **customer** keyspace."

A few things to note:

- In a real-world situation this process might take hours/days to complete if the table has millions or billions of rows.
- The workflow name (`commerce2customer` in this case) is arbitrary, you can name it whatever you want. You will use this handle/alias for the other `MoveTables` related commands like `SwitchReads` and `SwitchWrites` in the next steps.

Check routing rules (optional)

To see what happens under the covers, let's look at the **routing rules** that the `MoveTables` operation created. These are instructions used by VTGate to determine which backend keyspace to send requests for a given table or schema/table combo:

```
$ vtctlclient GetRoutingRules commerce
{
  "rules": [
    {
      "fromTable": "customer",
      "toTables": [
        "commerce.customer"
      ]
    },
    {
      "fromTable": "customer.customer",
      "toTables": [
        "commerce.customer"
      ]
    },
    {
      "fromTable": "corder",
      "toTables": [
        "commerce.corder"
      ]
    },
    {
      "fromTable": "customer.corder",
      "toTables": [
        "commerce.corder"
      ]
    }
  ]
}
```

Basically what the `MoveTables` operation has done is to create routing rules to explicitly route queries to the tables `customer` and `corder`, as well as the schema/table combos of `customer.customer` and `customer.corder` to the respective tables in the `commerce` keyspace. This is done so that when `MoveTables` creates the new copy of the tables in the `customer` keyspace, there is no ambiguity about where to route requests for the `customer` and `corder` tables. All requests for those tables will keep going to the original instance of those tables in `commerce` keyspace. Any changes to the tables after the `MoveTables` is executed will be copied faithfully to the new copy of these tables in the `customer` keyspace.

Monitoring Progress (optional)

In this example there are only a few rows in the tables, so the `MoveTables` operation only takes seconds. If the tables were large, you may need to monitor the progress of the operation. There is no simple way to get a percentage complete status, but you can estimate the progress by running the following against the master tablet of the target keyspace:

```
$ vtctlclient VReplicationExec zone1-0000000200 "select * from _vt.copy_state"
+----------+------------+--------+
| vrepl_id | table_name | lastpk |
+----------+------------+--------+
+----------+------------+--------+
```

In the above case the copy is already complete, but if it was still ongoing, there would be details about the last PK (primary key) copied by the VReplication copy process. You could use information about the last copied PK along with the max PK and data distribution of the source table to estimate progress.

Validate Correctness (optional)

We can use VDiff to checksum the two sources and confirm they are in sync:

```
$ vtctlclient VDiff customer.commerce2customer
```

You should see output similar to the following:

```
Summary for corder: {ProcessedRows:5 MatchingRows:5 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
Summary for customer: {ProcessedRows:5 MatchingRows:5 MismatchedRows:0 ExtraRowsSource:0
   ExtraRowsTarget:0}
```

This can obviously take a long time on very large tables.

Phase 1: Switch Non-Primary Reads

Once the MoveTables operation is complete, the first step in making the changes live is to *switch* `SELECT` statements to read from the new keyspace. Other statements will continue to route to the `commerce` keyspace. By staging this as two operations, Vitess allows you to test the changes and reduce the associated risks. For example, you may have a different configuration of hardware or software on the new keyspace.

```
vtctlclient MoveTables -tablet_types=rdonly,replica SwitchTraffic customer.commerce2customer
```

Interlude: check the routing rules (optional)

Lets look at what has happened to the routing rules since we checked the last time. The `SwitchTraffic` commands above added a number of new routing rules for the tables involved in the `MoveTables` operation/workflow, e.g.:

```
$ vtctlclient GetRoutingRules commerce
{
  "rules": [
    {
      "fromTable": "commerce.corder@rdonly",
      "toTables": [
        "customer.corder"
```

```
    ]
  },
  {
    "fromTable": "commerce.corder@replica",
    "toTables": [
      "customer.corder"
    ]
  },
  {
    "fromTable": "customer.customer@rdonly",
    "toTables": [
      "customer.customer"
    ]
  },
  {
    "fromTable": "customer@rdonly",
    "toTables": [
      "customer.customer"
    ]
  },
  {
    "fromTable": "commerce.customer@replica",
    "toTables": [
      "customer.customer"
    ]
  },
  {
    "fromTable": "corder",
    "toTables": [
      "commerce.corder"
    ]
  },
  {
    "fromTable": "customer.corder@replica",
    "toTables": [
      "customer.corder"
    ]
  },
  {
    "fromTable": "customer.customer@replica",
    "toTables": [
      "customer.customer"
    ]
  },
  {
    "fromTable": "customer.corder",
    "toTables": [
      "commerce.corder"
    ]
  },
  {
    "fromTable": "corder@rdonly",
    "toTables": [
      "customer.corder"
    ]
  },
```

```
    {
      "fromTable": "customer.corder@rdonly",
      "toTables": [
        "customer.corder"
      ]
    },
    {
      "fromTable": "customer",
      "toTables": [
        "commerce.customer"
      ]
    },
    {
      "fromTable": "customer.customer",
      "toTables": [
        "commerce.customer"
      ]
    },
    {
      "fromTable": "commerce.customer@rdonly",
      "toTables": [
        "customer.customer"
      ]
    },
    {
      "fromTable": "corder@replica",
      "toTables": [
        "customer.corder"
      ]
    },
    {
      "fromTable": "customer@replica",
      "toTables": [
        "customer.customer"
      ]
    }
  ]
}
```

As you can see, we now have requests to the `rdonly` and `replica` tablets for the source `commerce` keyspace being redirected to the in-sync copy of the table in the target `customer` keyspace.

Phase 2: Switch Writes and Primary Reads

After the replica/rdonly reads have been *switched*, and you have verified that the system is operating as expected, it is time to *switch* the *write* and primary read operations. The command to execute the switch is very similar to the one in Phase 1:

```
$ vtctlclient MoveTables -tablet_types=master SwitchTraffic customer.commerce2customer
```

Note

While we have switched reads and writes separately in this example, you can also switch all traffic, read and write, at the same time. If you don't specify the `-tablet_types` parameter `SwitchTraffic` will start serving traffic from the target for all tablet

types.

Interlude: check the routing rules (optional)

Again, if we look at the routing rules after the `SwitchTraffic` process, we will find that it has been cleaned up, and replaced with a blanket redirect for the moved tables (`customer` and `corder`) from the source keyspace (`commerce`) to the target keyspace (`customer`), e.g.:

```
$ vtctlclient GetRoutingRules commerce
{
  "rules": [
    {
      "fromTable": "commerce.customer",
      "toTables": [
        "customer.customer"
      ]
    },
    {
      "fromTable": "customer",
      "toTables": [
        "customer.customer"
      ]
    },
    {
      "fromTable": "commerce.corder",
      "toTables": [
        "customer.corder"
      ]
    },
    {
      "fromTable": "corder",
      "toTables": [
        "customer.corder"
      ]
    }
  ]
}
```

Reverse workflow

As part of the `SwitchTraffic` operation above, Vitess will automatically (unless you supply the `-reverse_replication false` flag) setup a reverse VReplication workflow to copy changes now applied to the moved tables in the target keyspace (i.e. tables `customer` and `corder` in the `customer` keyspace) back to the original source tables in the source keyspace (`customer`). This allows us to reverse the process using additional `SwitchTraffic` commands without data loss, even after we have started writing to the new copy of the table in the new keyspace. Note that the workflow for this reverse process is given the name of the original workflow with `_reverse` appended. So in our example where the MoveTables workflow was called `commerce2customer`; the reverse workflow would be `commerce2customer_reverse`.

Finalize and Cleanup

The final step is to **remove** the data from the original keyspace. As well as freeing space on the original tablets, this is an important step to eliminate potential future confusion. If you have a misconfiguration down the line and accidentally route queries for the `customer` and `corder` tables to `commerce`, it is much better to return a *"table not found"* error, rather than return stale data:

```
$ vtctlclient MoveTables Complete customer.commerce2customer
```

After this step is complete, you should see an error (in Vitess 9.0 and later) similar to:

```
# Expected to fail!
mysql --table < ../common/select_commerce_data.sql
Using commerce/0
Customer
ERROR 1146 (42S02) at line 4: vtgate: http://localhost:15001/: target: commerce.0.master,
   used tablet: zone1-100
(localhost): vttablet: rpc error: code = NotFound desc = Table 'vt_commerce.customer'
   doesn't exist (errno 1146)
(sqlstate 42S02) (CallerID: userData1): Sql: "select * from customer", BindVars: {}
```

This confirms that the data has been correctly cleaned up. Note that the `Complete` process also cleans up the reverse VReplication workflow mentioned above. Regarding the routing rules, Vitess behavior here has changed recently:

- Before Vitess 9.0, the the routing rules from the source keyspace to the target keyspace was not cleaned up. The assumption was that you might still have applications that refer to the tables by their explicit `schema.table` designation, and you want these applications to (still) transparently be forwarded to the new location of the data. When you are absolutely sure that no applications are using this access pattern, you can clean up the routing rules by manually adjusting the routing rules via the `vtctlclient ApplyRoutingRules` command.
- From Vitess 9.0 onwards, the routing rules from the source keyspace to the target keyspace are also cleaned up as part of the `Complete` operation. If this is not the behavior you want, you can choose to either delay the `Complete` until you are sure the routing rules (and source data) are no longer required; or you can perform the same steps as `Complete` manually.

Next Steps

Congratulations! You've successfully moved tables between keyspaces. The next step to try out is to shard one of your keyspaces in Resharding.

Operational

description: User guides covering operational aspects of Vitess skip_sections: true

Backup and Restore

description: User guides covering how to backup and restore in Vitess

Backup and Restore

Backup and Restore are integrated features provided by tablets managed by Vitess. As well as using *backups* for data integrity, Vitess will also create and restore backups for provisioning new tablets in an existing shard.

Concepts

Vitess supports pluggable interfaces for both Backup Storage Services and Backup Engines.

Before backing up or restoring a tablet, you need to ensure that the tablet is aware of the Backup Storage system and Backup engine that you are using. To do so, use the following command-line flags when starting a vttablet that has access to the location where you are storing backups.

Backup Storage Services Currently, Vitess has plugins for:

- A network-mounted path (e.g. NFS)
- Google Cloud Storage
- Amazon S3
- Ceph

Backup Engines The engine is the techology used for generating the backup. Currently Vitess has plugins for:

- Builtin: Shutdown an instance and copy all the database files (default)
- XtraBackup: An online backup using Percona's XtraBackup

VTTablet Configuration

The following options can be used to configure VTTablet for backups:

Flags

backup_storage_implementation

Specifies the implementation of the Backup Storage interface to use. Current plugin options available are:

file: NFS or any other filesystem-mounted network drive.

gcs: Google Cloud Storage.

s3: Amazon S3.

ceph: Ceph Object Gateway S3 API.

```
  </td>
</tr>
<tr>
  <td><code>backup_engine_implementation</code></td>
  <td>Specifies the implementation of the Backup Engine to
    use.<br><br>
    Current options available are:
    <ul>
      <li><code>builtin</code>: Copy all the database files into specified storage. This is
         the default.</li>
      <li><code>xtrabackup</code>: Percona Xtrabackup.</li>
    </ul>
```

```
  </td>
</tr>
<tr>
  <td><code>backup_storage_hook</code></td>
  <td>If set, the content of every file to backup is sent to a hook. The
    hook receives the data for each file on stdin. It should echo the
    transformed data to stdout. Anything the hook prints to stderr will
    be printed in the vttablet logs.<br>
    Hooks should be located in the <code>vthook</code> subdirectory of the
    <code>VTROOT</code> directory.<br>
    The hook receives a <code>-operation write</code> or a
    <code>-operation read</code> parameter depending on the direction
    of the data processing. For instance, <code>write</code> would be for
    encryption, and <code>read</code> would be for decryption.</br>
  </td>
</tr>
<tr>
  <td><code>backup_storage_compress</code></td>
  <td>This flag controls if the backups are compressed by the Vitess code.
    By default it is set to true. Use
    <code>-backup_storage_compress=false</code> to disable.</br>
    This is meant to be used with a <code>-backup_storage_hook</code>
    hook that already compresses the data, to avoid compressing the data
    twice.
  </td>
</tr>
<tr>
  <td><code>file_backup_storage_root</code></td>
  <td>For the <code>file</code> plugin, this identifies the root directory
    for backups.
  </td>
</tr>
<tr>
  <td><code>gcs_backup_storage_bucket</code></td>
  <td>For the <code>gcs</code> plugin, this identifies the
    <a href="https://cloud.google.com/storage/docs/concepts-techniques#concepts">bucket</a>
    to use.</td>
</tr>
<tr>
  <td><code>s3_backup_aws_region</code></td>
  <td>For the <code>s3</code> plugin, this identifies the AWS region.</td>
</tr>
<tr>
  <td><code>s3_backup_storage_bucket</code></td>
  <td>For the <code>s3</code> plugin, this identifies the AWS S3
    bucket.</td>
</tr>
<tr>
  <td><code>ceph_backup_storage_config</code></td>
  <td>For the <code>ceph</code> plugin, this identifies the path to a text
    file with a JSON object as configuration. The JSON object requires the
    following keys: <code>accessKey</code>, <code>secretKey</code>,
    <code>endPoint</code> and <code>useSSL</code>. Bucket name is computed
    from keyspace name and shard name is separated for different
    keyspaces / shards.</td>
</tr>
```

```
<tr>
  <td><code>restore_from_backup</code></td>
  <td>Indicates that, when started with an empty MySQL instance, the
    tablet should restore the most recent backup from the specified
    storage plugin.</td>
</tr>
<tr>
  <td><code>xbstream_restore_flags</code></td>
  <td>String<br>Flags to pass to xbstream command during restore. These should be space
     separated and will be added to the end of the command. These need to match the ones
     used for backup e.g. --compress / --decompress, --encrypt / --decrypt</td>
</tr>
<tr>
  <td><code>xtrabackup_root_path</code></td>
  <td>For the <code>xtrabackup</code> backup engine, directory location of the xtrabackup
     executable, e.g., /usr/bin</td>
</tr>
<tr>
  <td><code>xtrabackup_backup_flags</code></td>
  <td>String<br>For the <code>xtrabackup</code> backup engine, flags to pass to backup
     command. These should be space separated and will be added to the end of the
     command.</td>
</tr>
<tr>
  <td><code>xtrabackup_stream_mode</code></td>
  <td>String<br>For the <code>xtrabackup</code> backup engine, which mode to use if
     streaming, valid values are <code>tar</code> and <code>xbstream</code>. Defaults to
     <code>tar</code>.</td>
</tr>
<tr>
  <td><code>xtrabackup_user</code></td>
  <td>For the <code>xtrabackup</code> backup engine, required user that xtrabackup will use
     to connect to the database server. This user must have all necessary privileges. For
     details, please refer to xtrabackup documentation.</td>
</tr>
<tr>
  <td><code>xtrabackup_stripes</code></td>
  <td>Unit<br>For the <code>xtrabackup</code> backup engine, if greater than 0, use data
     striping across this many destination files to parallelize data transfer and
     decompression.</td>
</tr>
<tr>
  <td><code>xtrabackup_stripe_block_size</code></td>
  <td>Unit<br>For the <code>xtrabackup</code> backup engine, size in bytes of each block
     that gets sent to a given stripe before rotating to the next stripe.Defaults to
     <code>102400</code>.</td>
</tr>
<tr>
  <td><code>xtrabackup_prepare_flags</code></td>
  <td>String<br>Flags to pass to prepare command. These should be space separated and will
     be added to the end of the command.</td>
</tr>
```

Authentication Note that for the Google Cloud Storage plugin, we currently only support Application Default Credentials. It means that access to Cloud Storage is automatically granted by virtue of the fact that you're already running within Google Compute Engine or Container Engine.

For this to work, the GCE instances must have been created with the scope that grants read-write access to Cloud Storage. When using Container Engine, you can do this for all the instances it creates by adding `--scopes storage-rw` to the `gcloud container clusters create` command.

Backup Frequency We recommend to take backups regularly e.g. you should set up a cron job for it.

To determine the proper frequency for creating backups, consider the amount of time that you keep replication logs and allow enough time to investigate and fix problems in the event that a backup operation fails.

For example, suppose you typically keep four days of replication logs and you create daily backups. In that case, even if a backup fails, you have at least a couple of days from the time of the failure to investigate and fix the problem.

Concurrency The backup and restore processes simultaneously copy and either compress or decompress multiple files to increase throughput. You can control the concurrency using command-line flags:

- The vtctl Backup command uses the `-concurrency` flag.
- vttablet uses the `-restore_concurrency` flag.

If the network link is fast enough, the concurrency matches the CPU usage of the process during the backup or restore process.

Creating a Backup

Creating a backup

The default backup implementation is 'builtin', however we strongly recommend using the xtrabackup engine as it is more robust and allows for non-blocking backups. Restores will always be done with whichever engine was used to create the backup.

Prerequisite A compatible version of xtrabackup and xbstream, if needed, must be present in $PATH prior to running the backup command.

Supported Versions of Xtrabackup

- For MySQL 5.7 and MariaDB 10
- MySQL 8.0
- MariaDB 10.3 is not compatible with xtrabackup.

Basic VTTablet Configuration Required vttablet flags:

- -backup_engine_implementation=xtrabackup
- -xtrabackup_user string
  - The string should be the user that xtrabackup will use to connect to the database server. This user must have the necessary privileges.

Required for MySQL 8.0:

- -xtrabackup_stream_mode=xbstream

```
vtctl Backup <tablet-alias>
```

If the engine is `builtin`, replication will be stopped prior to shutting down mysqld for the backup.

If the engine is `xtrabackup`, the tablet can continue to serve traffic while the backup is running.

```
vtctl BackupShard [-allow_master=false] <keyspace/shard>
```

Restoring a backup

When a tablet starts, Vitess checks the value of the `-restore_from_backup` command-line flag to determine whether to restore a backup to that tablet.

- If the flag is present, Vitess tries to restore the most recent backup from the Backup Storage system when starting the tablet.
- If the flag is absent, Vitess does not try to restore a backup to the tablet. This is the equivalent of starting a new tablet in a new shard.

As noted in the Configuration section, the flag is generally enabled all of the time for all of the tablets in a shard. By default, if Vitess cannot find a backup in the Backup Storage system, the tablet will start up empty. This behavior allows you to bootstrap a new shard before any backups exist.

If the `-wait_for_backup_interval` flag is set to a value greater than zero, the tablet will instead keep checking for a backup to appear at that interval. This can be used to ensure tablets launched concurrently while an initial backup is being seeded for the shard (e.g. uploaded from cold storage or created by another tablet) will wait until the proper time and then pull the new backup when it's ready.

```
vttablet ... -backup_storage_implementation=file \
             -file_backup_storage_root=/nfs/XXX \
             -restore_from_backup
```

Managing backups

vtctl provides two commands for managing backups:

- ListBackups displays the existing backups for a keyspace/shard in chronological order.

  ```
  vtctl ListBackups <keyspace/shard>
  ```

- RemoveBackup deletes a specified backup for a keyspace/shard.

  ```
  RemoveBackup <keyspace/shard> <backup name>
  ```

You can also confirm your backup finished by viewing the files on disk. You will still need to test and verify these backups for completeness. Note that backups are stored by keyspace and shard under backup_storage_root:

```
~/vtdataroot/backups/commerce/0/2021-03-10.205419.zone1-0000000102:
backup.xbstream.gz  MANIFEST
```

Each backup contains a manifest file with general information about the backup:

```
MySQL 8.0 xbstream Manifest
{
  "BackupMethod": "xtrabackup",
  "Position": "MySQL56/c022ad67-81fc-11eb-aa0e-1c1bb572885f:1-50",
  "BackupTime": "2021-03-11T00:01:37Z",
```

```
  "FinishedTime": "2021-03-11T00:01:42Z",
  "FileName": "backup.xbstream.gz",
  "ExtraCommandLineParams": "--no-server-version-check",
  "StreamMode": "xbstream",
  "NumStripes": 0,
  "StripeBlockSize": 102400,
  "SkipCompress": false
}
```

Bootstrapping a new tablet

Bootstrapping a new tablet is almost identical to restoring an existing tablet. The only thing you need to be cautious about is that the tablet specifies its keyspace, shard and tablet type when it registers itself at the topology. Specifically, make sure that the following additional vttablet parameters are set:

```
    -init_keyspace <keyspace>
    -init_shard <shard>
    -init_tablet_type replica|rdonly
```

The bootstrapped tablet will restore the data from the backup and then apply changes, which occurred after the backup, by restarting replication.

Common Errors and Resolutions

```
E0310 08:15:45.336083  197442 main.go:72] remote error: rpc error: code = Unknown desc =
   TabletManager.Backup on zone1-0000000102 error: xtrabackupUser must be specified.:
   xtrabackupUser must be specified
```

Fix: Set vttablet flag -xtrabackup_user

```
E0310 08:22:22.260044  200147 main.go:72] remote error: rpc error: code = Unknown desc =
   TabletManager.Backup on zone1-0000000102 error: unable to start backup: exec:
   "xtrabackup": executable file not found in $PATH: unable to start backup: exec:
   "xtrabackup": executable file not found in $PATH
```

Fixes:

```
* Ensure the xtrabackup binary is in the $PATH for the $USER running vttablet
* Alternatively, set -xtrabackup_root_path on vttablet provide path to xtrabackup/xbstream
   binaries via vttablet flag
```

```
I0310 12:34:47.900363  211809 backup.go:163] I0310 20:34:47.900004 xtrabackupengine.go:310]
   xtrabackup stderr: Invalid --stream argument: tar
Streaming in tar format is no longer supported in 8.0; use xbstream instead
```

Fix: Set -xtrabackup_stream_mode to xbstream on vttablet

```
I0310 12:49:32.279729  215835 backup.go:163] I0310 20:49:32.279435 xtrabackupengine.go:310]
   xtrabackup stderr: Error: Unsupported server version 8.0.23-0ubuntu0.20.04.1.
I0310 12:49:32.279773  215835 backup.go:163] I0310 20:49:32.279485 xtrabackupengine.go:310]
   xtrabackup stderr: Please upgrade PXB, if a new version is available. To continue with
   risk, use the option --no-server-version-check.
```

To continue with risk: Set -xtrabackup_backup_flags=–no-server-version-check. Note this occurs when your MySQL server version is technically unsupported by `xtrabackup`.

Backing up Topology Server

The Topology Server stores metadata (and not tablet data). It is recommended to create a backup using the method described by the underlying plugin:

- etcd
- ZooKeeper
- Consul

Upgrading Vitess

This document highlights things to be aware of when upgrading a Vitess production installation to a newer Vitess release.

Generally speaking, upgrading Vitess is a safe and easy process because it is explicitly designed for it. This is because at YouTube we followed the practice of releasing new versions often (usually from the tip of the Git master branch).

Compatibility

Our versioning strategy is based on Semantic Versioning.

Vitess version numbers follow the format `MAJOR.MINOR.PATCH`. We guarantee compatibility when upgrading to a newer **patch** or **minor** version. Upgrades to a higher **major** version may require manual configuration changes.

In general, always **read the 'Upgrading' section of the release notes**. It will mention any incompatible changes and necessary manual steps.

Upgrade Order

We recommend to upgrade components in a bottom-to-top order such that "old" clients will talk to "new" servers during the transition.

Please use this upgrade order (unless otherwise noted in the release notes):

- vttablet
- vtctld
- vtgate
- application code which links client libraries

Canary Testing

Within the vtgate and vttablet components, we recommend to canary single instances, keyspaces and cells. Upgraded canary instances can "bake" for several hours or days to verify that the upgrade did not introduce a regression. Eventually, you can upgrade the remaining instances.

Rolling Upgrades

We recommend to automate the upgrade process with a configuration management software. It will reduce the possibility of human errors and simplify the process of managing all instances.

As of June 2016 we do not have templates for any major open-source configuration management software because our internal upgrade process is based on a proprietary software. Therefore, we invite open-source users to contribute such templates.

Any upgrade should be a rolling release i.e. usually one tablet at a time within a shard. This ensures that the remaining tablets continue serving live traffic and there is no interruption.

Upgrading the Master Tablet

The *master* tablet of each shard should always be updated last in the following manner:

- verify that all *replica* tablets in the shard have been upgraded
- reparent away from the current *master* to a *replica* tablet
- upgrade old *master* tablet

Making Schema Changes

description:

This user guide describes the problem space of schema changes and the various approaches you may use with Vitess.

Quick links:

- Vitess supports EXPERIMENTAL managed, online schema changes via `gh-ost` or `pt-online-schema-change`, and with visibility and control over the migration process
- Multiple approaches to unmanaged schema changes, either blocking, or owned by the user/DBA.

Some background on schema changes follows.

The schema change problem

Schema change is one of the oldest problems in MySQL. With accelerated development and deployment flows, engineers find they need to deploy schema changes sometimes on a daily basis. With the growth of data this task becomes more and more difficult. A direct MySQL `ALTER TABLE` statement is a blocking (no reads nor writes are possible on the migrated table) and resource heavy operation; variants of `ALTER TABLE` include `InnoDB` Online DDL, which allows for some concurrency on a `primary` (aka `master`) server, but still blocking on replicas, leading to unacceptable replication lags once the statement hits the replicas.

`ALTER TABLE` operations are greedy, consume as much CPU/Disk IO as needed, are uninterruptible and uncontrollable. Once the operation has begun, it must run to completion; aborting an `ALTER TABLE` may be more expensive than letting it run through, depending on the progress the migration has made.

Direct `ALTER TABLE` is fine in development or possibly staging environments, where datasets are either small, or where table locking is acceptable.

ALTER TABLE solutions

Busy production systems tend to use either of these two approaches, to make schema changes less disruptive to ongoing production traffic:

- Using online schema change tools, such as gh-ost and pt-online-schema-change. These tools *emulate* an `ALTER TABLE` statement by creating a *ghost* table in the new desired format, and slowly working through copying data from the existing table, while also applying ongoing changes throughout the migration. Online schema change tools can be throttled on high load, and can be interrupted at will.
- Run the migration independently on replicas; when all replicas have the new schema, demote the `primary` and promote a `replica` as the new `primary`; then, at leisure, run the migration on the demoted server. Two considerations if using this approach are:
  - Each migration requires a failover (aka successover, aka planned reparent).
  - Total wall clock time is higher since we run the same migration in sequence on different servers.

Schema change cycle and operation

The cycle of schema changes, from idea to production, is complex, involves multiple environments and possibly multiple teams. Below is one possible breakdown common in production. Notice how even interacting with the database itself takes multiple steps:

1. Design: the developer designs a change, tests locally
2. Publish: the developer calls for review of their changes (e.g. on a Pull Request)
3. Review: developer's colleagues and database engineers to check the changes and their impact
4. Formalize: what is the precise `ALTER TABLE` statement to be executed? If running with `gh-ost` or `pt-online-schema-change`, what are the precise command line flags?
5. Locate: where does this change need to go? Which keyspace/cluster? Is this cluster sharded? What are the shards? Having located the affected MySQL clusters, which is the `primary` server per cluster?
6. Schedule: is there an already running migration on the relevant keyspace/cluster(s)?
7. Execute: invoke the command. In the time we waited, did the identity of `primary` servers change?
8. Audit/control: is the migration in progress? Do we need to abort for some reason?
9. Cut-over/complete: a potential manual step to complete the migration process
10. Cleanup: what do you do with the old tables? An immediate `DROP` is likely not advisable. What's the alternative?
11. Notify user: let the developer know their changes are now in production.
12. Deploy & merge: the developer completes their process.

Steps `4 - 10` are tightly coupled with the database or with the infrastrcture around the database.

Schema change and Vitess

Vitess solves or automates multiple parts of the flow:

Formalize In managed, online schema changes the user supplies a valid SQL `ALTER TABLE` statement, and Vitess generates the `gh-ost` or `pt-online-schema-change` command line invocation. It will also auto generate config files and set up the environment for those tools. This is hidden from the user.

Locate For a given table in a given keyspace, Vitess knows at all times:

- In which shards (MySQL clusters) the table is found
- Which is the `primary` server per shard.

When using either managed schema changes, or direct schema changes via `vtctl` or `vtgate`, Vitess resolves the discovery of the affected servers automatically, and this is hidden from the user.

Schedule In managed, online schema changes, Vitess owns and tracks all pending and active migrations. As a rule of thumb, it is generally advisable to only run one online schema change at a time on a given server. Following that rule of thumb, Vitess will queue incoming schema change requests and schedule them to run sequentially.

Execute In managed, online schema changes, Vitess owns the execution of `gh-ost` or `pt-online-schema-change`. While these run in the background, Vitess keeps track of the migratoin state.

In direct schema changes via `vtctl` or `vtgate`, Vitess issues a synchronous `ALTER TABLE` statement on the relevant shards.

Audit/control In managed, online schema changes, Vitess keeps track of the state of the migration. It automatically detects when the migration is complete or has failed. It will detect failure even if the tablet itself, which is running the migration, fails. Vitess allows the user to cancel a migration. If such a migration is queued by the scheduler, then it is unqueued. If it's already running, it is interrupted and aborted. Vitess allows the user to check on a migration status across the relevant shards.

Cut-over/complete Vitess runs automated cut-overs. The migration will complete as soon as it's able to.

Cleanup In the case of managed, online schema changes via `pt-online-schema-change`, Vitess will ensure to drop the triggers in case the tool failed to do so for whatever reason.

Vitess automatically garbage-collects the "old" tables, artifacts of `gh-ost` and `pt-online-schema-change`. It drops those tables in an incremental, non blocking method.

The various approaches

Vitess allows a variety of approaches to schema changes, from fully automated to fully owned by the user.

- Managed, online schema changes are *experimental* at this time, but are Vitess's way forward
- Direct, blocking ALTERs are generally impractical in production given that they can block writes for substantial lengths of time.
- User controlled migrations are allowed, and under the user's responsibility.

See breakdown in managed, online schema changes and in unmanaged schema changes.

Applying, auditing, and controlling Online DDL

Vitess provides two interfaces to interacting with Online DDL:

- SQL commands, via `VTGate`
- Command line interface, via `vtctl`

Supported interactions are:

- Running migrations (submitting Online DDL requests)
- Tracking migrations
- Cancelling a migration
- Cancelling all pending migrations
- Retrying a migration
- Reverting a migration

Running migrations

To run a managed schema migration, you should:

- Formulate your DDLs (`CREATE`, `ALTER`, `DROP`) queries
- Choose a ddl_strategy

When the user submits an online DDL, Vitess responds with a UUID, a job Id used to later track or control the migration. The migration does not start immediately. It is queued at the tablets and executed at some point in the future.

```
mysql> set @@ddl_strategy='online';

mysql> alter table corder add column ts timestamp not null default current_timestamp;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| bf4598ab_8d55_11eb_815f_f875a4d24e90 |
+--------------------------------------+

mysql> drop table customer;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 6848c1a4_8d57_11eb_815f_f875a4d24e90 |
+--------------------------------------+
```

- `@@ddl_strategy` behaves like a MySQL session variable, though is only recognized by `VTGate`. Setting `@@ddl_strategy` only applies to that same connection and does not affect other connections. The strategy applies to all migrations executed in that session. You may subsequently set `@@ddl_strategy` to different value.
- If you run `vtgate` without `-ddl_strategy`, then `@@ddl_strategy` defaults to `'direct'`, which implies schema migrations are synchronous. You will need to `set @@ddl_strategy='gh-ost'` to run followup `ALTER TABLE` statements via `gh-ost`.
- If you run `vtgate -ddl_strategy "gh-ost"`, then `@@ddl_strategy` defaults to `'gh-ost'` in each new session. Any `ALTER TABLE` will run via `gh-ost`. You may `set @@ddl_strategy='pt-osc'` to make migrations run through `pt-online-schema-change`, or `set @@ddl_strategy='direct'` to run migrations synchronously.

Via vtctl/ApplySchema You may use `vtctl` or `vtctlclient` (the two are interchangeable for the purpose of this document) to apply schema changes. The `ApplySchema` command supports both synchronous and online schema migrations. To run an online schema migration you will supply the `-ddl_strategy` command line flag:

```
$ vtctlclient ApplySchema -ddl_strategy "online" -sql "ALTER TABLE demo MODIFY id bigint
   UNSIGNED" commerce
a2994c92_f1d4_11ea_afa3_f875a4d24e90
```

Tracking migrations

You may track the status of a single or of multiple migrations. Since migrations run asycnhronously, it is the user's responsibility to audit the progress and state of submitted migrations. Users are likely to want to know when a migration is complete (or failed) so as to be able to deploy code changes or run other operations.

Common patterns are:

- Show state of a specific migration
- Show all `running`, `complete` or `failed` migrations
- Show recent migrations

Via VTGate/SQL Examples for a single shard cluster:

```
mysql> show vitess_migrations like 'bf4598ab_8d55_11eb_815f_f875a4d24e90' \G
*************************** 1. row ***************************
                  id: 23
     migration_uuid: bf4598ab_8d55_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: corder
migration_statement: alter table corder add column ts timestamp not null default
   current_timestamp()
           strategy: online
            options:
    added_timestamp: 2021-03-25 12:35:01
requested_timestamp: 2021-03-25 12:34:58
    ready_timestamp: 2021-03-25 12:35:04
  started_timestamp: 2021-03-25 12:35:04
 liveness_timestamp: 2021-03-25 12:35:06
completed_timestamp: 2021-03-25 12:35:06
  cleanup_timestamp: NULL
   migration_status: complete
           log_path:
          artifacts: _bf4598ab_8d55_11eb_815f_f875a4d24e90_20210325123504_vrepl,
            retries: 0
             tablet: zone1-0000000100
     tablet_failure: 0
           progress: 100
  migration_context: vtgate:a8352418-8d55-11eb-815f-f875a4d24e90
         ddl_action: alter
            message:
        eta_seconds: 0
```

```
mysql> show vitess_migrations like 'complete' \G
...
*************************** 21. row ***************************
```

```
                id: 24
     migration_uuid: 6848c1a4_8d57_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: customer
migration_statement: drop table customer
           strategy: online
            options:
    added_timestamp: 2021-03-25 12:46:53
requested_timestamp: 2021-03-25 12:46:51
    ready_timestamp: 2021-03-25 12:46:57
  started_timestamp: 2021-03-25 12:46:57
 liveness_timestamp: 2021-03-25 12:46:57
completed_timestamp: 2021-03-25 12:46:57
  cleanup_timestamp: NULL
   migration_status: complete
           log_path:
          artifacts: _vt_HOLD_6848c1a48d5711eb815ff875a4d24e90_20210326104657,
            retries: 0
             tablet: zone1-0000000100
     tablet_failure: 0
           progress: 100
  migration_context: vtgate:a8352418-8d55-11eb-815f-f875a4d24e90
         ddl_action: drop
            message:
        eta_seconds: 0
```

```
mysql> show vitess_migrations where completed_timestamp > now() - interval 1 day;
+----+--------------------------------------+----------+-------+--------------+-------------+-----
| id | migration_uuid                       | keyspace | shard | mysql_schema | mysql_table
   | migration_statement                                                                          |
   strategy | options | added_timestamp     | requested_timestamp | ready_timestamp     |
   started_timestamp   | liveness_timestamp  | completed_timestamp | cleanup_timestamp |
   migration_status | log_path | artifacts
                                                        | retries | tablet           |
   tablet_failure | progress | migration_context                           | ddl_action |
   message | eta_seconds |
+----+--------------------------------------+----------+-------+--------------+-------------+-----
| 23 | bf4598ab_8d55_11eb_815f_f875a4d24e90 | commerce | 0     | vt_commerce  | corder
   | alter table corder add column ts timestamp not null default current_timestamp() |
   online   |         | 2021-03-25 12:35:01 | 2021-03-25 12:34:58 | 2021-03-25 12:35:04 |
   2021-03-25 12:35:04 | 2021-03-25 12:35:06 | 2021-03-25 12:35:06 | NULL              |
   complete         |          |
   _bf4598ab_8d55_11eb_815f_f875a4d24e90_20210325123504_vrepl, |       0 | zone1-0000000100
   |              0 |      100 | vtgate:a8352418-8d55-11eb-815f-f875a4d24e90 | alter      |
          |           0 |
| 24 | 6848c1a4_8d57_11eb_815f_f875a4d24e90 | commerce | 0     | vt_commerce  | customer
   | drop table customer                                                                          |
   online   |         | 2021-03-25 12:46:53 | 2021-03-25 12:46:51 | 2021-03-25 12:46:57 |
   2021-03-25 12:46:57 | 2021-03-25 12:46:57 | 2021-03-25 12:46:57 | NULL              |
   complete         |          | _vt_HOLD_6848c1a48d5711eb815ff875a4d24e90_20210326104657,
     |       0 | zone1-0000000100 |              0 |      100 |
   vtgate:a8352418-8d55-11eb-815f-f875a4d24e90 | drop       |         |           0 |
| 25 | 6fd57dd3_8d57_11eb_815f_f875a4d24e90 | commerce | 0     | vt_commerce  | customer
   | revert 6848c1a4_8d57_11eb_815f_f875a4d24e90                                                  |
   online   |         | 2021-03-25 12:47:08 | 2021-03-25 12:47:04 | 2021-03-25 12:47:12 |
```

```
   2021-03-25 12:47:12 | 2021-03-25 12:47:12 | 2021-03-25 12:47:12 | NULL                |
   complete         |          | _vt_HOLD_6848c1a48d5711eb815ff875a4d24e90_20210326104657,
     |       0 | zone1-0000000100 |               0 |      100 |
   vtgate:a8352418-8d55-11eb-815f-f875a4d24e90 | create     |         |           0 |
+----+----------------------------------------+----------+------+-------------+-------------+-----
```

- `show vitess_migrations` shows the entire history of migrations.
- `show vitess_migrations like ...` filters migrations by `migration_uuid`, or `migration_context`, or `migration_status`.
- `show vitess_migrations where ...` lets the user specify arbitrary conditions.
- All commands return results for the keyspace (schema) in use.

Via vtctl/ApplySchema Examples for a 4-shard cluster:

```
$ vtctlclient OnlineDDL commerce show ab3ffdd5_f25c_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:24:33 | 2020-09-09
   05:24:34 | complete         |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
+-----------------+-------+--------------+-------------+------------+-----------------------------

$ vtctlclient OnlineDDL commerce show 8a797518_f25c_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | running          |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 | 2020-09-09
   05:23:33 | complete         |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | running          |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | running          |
+-----------------+-------+--------------+-------------+------------+-----------------------------

$ vtctlclient OnlineDDL commerce show 8a797518_f25c_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
```

```
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+--------------------------
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | failed           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | failed           |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | failed           |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 | 2020-09-09
   05:23:33 | complete         |
+-----------------+-------+--------------+-------------+------------+--------------------------

$ vtctlclient OnlineDDL commerce show recent
+-----------------+-------+--------------+-------------+------------+--------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+--------------------------
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   63b5db0c_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:22:41 | 2020-09-09
   05:22:42 | complete         |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 | 2020-09-09
   05:23:33 | complete         |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:24:33 | 2020-09-09
   05:24:34 | complete         |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   63b5db0c_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:22:41 | 2020-09-09
   05:22:42 | complete         |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | failed           |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   63b5db0c_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:22:41 | 2020-09-09
   05:22:42 | complete         |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                     | failed           |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   63b5db0c_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:22:41 | 2020-09-09
   05:22:42 | complete         |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
```

```
                        | failed           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   ab3ffdd5_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:25:13 | 2020-09-09
   05:25:14 | complete         |
+-----------------+-------+--------------+-------------+------------+-----------------------------

$ vtctlclient OnlineDDL commerce show failed
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                        | failed           |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                        | failed           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   8a797518_f25c_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 05:23:32 |
                        | failed           |
+-----------------+-------+--------------+-------------+------------+-----------------------------
```

The syntax for tracking migrations is:

```
vtctlclient OnlineDDL <keyspace> show
   <migration_id|all|recent|queued|ready|running|complete|failed|cancelled>
```

Showing migration logs

`gh-ost` and `pt-osc` tools generate logs files, which are retrievable for 24 hours after migration completion/failure.

```
mysql> show vitess_migration '3a273866_e867_11eb_ab12_0a43f95f28a3' logs \G
*************************** 1. row ***************************
migration_log: 2021-07-19 07:59:23 INFO starting gh-ost
   261355426d8fc31b590733ca8ff8e79012103c18
2021-07-19 07:59:23 INFO Migrating `vt_commerce`.`corder`
2021-07-19 07:59:23 INFO executing gh-ost-on-startup hook:
   /tmp/online-ddl-3a273866_e867_11eb_ab12_0a43f95f28a3-943208852/gh-ost-on-startup
ok
2021-07-19 07:59:23 INFO inspector connection validated on ip-REDACTED:17100
2021-07-19 07:59:23 INFO User has SUPER, REPLICATION SLAVE privileges, and has ALL
   privileges on `vt_commerce`.*
2021-07-19 07:59:23 INFO binary logs validated on ip-REDACTED:17100
2021-07-19 07:59:23 INFO Restarting replication on ip-REDACTED:17100 to make sure binlog
   settings apply to replication thread
2021-07-19 07:59:23 INFO Inspector initiated on ip-REDACTED:17100, version 5.7.30-log
2021-07-19 07:59:23 INFO Table found. Engine=InnoDB
...
```

Cancelling a migration

The user may cancel a migration, as follows:

- If the migration hasn't started yet (it is `queued` or `ready`), then it is removed from queue and will not be executed.
- If the migration is `running`, then it is forcibly interrupted. The migration is expected to transition to `failed` state.
- In all other cases, cancelling a migration has no effect.

Via VTGate/SQL Examples for a single shard cluster:

```
                 id: 28
     migration_uuid: aa89f255_8d68_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: corder
migration_statement: alter table corder add column handler_id int not null
           strategy: gh-ost
            options:
    added_timestamp: 2021-03-25 14:50:27
requested_timestamp: 2021-03-25 14:50:24
    ready_timestamp: 2021-03-25 14:50:31
  started_timestamp: 2021-03-25 14:50:32
 liveness_timestamp: 2021-03-25 14:50:32
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: running
...

mysql> alter vitess_migration 'aa89f255_8d68_11eb_815f_f875a4d24e90' cancel;
Query OK, 1 row affected (0.01 sec)

mysql> show vitess_migrations like 'aa89f255_8d68_11eb_815f_f875a4d24e90' \G
*************************** 1. row ***************************
                 id: 28
     migration_uuid: aa89f255_8d68_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: corder
migration_statement: alter table corder add column handler_id int not null
           strategy: gh-ost
            options: --throttle-flag-file=/tmp/throttle.flag
    added_timestamp: 2021-03-25 14:50:27
requested_timestamp: 2021-03-25 14:50:24
    ready_timestamp: 2021-03-25 14:50:31
  started_timestamp: 2021-03-25 14:50:32
 liveness_timestamp: 2021-03-25 14:50:32
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: failed
...
```

- `alter vitess_migration ... cancel` takes exactly one migration's UUID.
- `alter vitess_migration ... cancel` responds with number of affected migrations.

Via vtctl/ApplySchema Examples for a 4-shard cluster:

```
vtctlclient OnlineDDL <keyspace> cancel <migration_id>
```

Example:

```
$ vtctlclient OnlineDDL commerce show 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+------------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+------------------------------
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | running          |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | running          |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | running          |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | running          |
+-----------------+-------+--------------+-------------+------------+------------------------------

$ vtctlclient OnlineDDL commerce cancel 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+--------------+
|     Tablet      | RowsAffected |
+-----------------+--------------+
| test-0000000401 |            1 |
| test-0000000101 |            1 |
| test-0000000201 |            1 |
| test-0000000301 |            1 |
+-----------------+--------------+

$ vtctlclient OnlineDDL commerce show 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+------------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+------------------------------
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | failed           |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | failed           |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | failed           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                       | failed           |
+-----------------+-------+--------------+-------------+------------+------------------------------
```

Cancelling all keyspace migrations

The user may cancel all migrations in a keyspace. A migration is cancellable if it is in `queued`, `ready` or `running` states, as described previously. It is a high impact operation and should be used with care.

Via VTGate/SQL Examples for a single shard cluster:

```
mysql> alter vitess_migration cancel all;
Query OK, 1 row affected (0.02 sec)
```

Via vtctl/ApplySchema Examples for a 4-shard cluster:

```
vtctlclient OnlineDDL <keyspace> cancel-all
```

Example:

```
$ vtctlclient OnlineDDL commerce show all
+------------------+-------+--------------+-------------+--------------------------------------+--
|      Tablet      | shard | mysql_schema | mysql_table |            migration_uuid
              | strategy | started_timestamp | completed_timestamp | migration_status |
+------------------+-------+--------------+-------------+--------------------------------------+--
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c581994_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | queued           |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c6420c9_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | queued           |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c7040df_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | queued           |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c7c0572_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | queued           |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c87f7cd_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | queued           |
+------------------+-------+--------------+-------------+--------------------------------------+--

$  vtctlclient OnlineDDL commerce cancel-all
+------------------+--------------+
|      Tablet      | RowsAffected |
+------------------+--------------+
| zone1-0000000100 |            5 |
+------------------+--------------+

 vtctlclient OnlineDDL commerce show all
+------------------+-------+--------------+-------------+--------------------------------------+--
|      Tablet      | shard | mysql_schema | mysql_table |            migration_uuid
              | strategy | started_timestamp | completed_timestamp | migration_status |
+------------------+-------+--------------+-------------+--------------------------------------+--
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c581994_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | cancelled        |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c6420c9_353a_11eb_8b72_f875a4d24e90 | online   |                   |
                       | cancelled        |
```

```
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c7040df_353a_11eb_8b72_f875a4d24e90 | online   |                  |
                       | cancelled        |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c7c0572_353a_11eb_8b72_f875a4d24e90 | online   |                  |
                       | cancelled        |
| zone1-0000000100 |     0 | vt_commerce  | corder      |
   2c87f7cd_353a_11eb_8b72_f875a4d24e90 | online   |                  |
                       | cancelled        |
+------------------+-------+--------------+-------------+--------------------------------------+--
```

Retrying a migration

The user may retry running a migration. If the migration is in `failed` or in `cancelled` state, Vitess will re-run the migration, with exact same arguments as previously intended. If the migration is in any other state, `retry` does nothing.

It is not possible to retry a migration with different options. e.g. if the user initially runs `ALTER TABLE demo MODIFY id BIGINT` with `@@ddl_strategy='gh-ost --max-load Threads_running=200'` and the migration fails, retrying it will use exact same options. It is not possible to retry with `@@ddl_strategy='gh-ost --max-load Threads_running=500'`.

Via VTGate/SQL Examples for a single shard cluster:

```
*************************** 1. row ***************************
                 id: 28
     migration_uuid: aa89f255_8d68_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: corder
migration_statement: alter table corder add column handler_id int not null
           strategy: gh-ost
            options: --throttle-flag-file=/tmp/throttle.flag
    added_timestamp: 2021-03-25 14:50:27
requested_timestamp: 2021-03-25 14:50:24
    ready_timestamp: 2021-03-25 14:56:22
  started_timestamp: 2021-03-25 14:56:22
 liveness_timestamp: 2021-03-25 14:56:22
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: failed
...

mysql> alter vitess_migration 'aa89f255_8d68_11eb_815f_f875a4d24e90' retry;
Query OK, 1 row affected (0.00 sec)

mysql> show vitess_migrations like 'aa89f255_8d68_11eb_815f_f875a4d24e90' \G
*************************** 1. row ***************************
                 id: 28
     migration_uuid: aa89f255_8d68_11eb_815f_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: corder
migration_statement: alter table corder add column handler_id int not null
           strategy: gh-ost
            options: --throttle-flag-file=/tmp/throttle.flag
```

```
    added_timestamp: 2021-03-25 14:50:27
requested_timestamp: 2021-03-25 14:50:24
    ready_timestamp: 2021-03-25 14:56:42
  started_timestamp: 2021-03-25 14:56:42
 liveness_timestamp: 2021-03-25 14:56:42
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: running
...
```

- `alter vitess_migration ... cancel` takes exactly one migration's UUID.
- `alter vitess_migration ... cancel` responds with number of affected migrations.

Via vtctl/ApplySchema Examples for a 4-shard cluster:

```
$ vtctlclient OnlineDDL commerce show 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                     | failed           |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                     | failed           |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                     | failed           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:32:31 |
                     | failed           |
+-----------------+-------+--------------+-------------+------------+-----------------------------

$ vtctlclient OnlineDDL commerce retry 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+--------------+
|     Tablet      | RowsAffected |
+-----------------+--------------+
| test-0000000101 |            1 |
| test-0000000201 |            1 |
| test-0000000301 |            1 |
| test-0000000401 |            1 |
+-----------------+--------------+

$ vtctlclient OnlineDDL commerce show 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy | started_timestamp | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   |                    |
                     | queued           |
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   |                    |
```

```
                      | queued           |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   |                    |
                      | queued           |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   |                    |
                      | queued           |
+-----------------+-------+--------------+-------------+------------+-----------------------------

$ vtctlclient OnlineDDL commerce show 2201058f_f266_11ea_bab4_0242c0a8b007
+-----------------+-------+--------------+-------------+------------+-----------------------------
|     Tablet      | shard | mysql_schema | mysql_table | ddl_action |
   migration_uuid            | strategy |  started_timestamp  | completed_timestamp |
   migration_status |
+-----------------+-------+--------------+-------------+------------+-----------------------------
| test-0000000101 |   -40 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:37:33 |
                      | running          |
| test-0000000401 | c0-   | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:37:33 |
                      | running          |
| test-0000000201 | 40-80 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:37:33 |
                      | running          |
| test-0000000301 | 80-c0 | vt_commerce  | demo        | alter      |
   2201058f_f266_11ea_bab4_0242c0a8b007 | online   | 2020-09-09 06:37:33 |
                      | running          |
+-----------------+-------+--------------+-------------+------------+-----------------------------
```

Reverting a migration

Vitess offers *lossless revert* for online schema migrations: the user may regret a table migration after completion, and roll back the table's schema to previous state *without loss of data*. See Revertible Migrations.

Via VTGate/SQL Examples for a single shard cluster:

```
mysql> show create table corder\G

Create Table: CREATE TABLE `corder` (
  `order_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `customer_id` bigint(20) DEFAULT NULL,
  `sku` varbinary(128) DEFAULT NULL,
  `price` bigint(20) DEFAULT NULL,
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.01 sec)

mysql> alter table corder drop column ts, add key customer_idx(customer_id);
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 1a689113_8d77_11eb_815f_f875a4d24e90 |
+--------------------------------------+
```

```
mysql> show create table corder\G

Create Table: CREATE TABLE `corder` (
  `order_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `customer_id` bigint(20) DEFAULT NULL,
  `sku` varbinary(128) DEFAULT NULL,
  `price` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`order_id`),
  KEY `customer_idx` (`customer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

mysql> revert vitess_migration '1a689113_8d77_11eb_815f_f875a4d24e90';
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| a02e6612_8d79_11eb_815f_f875a4d24e90 |
+--------------------------------------+

mysql> show create table corder\G

Create Table: CREATE TABLE `corder` (
  `order_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `customer_id` bigint(20) DEFAULT NULL,
  `sku` varbinary(128) DEFAULT NULL,
  `price` bigint(20) DEFAULT NULL,
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

- A `revert` is its own migration, hence has its own UUID

```
$ vtctlclient OnlineDDL commerce revert 2201058f_f266_11ea_bab4_0242c0a8b007
```

VExec commands for greater control and visibility

`vtctlclient OnlineDDL` command should provide with most needs. However, Vitess gives the user greater control through the `VExec` command and via SQL queries.

For schema migrations, Vitess allows operations on the virtual table `_vt.schema_migrations`. Queries on this virtual table scatter to the underlying tablets and gather or manipulate data on their own, private backend tables (which incidentally are called by the same name). `VExec` only allows specific types of queries on that table.

- `SELECT`: you may SELECT any column, or `SELECT *`. `vtctlclient OnlineDDL show` commands only present with a subset of columns, and so running `VExec SELECT` provides greater visibility. Some columns that are not shown are:
  - `log_path`: tablet server and path where migration logs are.
  - `artifacts`: tables created by the migration. This can be used to determine which tables need cleanup.
  - `alter`: the exact `alter` statement used by the migration
  - `options`: any options passed by the user (e.g. `--max-load=Threads_running=200`)
  - Various timestamps indicating the migration progress Aggregate functions do not work as expected and should be avoided. `LIMIT` and `OFFSET` are not supported.

- `UPDATE`: you may directly update the status of a migration. You may only change status into `cancel` or `retry`, which Vitess interprets similarly to a `vtctlclient OnlineDDL cancel/retry` command. However, you get greater control as you may filter on a specific `shard`.
- `DELETE`: unsupported
- `INSERT`: unsupported, used internally only to advertise new migration requests to the tablets.

The syntax to run `VExec` queries is:

```
vtctlclient VExec <keyspace>.<migration_id> "<sql query>"
```

Examples:

```
$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "select * from
   _vt.schema_migrations"

$ vtctlclient VExec commerce.91b5c953-e1e2-11ea-a097-f875a4d24e90 "update
   _vt.schema_migrations set migration_status='retry'"

$ vtctlclient VExec commerce.91b5c953-e1e2-11ea-a097-f875a4d24e90 "update
   _vt.schema_migrations set migration_status='retry' where shard='40-80'"
```

```
$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "select shard,
   mysql_table, migration_uuid, started_timestamp, completed_timestamp, migration_status
   from _vt.schema_migrations"
+-----------------+-------+-------------+--------------------------------------+-----------------
|     Tablet      | shard | mysql_table |            migration_uuid            |
   started_timestamp  | completed_timestamp | migration_status |
+-----------------+-------+-------------+--------------------------------------+-----------------
| test-0000000301 | 80-c0 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                     | failed           |
| test-0000000101 |   -40 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                     | failed           |
| test-0000000201 | 40-80 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   08:31:47 |                     | failed           |
| test-0000000401 | c0-   | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                     | failed           |
+-----------------+-------+-------------+--------------------------------------+-----------------

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "update
   _vt.schema_migrations set migration_status='retry' where
   migration_uuid='2201058f_f266_11ea_bab4_0242c0a8b007' and shard='40-80'"
+-----------------+--------------+
|     Tablet      | RowsAffected |
+-----------------+--------------+
| test-0000000201 |            1 |
+-----------------+--------------+

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "select shard,
   mysql_table, migration_uuid, started_timestamp, completed_timestamp, migration_status
   from _vt.schema_migrations"
+-----------------+-------+-------------+--------------------------------------+-----------------
|     Tablet      | shard | mysql_table |            migration_uuid            |
   started_timestamp  | completed_timestamp | migration_status |
+-----------------+-------+-------------+--------------------------------------+-----------------
| test-0000000301 | 80-c0 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                     | failed           |
| test-0000000201 | 40-80 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   08:34:59 |                     | running          |
```

```
| test-0000000101 |   -40 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                      | failed           |
| test-0000000401 | c0-   | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                      | failed           |
+-----------------+-------+-------------+--------------------------------------+-----------------

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "update
   _vt.schema_migrations set migration_status='cancel' where
   migration_uuid='2201058f_f266_11ea_bab4_0242c0a8b007' and shard='40-80'"
+-----------------+--------------+
|     Tablet      | RowsAffected |
+-----------------+--------------+
| test-0000000201 |            1 |
+-----------------+--------------+

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "select shard,
   mysql_table, migration_uuid, started_timestamp, completed_timestamp, migration_status
   from _vt.schema_migrations"
+-----------------+-------+-------------+--------------------------------------+-----------------
|     Tablet      | shard | mysql_table |            migration_uuid            |
   started_timestamp  | completed_timestamp | migration_status |
+-----------------+-------+-------------+--------------------------------------+-----------------
| test-0000000401 | c0-   | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                      | failed           |
| test-0000000101 |   -40 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                      | failed           |
| test-0000000201 | 40-80 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   08:34:59 |                      | failed           |
| test-0000000301 | 80-c0 | demo        | 2201058f_f266_11ea_bab4_0242c0a8b007 | 2020-09-09
   06:37:33 |                      | failed           |
+-----------------+-------+-------------+--------------------------------------+-----------------

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "update
   _vt.schema_migrations set migration_status='cancel' where
   migration_uuid='2201058f_f266_11ea_bab4_0242c0a8b007' and shard='40-80'"
<no result>

$ vtctlclient VExec commerce.2201058f_f266_11ea_bab4_0242c0a8b007 "select shard, log_path
   from _vt.schema_migrations"
+-----------------+-------+-------------------------------------------------------------------------
|     Tablet      | shard |                                    log_path
                                  |
+-----------------+-------+-------------------------------------------------------------------------
| test-0000000201 | 40-80 |
   11ac2af6e63e:/tmp/online-ddl-2201058f_f266_11ea_bab4_0242c0a8b007-657478384 |
| test-0000000101 |   -40 |
   e779a82d35d7:/tmp/online-ddl-2201058f_f266_11ea_bab4_0242c0a8b007-901629215 |
| test-0000000401 | c0-   |
   5aad1249ab91:/tmp/online-ddl-2201058f_f266_11ea_bab4_0242c0a8b007-039568897 |
| test-0000000301 | 80-c0 |
   5e7c662679d3:/tmp/online-ddl-2201058f_f266_11ea_bab4_0242c0a8b007-532703073 |
+-----------------+-------+-------------------------------------------------------------------------
```

Online DDL strategies

Vitess supports both managed, online schema migrations (aka Online DDL) as well as unmanaged migrations. How Vitess runs a schema migration depends on the *DDL strategy*. Vitess allows these strategies:

- `direct`: the direct apply of DDL to your database. This is not an online DDL. It is a synchronous and blocking operation. This is the default strategy.
- `online`: utilizes Vitess's built in VReplication mechanism.
- `gh-ost`: uses 3rd party GitHub's gh-ost tool.
- `pt-osc`: uses 3rd party Percona's pt-online-schema-change as part of Percona Toolkit

`CREATE` and `DROP` are managed in the same way, by Vitess, whether strategy is `online`, `gh-ost` or `pt-osc`.

Specifying a DDL strategy

You will set either `@@ddl_strategy` session variable, or `-ddl_strategy` command line flag. Examples:

```
$ vtctlclient ApplySchema -ddl_strategy "online" -sql "ALTER TABLE demo MODIFY id bigint
   UNSIGNED" commerce
a2994c92_f1d4_11ea_afa3_f875a4d24e90
```

```
$ vtctlclient ApplySchema -ddl_strategy "gh-ost --max-load Threads_running=200" -sql "ALTER
   TABLE demo add column status int" commerce
```

```
$ mysql -h 127.0.0.1 -P 15306 commerce
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> SET @@ddl_strategy='online';
Query OK, 0 rows affected (0.00 sec)

mysql> ALTER TABLE demo ADD COLUMN sample INT;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| fa2fb689_f1d5_11ea_859e_f875a4d24e90 |
+--------------------------------------+
1 row in set (0.00 sec)
```

Choosing a DDL strategy

Different strategies have different behavior for `ALTER` statements. Sections below first break down specific handling and notes for each strategy, followed by an evaluation of the differences.

online/VReplication The `online` strategy invokes Vitess's built in VReplication mechanism. It is the mechanism behind resharding, materialized views, imports from external databases, and more. VReplication migrations use the same logic for copying data as do other VReplication operations, and as such the `online` strategy is known to be compatible with overall Vitess behavior. VReplication is authored by the maintainers of Vitess.

VReplication migrations enjoy the general features of VReplication:

- Seamless integration with Vitess.
- Seamless use of the throttler mechanism.
- Visibility into internal working and status of VReplication.

gh-ost gh-ost was developed by GitHub as a lightweight and safe schema migration tool.

To be able to run online schema migrations via `gh-ost`:

- If you're on Linux/amd64 architecture, and on `glibc 2.3` or similar, there are no further dependencies. Vitess comes with a built-in `gh-ost` binary, that is compatible with your system. Note that the Vitess Docker images use this architecture, and `gh-ost` comes pre-bundled and compatible.
- On other architectures:
  - Have `gh-ost` executable installed
  - Run `vttablet` with `-gh-ost-path=/full/path/to/gh-ost` flag

Vitess automatically creates a MySQL account for the migration, with a randomly generated password. The account is destroyed at the end of the migration.

Vitess takes care of setting up the necessary command line flags. It automatically creates a hooks directory and populates it with hooks that report `gh-ost`'s progress back to Vitess. You may supply additional flags for your migration as part of `@@ddl_strategy` session variable (using `VTGate`) or `-ddl_strategy` command line flag (using `vtctl`). Examples:

- `set @@ddl_strategy='gh-ost --max-load Threads_running=200';`
- `set @@ddl_strategy='gh-ost --max-load Threads_running=200 --critical-load Threads_running=500 --critical-lo --default-retries=512';`
- `vtctl ApplySchema -ddl_strategy "gh-ost --allow-nullable-unique-key --chunk-size 200" ...`

Note: Do not override the following flags: `alter, database, table, execute, max-lag, force-table-names, serve-socket-file, hooks-path, hooks-hint-token, panic-flag-file`. Overriding any of these may cause Vitess to lose control and track of the migration, or even to migrate the wrong table.

`gh-ost` throttling is done via Vitess's own tablet throttler, based on replication lag.

Using pt-online-schema-change pt-online-schema-change is part of Percona Toolkit, a set of Perl scripts. To be able to use `pt-online-schema-change`, you must have the following setup on all your tablet servers (normally tablets are co-located with MySQL on same host and so this implies setting up on all MySQL servers):

- `pt-online-schema-change` tool installed and is executable
- Perl `libdbi` and `libdbd-mysql` modules installed. e.g. on Debian/Ubuntu, `sudo apt-get install libdbi-perl libdbd-mysql-perl`
- Run `vttablet` with `-pt-osc-path=/full/path/to/pt-online-schema-change` flag.

Note that on Vitess Docker images, `pt-online-schema-change` and dependencies are pre-installed.

Vitess automatically creates a MySQL account for the migration, with a randomly generated password. The account is destroyed at the end of the migration.

Vitess takes care of supplying the command line flags, the DSN, the username & password. It also sets up `PLUGINS` used to communicate migration progress back to the tablet. You may supply additional flags for your migration as part of `@@ddl_strategy` session variable (using `VTGate`) or `-ddl_strategy` command line flag (using `vtctl`). Examples:

- `set @@ddl_strategy='pt-osc --null-to-not-null';`
- `set @@ddl_strategy='pt-osc --max-load Threads_running=200';`
- `vtctl ApplySchema -ddl_strategy "pt-osc --alter-foreign-keys-method auto --chunk-size 200" ...`

Vitess tracks the state of the `pt-osc` migration. If it fails, Vitess makes sure to drop the migration triggers. Vitess keeps track of the migration even if the tablet itself restarts for any reason. Normally that would terminate the migration; Vitess will cleanup the triggers if so, or will happily let the migration run to completion if not.

Do not override the following flags: `alter, pid, plugin, dry-run, execute, new-table-name, [no-]drop-new-table, [no-]drop-old-table`.

`pt-osc` throttling is done via Vitess's own tablet throttler, based on replication lag, and via a `pt-online-schema-change` plugin.

Comparing the options There are pros and cons to using any of the strategies. Some notable differences:

General

- All three options mimic an `ALTER TABLE` statement by creating and populating a shadow/ghost table behind the scenes, slowly bringing it up to date, and finally switching between the original and shadow tables.
- All three options utilize the Vitess throttler.

Support

- VReplication (`online` strategy) is internal to Vitess and supported by the Vitess maintainers.
- `gh-ost` enjoys partial, informal support from Vitess maintainers.
- `pt-online-schema-change` is out of the maintainers control.

Setup

- VReplication is part of Vitess
- A `gh-ost` binary is embedded within the Vitess binary, compatible with `glibc 2.3` and `Linux/amd64`. The user may choose to use their own `gh-ost` binary, configured with `-gh-ost-path`.
- `pt-online-schema-change` is not included in Vitess, and the user needs to set it up on tablet hosts.
  - Note that on Vitess Docker images, `pt-online-schema-change` and dependencies *are* pre-installed.

Load

- `pt-online-schema-change` uses triggers to propagate changes. This method is traditionally known to generate high load on the server. Both VReplication and `gh-ost` tail the binary logs to capture changes, and this approach is known to be more lightweight.
- When throttled, `pt-online-schema-change` still runs trigger actions, whereas both VReplication and `gh-ost` cease transfer of data (they may keep minimal bookkeeping operations).

Cut-over

- Both `pt-online-schema-change` and `gh-ost` have an atomic cut-over: at the end of the migration, the tables are switched, and incoming queries are momentarily blocked, but not lost.
- VReplication causes a brief outage at time of cut-over (subject to change): apps will not be able to *write* to the original table during cut-over, and will return with error.
- VReplication cut-over is only safe when all traffic comes through Vitess/VTGate (subject to change). Any DML query running on migrated table at time of cut-over, and which executes directly on the MySQL server without going through Vitess, might lose its data.

MySQL compatibility

- `pt-online-schema-change` partially supports foreign keys. Neither `gh-ost` nor `VReplication` support foreign keys.

Vitess functionality comparison

Strategy	Managed	Online	Trackable	Declarative	Revertible	Traffic
`direct`	No	MySQL*	No	No	No	Any
`pt-osc`	Yes	Yes*	Yes	Yes	`CREATE,DROP`	Any
`gh-ost`	Yes	Yes*	Yes+	Yes	`CREATE,DROP`	Any
`online`	Yes	Yes*	Yes	Yes	`CREATE,DROP,ALTER`	Vitess

- **Managed**: whether Vitess schedules and operates the migration
- **Online**:
  - MySQL supports limited online ("In place") DDL and instant DDL. See support chart.
  - `gh-ost`, `online` do not support foreign keys
  - `pt-osc` has support for foreign keys (may apply collateral blocking operations)
- **Trackable**: able to determine migration state (`ready`, `running`, `complete` etc)
  - `gh-ost` also makes available *progress %* and *ETA seconds*
- **Declarative**: support `-declarative` flag
- **Revertible**: `online` strategy supports revertible `ALTER` statements (or `ALTER`s implied by `-declarative` migrations). All managed strategies supports revertible `CREATE` and `ALTER`.
- **Traffic**: `online` migration cut-over uses Vitess specific blocking of traffic, and is therefore only safe when write traffic to the tables runs entirely through Vitess/VTGate. `gh-ost` and `pt-osc` use generic MySQL blocking/locking mechanisms, and it is safe to run some write traffic on the migrated table outside Vitess.

Declarative migrations

Vitess's managed schema changes offer *declarative* online schema migrations:

- The user may indicate a desired table schema and Vitess will make it so, whether the table exists or not, or
- The user may indicate a table should not exist, and Vitess will make it so.

Declarative DDLs are expressed via:

- Complete `CREATE TABLE` statement (make the table in desired state)
- `DROP TABLE` statement (make the table go)

Altering tables in declarative DDL is done by issuing `CREATE TABLE` statements with the desired state. `ALTER` statements are not allowed.

Declarative DDLs have the property of being idempotent. For example, a user may submit the same `CREATE TABLE` statement *twice*, one after another. If the 1st is successful, then the 2nd is a noop, and considered as implicitly successful. Likewise, two `DROP TABLE` DDLs for same statement will each ensure the table does not exist. If the 1st is successful, then the 2nd has nothing to do and is implicitly successful.

Usage

Add `-declarative` to any of the online DDL strategies. Example:

```
mysql> set @@ddl_strategy='online -declarative';

-- The following migration creates a new table:
mysql> create table decl_table(id int primary key);
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| b06475e5_8a74_11eb_badd_f875a4d24e90 |
+--------------------------------------+

-- The next migration will implicitly ALTER the table decl_table into desired state:
mysql> create table decl_table(id int primary key, ts timestamp not null);
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| b7d6e6fb_8a74_11eb_badd_f875a4d24e90 |
+--------------------------------------+

-- Next migration does not change table structure, hence is a noop and implicitly
   successful:
mysql> create table decl_table(id int primary key, ts timestamp not null);
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 110574b1_8a75_11eb_badd_f875a4d24e90 |
+--------------------------------------+
```

Consider migration `b7d6e6fb_8a74_11eb_badd_f875a4d24e90` above, which results in an `ALTER`. A look into the migration shows:

```
mysql> show vitess_migrations like 'b7d6e6fb_8a74_11eb_badd_f875a4d24e90'\G
*************************** 1. row ***************************
                id: 19
     migration_uuid: b7d6e6fb_8a74_11eb_badd_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: decl_table
migration_statement: create table decl_table (
    id int primary key,
    ts timestamp not null
)
           strategy: online
            options: -declarative
    added_timestamp: 2021-03-21 20:39:08
requested_timestamp: 2021-03-21 20:39:07
    ready_timestamp: 2021-03-21 20:39:10
  started_timestamp: 2021-03-21 20:39:10
 liveness_timestamp: 2021-03-21 20:39:13
completed_timestamp: 2021-03-21 20:39:13
  cleanup_timestamp: NULL
   migration_status: complete
           log_path:
          artifacts: _b7d6e6fb_8a74_11eb_badd_f875a4d24e90_20210321203910_vrepl,
```

```
           retries: 0
            tablet: zone1-0000000100
    tablet_failure: 0
          progress: 100
 migration_context: vtgate:38368dbe-8a60-11eb-badd-f875a4d24e90
        ddl_action: alter
           message: ADD COLUMN `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
              CURRENT_TIMESTAMP
       eta_seconds: 0
```

Note how while the migration statement is `create`, the migration's `ddl_action` ends up being `alter`, and `message` indicates the alter options.

You may add `-declarative` even if you otherwise supply flags to your favorite strategy. For example, the following is valid:

```
set @@ddl_strategy='gh-ost -declarative -max-load=Threads_running=100';
```

Vitess notes down the `-declarative` flag and does not pass it to `gh-ost`, `pt-osc` or `VReplication`.

Implementation details

The user submits a declarative DDL. Tables schedule the migration to execute, but at time of execution, may modify the migration on the fly and end up running a different migration.

Consider the following types of migrations:

- A `REVERT` has no special behavior, it acts as a normal revert.
- `ALTER` is rejected (migration will fail)
- `DROP`: silently mark as successful if the table does not exist. Otherwise treat the DDL as a normal `DROP`.
- `CREATE`: either,
  - The table does not exist: proceed as normal `CREATE`
  - The table exists: evaluate the SQL diff between the existing table schema and the proposed schema. Either:
    - There is no diff (exact same schema): silently mark as successful
    - There is a diff: rewrite the DDL as an actual `ALTER`, run using relevant strategy.

Declarative DDLs are revertible. Note:

- A declarative migration which ends up being an `ALTER` is only revertible if executed with `online` strategy.
- A declarative migration which ends up being a noop (and implicitly successful), implies a noop revert.

Managed, Online Schema Changes

Note: this feature is **EXPERIMENTAL**.

Vitess offers managed, online schema migrations (aka Online DDL), transparently to the user. Vitess Onine DDL offers:

- Non-blocking migrations
- Migrations are asyncronously auto-scheduled, queued and executed by tablets
- Migration state is trackable
- Migrations are cancellable
- Migrations are retry-able
- Lossless, revertible migrations
- Support for declarative migrations

As general overview: - User chooses a strategy for online DDL (online DDL is opt in) - User submits one or more schema change queries, using the standard MySQL `CREATE TABLE`, `ALTER TABLE` and `DROP TABLE` syntax. - Vitess responds with a Job ID for each schema change query. - Vitess resolves affected shards. - A shard's `primary` tablet schedules the migration to run when possible. - Tablets will independently run schema migrations: - `ALTER TABLE` statements run via `VReplication`, `gh-ost` or `pt-online-schema-change`, as per selected strategy - `CREATE TABLE` statements run directly. - `DROP TABLE` statements run safely and lazily. - Vitess provides the user a mechanism to view migration status, cancel or retry migrations, based on the job ID.

Syntax

The standard MySQL syntax for `CREATE`, `ALTER` and `DROP` is supported.

ALTER TABLE Use the standard MySQL `ALTER TABLE` syntax to run online DDL. Whether your schema migration runs synchronously (the default MySQL behavior) or asynchronously (aka online), is determined by `ddl_strategy`.

We assume we have a keyspace (schema) called `commerce`, with a table called `demo`, that has the following definition:

```
CREATE TABLE `demo` (
  `id` int NOT NULL,
  `status` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB
```

Consider the following schema migration statement:

```
ALTER TABLE demo MODIFY id bigint UNSIGNED;
```

This statement can be executed as:

- a `VReplication`, managed online migration
- a `gh-ost`, managed online migration
- a `pt-online-schema-change`, managed online migration
- a synchronous, unmanaged schema change

See DDL Strategies for discussion around the different options.

CREATE TABLE Use the standard MySQL `CREATE TABLE` syntax. The query goes through the same migration flow as `ALTER TABLE` does. The tablets eventually run the query directly on the MySQL backends.

DROP TABLE Use the standard MySQL `DROP TABLE` syntax. The query goes through the same migration flow as `ALTER TABLE` does. Tables are not immediately dropped. Instead, they are renamed into special names, recognizable by the table lifecycle mechanism, to then slowly and safely transition through lifecycle states into finally getting dropped.

Statement transformations Vitess may modify your queries to qualify for online DDL statement. Modifications include:

- A multi-table `DROP` statement is replaced by multiple `DROP` statements, each operating on a single table (and each tracked by its own job ID).
- A `CREATE INDEX` statement is replaced by the equivalent `ALTER TABLE` statement.

ddl_strategy

You will set either `@@ddl_strategy` session variable, or `-ddl_strategy` command line flag, to control your schema migration strategy, and specifically, to enable and configure online DDL. Details in DDL Strategies. A quick overview:

- The value `"direct"`, means not an online DDL. The empty value (`""`) is also interpreted as `direct`. A query is immediately pushed and applied on backend servers. This is the default strategy.
- The value `"online"` instructs Vitess to run an `ALTER TABLE` online DDL via `VReplication`.
- The value `"gh-ost"` instructs Vitess to run an `ALTER TABLE` online DDL via `gh-ost`.
- The value `"pt-osc"` instructs Vitess to run an `ALTER TABLE` online DDL via `pt-online-schema-change`.
- You may specify arguments for your tool of choice, e.g. `"gh-ost --max-load Threads_running=200"`. Details follow.

`CREATE` and `DROP` statements run in the same way for `"online"`, `"gh-ost"` and `"pt-osc"` strategies, and we consider them all to be *online*.

Running, tracking and controlling Online DDL

Vitess provides two interfaces to interacting with Online DDL:

- SQL commands, via `VTGate`
- Command line interface, via `vtctl`

Supported interactions are:

- Running migrations (submitting Online DDL requests)
- Tracking migrations
- Cancelling a migration
- Cancelling all pending migrations
- Retrying a migration
- Reverting a migration

See Audit and Control for a detailed breakdown. As quick examples:

```
mysql> set @@ddl_strategy='online';

mysql> alter table corder add column ts timestamp not null default current_timestamp;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| bf4598ab_8d55_11eb_815f_f875a4d24e90 |
+--------------------------------------+

mysql> drop table customer;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 6848c1a4_8d57_11eb_815f_f875a4d24e90 |
+--------------------------------------+
```

```
$ vtctlclient ApplySchema -ddl_strategy "online" -sql "ALTER TABLE demo MODIFY id bigint
   UNSIGNED" commerce
a2994c92_f1d4_11ea_afa3_f875a4d24e90
```

Migration flow and states

We highlight how Vitess manages migrations internally, and explain what states a migration goes through.

- Whether via `vtctlclient ApplySchema` or via `VTGate` as described above, a migration request entry is persisted in global `topo` (e.g. the global `etcd` cluster).
- `vtctld` periodically checks on new migration requests.
- `vtctld` resolves the relevant shards, and the `primary` tablet for each shard.
- `vtctld` pushes the request to all relevant `primary` tablets.
- If not all shards confirm receipt, `vtctld` periodically keeps retrying pushing the request to the shards until all approve.
- Internally, tablets persist the request in a designated table in the `_vt` schema. **Do not** manipulate that table directly as that can cause inconsistencies.
- A shard's `primary` tablet owns running the migration. It is independent of other shards. It will schedule the migration to run when possible. A tablet will not run two migrations at the same time.
- A migration is first created in `queued` state.
- If the tablet sees queued migration, and assuming there's no reason to wait, it picks the oldest requested migration in `queued` state, and moves it to `ready` state.
- For `ALTER TABLE` statements:
  - Tablet prepares for the migration. It creates a MySQL account with a random password, to be used by this migration only. It creates the command line invocation, and extra scripts if possible.
  - The tablet then runs the migration. Whether `gh-ost` or `pt-online-schema-change`, it first runs in *dry run* mode, and, if successful, in actual *execute* mode. The migration is then in `running` state.
  - The migration will either run to completion, fail, or be interrupted. If successful, it transitions into `complete` state, which is the end of the road for that migration. If failed or interrupted, it transitions to `failed` state. The user may choose to *retry* failed migrations (see below).
  - The user is able to *cancel* a migration (details below). If the migration hasn't started yet, it transitions to `cancelled` state. If the migration is `running`, then it is interrupted, and is expected to transition into `failed` state.
- For `CREATE TABLE` statements:
  - Tablet runs the statement directly on the MySQL backend
- For `DROP TABLE` statements:
  - A multi-table `DROP TABLE` statement is converted to multiple single-table `DROP TABLE` statements
  - Each `DROP TABLE` is internally replaced with a `RENAME TABLE`
  - Table is renamed using a special naming convention, identified by the Table Lifecycle mechanism
  - As result, a single `DROP TABLE` statement may generate multiple distinct migrations with distinct migration UUIDs.

By way of illustration, suppose a migration is now in `running` state, and is expected to keep on running for the next few hours. The user may initiate a new `ALTER TABLE...` statement. It will persist in global `topo`. `vtctld` will pick it up and advertise it to the relevant tablets. Each will persist the migration request in `queued` state. None will run the migration yet, since another migration is already in progress. In due time, and when the executing migration completes (whether successfully or not), and assuming no other migrations are `queued`, the `primary` tablets, each in its own good time, will execute the new migration.

At this time, the user is responsible to track the state of all migrations. VTTablet does not report back to `vtctld`. This may change in the future.

At this time, there are no automated retries. For example, a failover on a shard causes the migration to fail, and Vitess will not try to re-run the migration on the new `primary`. It is the user's responsibility to issue a `retry`. This may change in the future.

Configuration

- `-retain_online_ddl_tables`: (`vttablet`) determines how long vttablet should keep an old migrated table before purging it. Type: duration. Default: 24 hours.

  Example: `vttablet -retain_online_ddl_tables 48h`

- `-migration_check_interval`: (`vttablet`) interval between checks for submitted migrations. Type: duration. Default: 1 hour.

  Example: `vttablet -migration_check_interval 30s`

- `-enable_online_ddl`: (`vtgate`) whether Online DDL operations are at all possible through `VTGate`. Type: boolean. Default: `true`

  Example: `vtgate -enable_online_ddl=false` to disable access to Online DDL via `VTGate`.

Auto resume after failure

VReplication based migrations (`ddl_strategy="online"`) are resumable across failure and across primary failovers.

Auto retry after failure

Neither `gh-ost` and `pt-osc` are able to resume from point of failure, or after a failover. However, Vitess management can issue an automated retry (starting the migration afresh).

- which `vttablet` initiated the migration
- how many times a migration has been retried
- whether a migration failed due to a `vttablet` failure (as is the case in a failover scenario)

Vitess will auto-retry a failed migration when:

- The migration failed due to a `vttablet` failure, and
- it has not been retried (this is a temporary restriction)

The migration will be transitioned into `queued` state, as if the user requested a `retry` operation. Note that this takes place on a per-shard basis.

The primary use case is a primary failure and failover. The newly promoted tablet will be able to retry the migration that broke during the previous primary failure. To clarify, the migration will start anew, as at this time there is no mechanism to resume a broken migration.

Throttling

All three strategies: `online`, `gh-ost` and `pt-osc` utilize the tablet throttler, which is a cooperative throttler service based on replication lag. The tablet throttler automatically detects topology `REPLICA` tablets and adapts to changes in the topology. See Tablet throttler.

- `online` strategy uses the throttler by the fact VReplication natively uses the throttler on both source and target ends (for both reads and writes)

- `gh-ost` uses the throttler via `--throttle-http`, which is automatically provided by Vitess
- `pt-osc` uses the throttler by replication lag plugin, automatically injected by Vitess.

NOTE that at this time (and subject to change) the tablet throttler is disabled by default. Enable it with `vttablet`'s `-enable-lag-throttler` flag. If the tablet throttler is disabled, schema migrations will not throttle on replication lag.

Table cleanup

All `ALTER` strategies leave artifacts behind. Whether successful or failed, either the original table or the *ghost* table is left still populated at the end of the migration. Vitess explicitly makes sure the tables are not dropped at the end of the migration. This is for two reasons:

- Make the table/data still available for a while, and
- in MySQL pre `8.0.23`, a `DROP TABLE` operation can be dangerous in production as it commonly locks the buffer pool for a substantial period.

The tables are kept for 24 hours after migration completion. Vitess automatically cleans up those tables as soon as a migration completes (either successful or failed). You will normally not need to do anything.

Artifact tables are identifiable via `SELECT artifacts FROM _vt.schema_migrations` in a `VExec` command, see below. You should generally not touch these tables. It's possible to `DROP` those tables with `direct` DDL strategy. Note that dropping tables in production can be risky and lock down your database for a substantial period of time.

Revertible migrations

Vitess's managed schema changes offer *lossless revert* for online schema migrations: the user may regret a table migration after completion, and roll back the table's schema to previous state *without loss of data*.

Revertible migrations supported for:

- `CREATE TABLE` statements: the *revert* is to *uncreate* the table
- `DROP TABLE` statements: the *revert* is to *reinstate* the table, populated with data from time of `DROP`
- `ALTER TABLE` statements: supported in `online` strategy, the *revert* is to reapply previous table schema, without losing any data added/modified since migration completion.
- Another `revert` migration. It is possible to revert a *revert*, revert the revert of a *revert*, and so forth.

Behavior and limitations

- A revert is a migration of its own, with a migration UUID, similarly to "normal" migrations.
- Migrations are only for revertible for `24h` since completion.
- It's only possible to revert the last successful migration on a given table. Illustrated following.
  - In the future it may be possible to revert down the stack of completed migrations.
- `ALTER` migrations are revertible only in `online` strategy.
- If a DDL is a noop, then so is its revert:

  - If a table `t` exists, and an online DDL is `CREATE TABLE IF NOT EXISTS t (...)`, then the DDL does nothing, and its revert will do nothing.
  - If a table `t` does not exist, and an online DDL is `DROP TABLE IF EXISTS t`, then likewise the DDL does nothing, and its revert does nothing.
- Some `ALTER` reverts are not guaranteed to succeed. Examples:
  - An `ALTER` which modifies column `i` from `int` to `bigint`, followed by an `INSERT` that places a value larger than max `int`, cannot be reverted, because Vitess cannot place that new value in the old schema.
  - An `ALTER` which removes a `UNIQUE KEY`, followed by an `INSERT` that populates a duplicate value on some column, may not be reverted if that duplicate violates the removed `UNIQUE` constraint.

  Vitess cannot know ahead of time whether a *revert* is possible or not.

REVERT syntax

Via SQL:

```
REVERT VITESS_MIGRATION '69b17887_8a62_11eb_badd_f875a4d24e90';
```

Via `vtctl`:

```
$ vtctlclient OnlineDDL commerce revert 69b17887_8a62_11eb_badd_f875a4d24e90
```

Both operations return a UUID for the revert migration. The user can track the revert migration to find its state.

Usage & walkthrough

Consider the following annotated flow:

```
mysql> set @@ddl_strategy='online';

mysql> create table t(id int primary key);
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 3837e739_8a60_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete

mysql> alter table t add column ts timestamp not null default current_timestamp;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 6bc591b2_8a60_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete

mysql> show create table t \G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

-- it is now possible to revert 6bc591b2_8a60_11eb_badd_f875a4d24e90, because it was the
   last successful migration on table t.
-- it is not possible to revert 3837e739_8a60_11eb_badd_f875a4d24e90, because while it was
   successful, it is not the last
-- successful migration to run on table t t.

mysql> revert vitess_migration '6bc591b2_8a60_11eb_badd_f875a4d24e90';
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| ead67f31_8a60_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete

mysql> show create table t \G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

-- It is now possible to revert ead67f31_8a60_11eb_badd_f875a4d24e90 as it is the last
   successful migration to run on table t.
-- Reverting ead67f31_8a60_11eb_badd_f875a4d24e90 affectively means restoring the changes
   made by 6bc591b2_8a60_11eb_badd_f875a4d24e90

mysql> revert vitess_migration 'ead67f31_8a60_11eb_badd_f875a4d24e90';
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 3b99f686_8a61_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete

mysql> show create table t \G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

-- Let's try an invalid migration:
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 7fbdf1c7_8a61_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- This will fail because column `id` already exists.

                 id: 11
```

```
     migration_uuid: 7fbdf1c7_8a61_11eb_badd_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table: t
migration_statement: alter table t add column id bigint
           strategy: online
            options:
    added_timestamp: 2021-03-21 18:21:36
requested_timestamp: 2021-03-21 18:21:32
    ready_timestamp: 2021-03-21 18:21:36
  started_timestamp: 2021-03-21 18:21:36
 liveness_timestamp: 2021-03-21 18:21:36
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: failed
...
         ddl_action: alter
            message: Duplicate column name 'id' (errno 1060) (sqlstate 42S21) during query:
                ALTER TABLE `_7fbdf1c7_8a61_11eb_badd_f875a4d24e90_20210321182136_vrepl` add
                column id bigint
...

-- it is impossible to revert 7fbdf1c7_8a61_11eb_badd_f875a4d24e90 because it failed.

+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| c3dff91a_8a61_11eb_badd_f875a4d24e90 |
+--------------------------------------+

mysql> show vitess_migrations like 'c3dff91a_8a61_11eb_badd_f875a4d24e90' \G
*************************** 1. row ***************************
                 id: 12
     migration_uuid: c3dff91a_8a61_11eb_badd_f875a4d24e90
           keyspace: commerce
              shard: 0
       mysql_schema: vt_commerce
        mysql_table:
migration_statement: revert 7fbdf1c7_8a61_11eb_badd_f875a4d24e90
           strategy: online
            options:
    added_timestamp: 2021-03-21 18:23:31
requested_timestamp: 2021-03-21 18:23:26
    ready_timestamp: 2021-03-21 18:23:36
  started_timestamp: NULL
 liveness_timestamp: NULL
completed_timestamp: NULL
  cleanup_timestamp: NULL
   migration_status: failed
...
         ddl_action: revert
            message: can only revert a migration in a 'complete' state. Migration
               7fbdf1c7_8a61_11eb_badd_f875a4d24e90 is in 'failed' state
...
```

```
mysql> insert into t values (1, now());

mysql> select * from t;
+----+---------------------+
| id | ts                  |
+----+---------------------+
|  1 | 2021-03-21 18:26:47 |
+----+---------------------+

mysql> drop table t;
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 69b17887_8a62_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete

mysql> select * from t;
ERROR 1146 (42S02): ...

mysql> revert vitess_migration '69b17887_8a62_11eb_badd_f875a4d24e90';
+--------------------------------------+
| uuid                                 |
+--------------------------------------+
| 9eb00275_8a62_11eb_badd_f875a4d24e90 |
+--------------------------------------+
-- Wait until migration is complete
-- `t` was not really dropped, but renamed away. This REVERT reinstates it.

mysql> select * from t;
+----+---------------------+
| id | ts                  |
+----+---------------------+
|  1 | 2021-03-21 18:26:47 |
+----+---------------------+
```

Implementation details

Revert for `CREATE` and `DROP` are implemented similarly for all online strategies.

- The revert for a `CREATE` DDL is to rename the table away and into a table lifecycle name, rather than actually `DROP` it. This keeps th etale safe for a period of time, and makes it possible to reinstate the table, populated with all data, via a 2nd revert.
- The revert for a `DROP` relies on the fact that Online DDL `DROP TABLE` does not, in fact, drop the table, but actually rename it away. Thus, reverting the `DROP` is merely a `RENAME` back into its original place.
- The revert for `ALTER` is only available for `online` strategy, implemented by `VReplication`. VReplication keep track of a DDL migration by writing down the GTID position through the migration flow. In particular, at time of cut-over and when tables are swapped, VReplication notes the *final* GTID pos for the migration. When a revert is requested, Vitess computes a new VReplication rule/filter for the new stream. It them copies the *final* GTID pos from the reverted migration, and instructs VReplication to resume from that point. As result, a revert for an `ALTER` migration only needs to catch up with the changelog (binary log entries) since the cut-over of the original migration. To elaborate, it does not need to copy table data, and only needs to consider events for the specific table affected by the revert. This makes the revert operation efficient.

Unmanaged Schema Changes

Vitess offers multiple approaches to running unmanaged schema changes. Below, we review each of these approaches.

We assume we have a keyspace (schema) called `commerce`, with a table called `demo`, that has the following definition:

```
CREATE TABLE `demo` (
  `id` int NOT NULL,
  `status` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB
```

ApplySchema

`ApplySchema` is a `vtctlclient` command that can be used to apply a schema change to a keyspace. The main advantage of using this tool is that it performs some sanity checks about the schema before applying it. However, a downside is that it can be a little too strict and may not work for all use cases.

Consider the following examples:

```
$ vtctlclient ApplySchema -sql "ALTER TABLE demo modify id bigint unsigned" commerce
```

```
SHOW CREATE TABLE demo;


CREATE TABLE `demo` (
  `id` bigint unsigned NOT NULL,
  `status` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB
```

In the above, we run a direct, synchronous, blocking `ALTER TABLE` statement. Knowing the table is in `commerce` keyspace, Vitess autodetects the relevant shards, and then autodetects which is the `primary` server in each shard. It then directly invokes the `ALTER TABLE` statement on all shards (concurrently), and the `vtctlclient` command only returns when all are complete.

Vitess will run some validations:

```
$ vtctlclient ApplySchema -sql "ALTER TABLE demo add column status int" commerce
E0908 10:35:53.478462 3711762 main.go:67] remote error: rpc error: code = Unknown desc =
   schema change failed, ExecuteResult: {
  "FailedShards": null,
  "SuccessShards": null,
  "CurSQLIndex": 0,
  "Sqls": [
    "ALTER TABLE demo add column status int"
  ],
  "ExecutorErr": "rpc error: code = Unknown desc = TabletManager.PreflightSchema on
     zone1-0000000100 error: /usr/bin/mysql: exit status 1, output: ERROR 1060 (42S21) at
     line 3: Duplicate column name 'status'\ n: /usr/bin/mysql: exit status 1, output:
     ERROR 1060 (42S21) at line 3: Duplicate column name 'status'\ n",
  "TotalTimeSpent": 87104194
}
```

Vitess was able to determine that the migration is invalid because a column named `status` already exists. The statement never made it to the MySQL servers. These checks are not thorough, though they cover common scenarios.

If the table is large, then `ApplySchema` will reject the statement, try to protect the user from blocking their production servers. You may override that by supplying `-allow_long_unavailability` as follows:

```
$ vtctlclient ApplySchema -allow_long_unavailability -sql "ALTER TABLE demo modify id
   bigint unsigned" commerce
```

VTGate

You may run DDL directly from VTGate just like you would send to a MySQL instance. For example:

```
$ mysql -h 127.0.0.1 -P 15306 commerce
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> ALTER TABLE demo ADD COLUMN sample INT;
Query OK, 0 rows affected (0.04 sec)
```

Just like in the previous example, Vitess will find out what the affected shards are, what the identity is of each shard's `primary`, then invoke the statement on all shards.

You may apply the change to specific shards by connecting directly to those shards:

```
$ mysql -h 127.0.0.1 -P 15306 commerce/-80
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> ALTER TABLE demo ADD COLUMN sample INT;
Query OK, 0 rows affected (0.04 sec)
```

In the above we connect to VTGate via the `mysql` command line client, but of course you may connect with any standard MySQL client or from your application.

Please do note that if VTGate does not recognize a DDL syntax, the statement will get rejected and that this approach is not recommended for changing large tables.

Directly to MySQL

You can apply schema changes directly to the underlying MySQL shard master instances.

VTTablet will eventually notice the change and update itself. This is controlled by the `-queryserver-config-schema-reload-time` parameter which defaults to 1800 seconds.

You can also explicitly issue the `vtctlclient ReloadSchema` command to make it reload immediately. Specify a tablet to reload the schema from, as in:

```
$ vtctlclient ReloadSchema zone1-0000000100
```

Users will likely want to deploy schema changes via `gh-ost` or `pt-online-schema-change`, which do not block the table. Vitess offers managed, online schema changes where it automates the invocation and execution of these tools. Using these schema deployment tools can be a better approach for large tables, because they should incur no downtime.

SQL Statement Analysis

description: User guides covering analyzing SQL statements

Analyzing SQL statements in bulk

Introduction

This document covers the way the VTexplain tool can be used to evaluate if Vitess is compatible with a list of SQL statements. Enabling the evaluation of if queries from an existing application that accesses a MySQL database are generally Vitess-compatible. If there are any issues identified they can be used to target any necessary application changes needed for a successful migration from MySQL to Vitess.

Prerequisites

You can find a prebuilt binary version of the VTexplain tool in the most recent release of Vitess.

You can also build the `vtexplain` binary in your environment. To build this binary, refer to the Build From Source guide.

Overview

To analyze multiple SQL queries and determine how, or if, Vitess executes each statement, follow these steps:

1. Gather the queries from your current MySQL database environment
2. Filter out specific queries
3. Populate fake values for your queries
4. Run the VTexplain tool via a script
5. Add your SQL schema
6. Add your VSchema to the output file
7. Run the VTexplain tool and capture the output
8. Check your output for errors

1. Gather the queries from your current MySQL database environment

These queries should be most, if not all, of the queries that are sent to your current MySQL database over an extended period of time. You may need to record your queries for days or weeks depending on the nature of your application(s) and workload. You will need to normalize the queries you will be analyzing. Depending on the scope and complexity of your applications you may have a few hundred to thousands of distinct normalized queries. To obtain normalized queries you can use common MySQL monitoring tools like VividCortex, Monyog or PMM.

It is also possible to use the MySQL general query log feature to capture raw queries and then normalize it using post-processing.

2. Filter out specific queries

Remove from your list any unsupported queries or queries from non-application sources. The following are examples of queries to remove are:

- `LOCK/UNLOCK TABLES` - These likely come from schema management tools, which VTGate obviates.
- `FLUSH/PURGE LOGS` - Vitess performs its own log management.
- `performance_schema queries` - These queries are not supported by Vitess.
- `BEGIN/COMMIT` - Vitess supports these statements, but VTexplain does not.

The following is an example pipeline to filter out these specific queries:

```
cat queries.txt \
 | grep -v performance_schema \
 | grep -v information_schema \
 | grep -v @@ \
 | grep -v "SELECT ? $" \
 | grep -v "PURGE BINARY" \
 | grep -v "^SET" \
 | grep -v "^EXPLAIN" \
 | grep -v ^query \
 | grep -v ^BEGIN \
 | grep -v ^COMMIT \
 | grep -v ^FLUSH \
 | grep -v ^LOCK \
 | grep -v ^UNLOCK \
 | grep -v mysql > queries_for_vtexplain.txt
```

3. Populate fake values for your queries

Once the queries are normalized in prepared statement style, populate fake values to allow VTexplain to run properly. This is because `vtexplain` operates only on concrete (or un-normalized) queries. Doing this by textual substitution is shown below and typically requires some trial and error. An alternative is to use a MySQL monitoring tool. This tool sometimes has a feature where it can provide one concrete query example for every normalized query form, which is ideal for this purpose.

If you need to use textual substitution to obtain your concrete queries, the following is an example pipeline you can run:

```
cat queries.txt \
 | perl -p -e 's#\? = \?#1 = 1#g' \
 | perl -p -e 's#= \?#="1"#g' \
 | perl -p -e 's#LIMIT \?#LIMIT 1#g' \
 | perl -p -e 's#\> \?#> "1"#g' \
 | perl -p -e 's#IN \(\?\)#IN (1)#g' \
 | perl -p -e 's#\? AS ONE#1 AS ONE#g' \
 | perl -p -e 's#BINARY \?#BINARY \"1\"#g' \
 | perl -p -e 's#\< \?#< "2"#g' \
 | perl -p -e 's#, \?#, "1"#g' \
 | perl -p -e 's#VALUES \(...\)#VALUES \(1,2\)#g' \
 | perl -p -e 's#IN \(\.\.\.\)#IN \(1,2\)#g' \
 | perl -p -e 's#\- \? #\- 50 #g' \
 | perl -p -e 's#BETWEEN \? AND \?#BETWEEN 1 AND 10#g' \
 | perl -p -e 's#LIKE \? #LIKE \"1\" #g' \
 | perl -p -e 's#OFFSET \?#OFFSET 1#g' \
 | perl -p -e 's#\?, \.\.\.#\"1\", \"2\"#g' \
 | perl -p -e 's#\/ \? #\/ \"1\" #g' \
 | perl -p -e 's#THEN \? ELSE \?#THEN \"2\" ELSE \"3\"#g' \
 | perl -p -e 's#THEN \? WHEN#THEN \"4\" WHEN#g' \
 | perl -p -e 's#SELECT \? FROM#SELECT \"6\" FROM#g' \
 | perl -p -e 's#SELECT \?  AS#SELECT id AS#g' \
 | perl -p -e 's#\`DAYOFYEAR\` \(\?\)#DAYOFYEAR \("2020-01-20"\)#g' \
```

```
 | perl -p -e 's#YEAR \(\?\)#YEAR \("2020-01-01"\)#g' \
 | grep -v mysql > queries_for_vtexplain.txt
```

4. Run the VTexplain tool via a script

In order to analyze every query in your list, create and run a script. The following is an example Python script that assumes a sharded database with 4 shards. You can adjust this script to match your individual requirements.

```
$ cat testfull.py
for line in open("queries_for_vtexplain.txt", "r").readlines():
    sql = line.strip()
    print("vtexplain -schema-file schema.sql -vschema-file vschema.json -shards 4 -sql
        '%s'" % sql)
x
$ python testfull.py > run_vtexplain.sh
```

An alternative method is to use the `-sql-file` option for `vtexplain` to pass the whole file to a single vtexplain invocation. This is much more efficient, but we have found that it can be easier to find errors if you perform one `vtexplain` invocation per SQL query.

If you choose to use the single invocation, it would look something like:

```
$ vtexplain -schema-file schema.sql -vschema-file vschema.json -shards 4 -sql-file
   queries_for_vtexplain.txt
```

5. Add your SQL schema to the output file

Add your proposed SQL schema to the file created by the script (e.g. schema.sql). The following is an example SQL schema:

```
$ cat schema.sql
CREATE TABLE `user` (
  `user_id` bigint(20) NOT NULL,
  `name` varchar(128) DEFAULT NULL,
  `balance` decimal(13,4) DEFAULT NULL,
  PRIMARY KEY (`user_id`),
  KEY `balance` (`balance`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

6. Add your VSchema

Add your VSchema to the file created by the script: in this example, the file is named `schema.json`. The following is an example VSchema to match the example SQL schema above.

```
$ cat vschema.json
{
    "ks1": {
        "sharded": true,
        "vindexes": {
            "hash": {
                "type": "hash"
            }
```

```
        },
        "tables": {
            "user": {
                "column_vindexes": [
                    {
                        "column": "user_id",
                        "name": "hash"
                    }
                ]
            }
        }
    }
}
```

Note that unlike the VSchema used in Vitess, e.g. in `vtctlclient GetVSchema` and `vtctlclient ApplyVSchema`, the format required by `vtexplain` differs slightly. There is an extra level of JSON objects at the top-level of the JSON format to allow you to have a single file that represents the VSchema for multiple Vitess keyspaces. In the above example, there is just a single keyspace called `ks1`.

7. Run the VTexplain tool and capture the output

This step will generate the output you need to analyze to determine what queries may have issues with your proposed VSchema. It may take a long time to finish if you have a number of queries.

```
$ sh -x run_vtexplain.sh 2> vtexplain.output
```

8. Check your output

Once you have your full output in vtexplain.output, use `grep` to search for the string "ERROR" to review any issues that VTExplain found.

Example: Scatted across shards In the following example, VTGate scatters the example query across both shards, and then aggregates the query results.

```
$ vtexplain -schema-file schema.sql -vschema-file vschema.json -shards 2 -sql 'SELECT *
   FROM user;'
----------------------------------------------------------------------
SELECT * FROM user

1 ks1/-40: SELECT * FROM user limit 10001
1 ks1/40-80: SELECT * FROM user limit 10001
1 ks1/80-c0: SELECT * FROM user limit 10001
1 ks1/c0-: SELECT * FROM user limit 10001
----------------------------------------------------------------------
```

This is not an error, but illustrates a few things about the query:

- The query of this type will be scattered across all 4 the shards, given the schema and VSchema.
- The phases of the scatter operation will occur in parallel. This is because the number `1` on the left-hand-side of the output indicates the ordering of the operations in time. The same number indicates parallel processing.
- The implicit Vitess row limit of 10000 rows is also seen, even though that was not present in the original query.

Example: Query returns an error The following query produces an error because Vitess does not support the `AVG` function for scatter queries across multiple shards.

```
$ vtexplain -schema-file schema.sql -vschema-file vschema.json -shards 4 -sql 'SELECT
   AVG(balance) FROM user;'
ERROR: vtexplain execute error in 'SELECT AVG(balance) FROM user': unsupported: in scatter
   query: complex aggregate expression
```

Example: Targeting a single shard The following query only targets a single shard because the query supplies the sharding key.

```
$ vtexplain -schema-file schema.sql -vschema-file vschema.json -shards 2 -sql 'SELECT *
   FROM user WHERE user_id = 100;'
----------------------------------------------------------------------
SELECT * FROM user WHERE user_id = 100

1 ks1/80-c0: SELECT * FROM user WHERE user_id = 100 limit 10001
----------------------------------------------------------------------
```

Analyzing a SQL statement

Introduction

This document covers the way Vitess executes a particular SQL statement using the VTExplain tool. This tool works similarly to the MySQL `EXPLAIN` statement.

Prerequisites

You can find a prebuilt binary version of the VTExplain tool in the most recent release of Vitess.

You can also build the `vtexplain` binary in your environment. To build this binary, refer to the Build From Source guide.

Overview

To successfully analyze your SQL queries and determine how Vitess executes each statement, follow these steps:

1. Identify a SQL schema for the statement's source tables
2. Identify a VSchema for the statement's source tables
3. Run the VTExplain tool

If you have a large number of queries you want to analyze for issues, based on a Vschema you've created for your database, you can read through a detailed scripted example here.

1. Identify a SQL schema for tables in the statement

In order to explain a statement, first identify the SQL schema for the tables that the statement uses. This includes tables that a query targets and tables that a DML statement modifies.

Example SQL Schema The following example SQL schema creates two tables, `users` and `users_name_idx`, each of which contain the columns `user_id` and `name`, and define both columns as a composite primary key. The example statements in step 3 include these tables.

```
CREATE TABLE users(
  user_id bigint,
  name varchar(128),
  primary key(user_id)
);

CREATE TABLE users_name_idx(
  user_id bigint,
  name varchar(128),
  primary key(name, user_id)
);
```

2. Identify a VSchema for the statement's source tables

Next, identify a VSchema that contains the Vindexes for the tables in the statement.

The VSchema must use a keyspace name. VTExplain requires a keyspace name for each keyspace in an input VSchema:

```
"keyspace_name": {
    "_comment": "Keyspace definition goes here."
}
```

If no keyspace name is present, VTExplain will return the following error:

```
ERROR: initVtgateExecutor: json: cannot unmarshal bool into Go value of type
   map[string]json.RawMessage
```

Example VSchema The following example VSchema defines a single keyspace `mainkeyspace` and three Vindexes, and specifies vindexes for each column in the two tables `users` and `users_name_idx`. The keyspace name `"mainkeyspace"` precedes the keyspace definition object.

```
{
  "mainkeyspace": {
    "sharded": true,
    "vindexes": {
      "hash": {
        "type": "hash"
      },
      "md5": {
        "type": "unicode_loose_md5",
        "params": {},
        "owner": ""
      },
      "users_name_idx": {
        "type": "lookup_hash",
        "params": {
          "from": "name",
          "table": "users_name_idx",
          "to": "user_id"
        },
        "owner": "users"
      }
```

```
    },
    "tables": {
      "users": {
        "column_vindexes": [
          {
            "column": "user_id",
            "name": "hash"
          },
          {
            "column": "name",
            "name": "users_name_idx"
          }
        ],
        "auto_increment": null
      },
      "users_name_idx": {
        "type": "",
        "column_vindexes": [
          {
            "column": "name",
            "name": "md5"
          }
        ],
        "auto_increment": null
      }
    }
  }
}
```

3. Run the VTExplain tool

To explain a query, pass the SQL schema and VSchema files as arguments to the `VTExplain` command.

Example: Explaining a SELECT query In the following example, the `VTExplain` command takes a `SELECT` query and returns the sequence of queries that Vitess runs in order to execute the query:

```
vtexplain -shards 8 -vschema-file vschema.json -schema-file schema.sql -replication-mode
    "ROW" -output-mode text -sql "SELECT * from users"
----------------------------------------------------------------------
SELECT * from users

1 mainkeyspace/-20: select * from users limit 10001
1 mainkeyspace/20-40: select * from users limit 10001
1 mainkeyspace/40-60: select * from users limit 10001
1 mainkeyspace/60-80: select * from users limit 10001
1 mainkeyspace/80-a0: select * from users limit 10001
1 mainkeyspace/a0-c0: select * from users limit 10001
1 mainkeyspace/c0-e0: select * from users limit 10001
1 mainkeyspace/e0-: select * from users limit 10001

----------------------------------------------------------------------
```

In the example above, the output of `VTExplain` shows the sequence of queries that Vitess runs in order to execute the query. Each line shows the logical sequence of the query, the keyspace where the query executes, the shard where the query executes, and the query that executes, in the following format:

```
[Sequence number] [keyspace]/[shard]: [query]
```

In this example, each query has sequence number `1`, which shows that Vitess executes these in parallel. Vitess automatically adds the `LIMIT 10001` clause to protect against large results.

Example: Explaining an INSERT query In the following example, the `VTExplain` command takes an `INSERT` query and returns the sequence of queries that Vitess runs in order to execute the query:

```
vtexplain -shards 128 -vschema-file vschema.json -schema-file schema.sql -replication-mode
   "ROW" -output-mode text -sql "INSERT INTO users (user_id, name) VALUES(1, 'john')"

----------------------------------------------------------------------
INSERT INTO users (user_id, name) VALUES(1, 'john')

1 mainkeyspace/22-24: begin
1 mainkeyspace/22-24: insert into users_name_idx(name, user_id) values ('john', 1) /*
   vtgate:: keyspace_id:22c0c31d7a0b489a16332a5b32b028bc */
2 mainkeyspace/16-18: begin
2 mainkeyspace/16-18: insert into users(user_id, name) values (1, 'john') /* vtgate::
   keyspace_id:166b40b44aba4bd6 */
3 mainkeyspace/22-24: commit
4 mainkeyspace/16-18: commit

----------------------------------------------------------------------
```

The example above shows how Vitess handles an insert into a table with a secondary lookup Vindex:

- At sequence number `1`, Vitess opens a transaction on shard `22-24` to insert the row into the `users_name_idx` table.
- At sequence number `2`, Vitess opens a second transaction on shard `16-18` to perform the actual insert into the `users` table.
- At sequence number `3`, the first transaction commits.
- At sequence number `4`, the second transaction commits.

Example: Explaining an uneven keyspace In previous examples, we used the `-shards` flag to set up an evenly-sharded keyspace, where each shard covers the same fraction of the keyrange. `VTExplain` also supports receiving a JSON mapping of shard ranges to see how Vitess would handle a query against an arbitrarly-sharded keyspace.

To do this, we first create a JSON file containing a mapping of keyspace names to shardrange maps. The shardrange map has the same structure as the output of running `vtctl FindAllShardsInKeyspace <keyspace>`.

```
{
  "mainkeyspace": {
    "-80": {
      "master_alias": {
        "cell": "test",
        "uid": 00000000100
      },
      "master_term_start_time": {
        "seconds": 1599828375,
        "nanoseconds": 664404881
      },
      "key_range": {
        "end": "gA=="
      },
      "is_master_serving": true
    },
    "80-90": {
      "master_alias": {
```

```
      "cell": "test",
      "uid": 00000000200
    },
    "master_term_start_time": {
      "seconds": 1599828344,
      "nanoseconds": 868327074
    },
    "key_range": {
      "start": "gA==",
      "end": "kA=="
    },
    "is_master_serving": true
  },
  "90-a0": {
    "master_alias": {
      "cell": "test",
      "uid": 00000000300
    },
    "master_term_start_time": {
      "seconds": 1599828405,
      "nanoseconds": 152120945
    },
    "key_range": {
      "start": "kA==",
      "end": "oA=="
    },
    "is_master_serving": true
  },
  "a0-e8": {
    "master_alias": {
      "cell": "test",
      "uid": 00000000400
    },
    "master_term_start_time": {
      "seconds": 1599828183,
      "nanoseconds": 911677983
    },
    "key_range": {
      "start": "oA==",
      "end": "6A=="
    },
    "is_master_serving": true
  },
  "e8-": {
    "master_alias": {
      "cell": "test",
      "uid": 00000000500
    },
    "master_term_start_time": {
      "seconds": 1599827865,
      "nanoseconds": 770606551
    },
    "key_range": {
      "start": "6A=="
    },
    "is_master_serving": true
```

```
    }
}
```

After having saved that to a file called `shardmaps.json`:

```
vtexplain -vschema-file vschema.json -schema-file schema.sql -ks-shard-map shardmaps.json
   -replication-mode "ROW" -output-mode text -sql "SELECT * FROM users; SELECT * FROM users
   WHERE id IN (10, 17, 42, 1000);"
----------------------------------------------------------------------
SELECT * FROM users

1 mainkeyspace/-80: select * from users limit 10001
1 mainkeyspace/80-90: select * from users limit 10001
1 mainkeyspace/90-a0: select * from users limit 10001
1 mainkeyspace/a0-e8: select * from users limit 10001
1 mainkeyspace/e8-: select * from users limit 10001

----------------------------------------------------------------------
SELECT * FROM users WHERE id IN (10, 17, 42, 100000)

1 mainkeyspace/-80: select * from users where id in (10, 17, 42) limit 10001
1 mainkeyspace/80-90: select * from users where id in (100000) limit 10001

----------------------------------------------------------------------
```

See also

- For detailed configuration options for VTExplain, see the VTExplain syntax reference.

VSchema and Query Serving

description: Configuring VSchema for serving queries

Advanced VSchema Properties

With the exception of Multi-Column Vindexes, advanced VSchema Properties do not have DDL constructs. They can only be updated through `vtctld` CLI commands.

Multi-Column Vindexes

Multi-Column Vindexes are useful in the following two use cases:

- Grouping customers by their regions so they can be hosted in specific geographical locations. This may be required for compliance and also to achieve better performance.
- For a multi-tenant system, grouping all rows of a tenant in a separate set of shards. This limits the fan out of queries if searching only for rows that are related to a single tenant.

In both cases the leading column is the region or tenant, and is used to form the first few bits of the `keyspace_id`. The second column is used for the bits that follow. Since Vitess shards by keyrange, this approach will naturally group all rows of a region or tenant within the same shard, or within a group of consecutive shards. Since each shard is its own MySQL cluster, these can then be deployed to different regions as needed.

Please refer to Region-based Sharding for an example on how to use the `region_json` vindex.

Currently, the Vindex gets used for assigning a `keyspace_id` at the time of insert and at the time of resharding. Additional vindexes need to be added to the table for routing query constructs that contain WHERE clauses.

Vitess does not have the capability to route a query based on multiple values of a multi-column vindex in a where clause yet. This feature will be added soon.

Alternate approach You have the option to pre-combine the region and id bits into a single column and use that as an input for a single column vindex. This approach achieves the same goals as a multi-column vindex. Moreover, you avoid having to define additional vindexes for query routing.

The downside of this approach is that it is harder to migrate an id to a different region.

Reference Tables

Sharded databases often need the ability to join their tables with smaller "reference" tables. For example, the `product` table could be seen as a reference table. Other use cases are tables that map static information like zip code to city, etc.

Joining against these tables across keyspaces results in cross-shard joins that may not be very efficient or fast.

Vitess allows you to create a table in a sharded keyspace as a reference table. This means that it will treat the table as having an identical set of rows across all shards. A query that joins a sharded table against such reference tables is then performed locally within each shard.

A reference table should not have any vindex, and is defined in the VSchema as a reference type:

```
{
  "sharded": true,
  "tables": {
    "zip_detail": { "type": "reference" }
  }
}
```

It may become a challenge to keep a reference table correctly updated across all shards. Vitess supports the Materialize feature that allows you to maintain the original table in an unsharded keyspace and automatically propagate changes to that table in real-time across all shards.

Column List

The VSchema allows you to specify the list of columns along with their types for every table. This allows Vitess to make optimization decisions where necessary.

For example, specifying that a column contains numeric data allows VTGate to not request further collation specific information (`weight_string`) if additional sorting is needed after collecting results from all shards.

For example, issuing this query against `customer` would add the `weight_string` column while sending the query to the vttablets:

```
Query - select integer_col from customer order by integer_col;
Plan -
{
  "QueryType": "SELECT",
  "Original": "select integer_col from customer order by integer_col",
  "Instructions": {
    "OperatorType": "Route",
    "Variant": "SelectScatter",
    "Keyspace": {
      "Name": "customer",
      "Sharded": true
    },
```

```
    "FieldQuery": "select integer_col, weight_string(integer_col) from `customer` where 1
       != 1",
    "OrderBy": "0 ASC",
    "Query": "select integer_col, weight_string(integer_col) from `customer` order by
       integer_col asc",
    "Table": "`customer`"
  }
}
```

However, we can modify the VSchema as follows:

```
    "customer": {
      "column_vindexes": [{
        "column": "customer_id",
        "name": "hash"
      }],
      "auto_increment": {
        "column": "customer_id",
        "sequence": "product.customer_seq"
      },
      "columns": [{
        "name": "integer_col",
        "type": "INT16"
      }]
    }
```

Re-issuing the same query will now not use `weight_string`:

```
Query - select integer_col from customer order by integer_col;
Plan -
{
  "QueryType": "SELECT",
  "Original": "select integer_col from customer order by integer_col",
  "Instructions": {
    "OperatorType": "Route",
    "Variant": "SelectScatter",
    "Keyspace": {
      "Name": "customer",
      "Sharded": true
    },
    "FieldQuery": "select integer_col from `customer` where 1 != 1",
    "OrderBy": "0 ASC",
    "Query": "select integer_col from `customer` order by integer_col asc",
    "Table": "`customer`"
  }
}
```

Specifying columns against tables also allows VTGate to resolve ambiguous naming of columns against the right tables.

Authoritative List If you have listed all columns of a table in the VSchema, you can add the `column_list_authoritative` flag to the table:

```
    "customer": {
      "column_vindexes": [{
        "column": "customer_id",
        "name": "hash"
      }],
      "auto_increment": {
```

```
        "column": "customer_id",
        "sequence": "product.customer_seq"
      },
      "columns": [{
        "name": "uname",
        "type": "VARCHAR"
      }],
      "column_list_authoritative": true
    }
```

This flag causes VTGate to automatically expand expressions like `select *` or insert statements that don't specify the column list.

The caveat about using this feature is that you have to keep this column list in sync with the underlying schema.

In the future, Vitess will allow you to pull this information from the vttablets and automatically keep it up-to-date.

Routing Rules

Routing Rules are an advanced method of redirecting queries meant for one table to another. They are just pointers and are analogous to symbolic links in a file system. You should generally not have to use routing rules in Vitess.

Workflows like `MoveTables` make use of routing rules to create the existence of the target tables and manage traffic switch from source to target by manipulating these routing rules.

For more information, please refer to the Routing Rules section.

Lookup as Primary Vindex

It is likely that a customer order goes through a life cycle of events. This would best be represented in a separate `corder_event` table that will contain a `corder_id` column as a foreign key into `corder.corder_id`. It would also be beneficial to co-locate the event rows with their associated order.

Just like we shared the `hash` vindex between `customer` and `corder`, we can share `corder_keyspace_idx` between `corder` and `corder_event`. We can also make it the Primary Vindex for `corder_event`. When an order is created, the lookup row for it is also created. Subsequently, an insert into `corder_event` will request the vindex to compute the `keyspace_id` for that `corder_id`, and that will succeed because the lookup entry for it already exists. This is where the significance of the owner table comes into play: The owner table creates the entries, whereas other tables only read those entries.

Inserting a `corder_event` row without creating a corresponding `corder` entry will result in an error. This behavior is in line with the traditional foreign key constraint enforced by relational databases.

Sharing the lookup vindex also has the additional benefit of saving space because we avoid creating separate entries for the new table.

We start with creating the sequence table in the `product` keyspace.

Schema:

```
create table corder_event_seq(id bigint, next_id bigint, cache bigint, primary key(id))
   comment 'vitess_sequence';
insert into corder_event_seq(id, next_id, cache) values(0, 1, 3);
```

VSchema:

```
    "corder_event_seq": { "type": "sequence" }
```

We then create the `corder_event` table in `customer`:

```
create table corder_event(corder_event_id bigint, corder_id bigint, ename varchar(128),
   primary key(corder_id, corder_event_id));
```

In the VSchema, there is no need to create a vindex because we are going to reuse the existing one:

```
    "corder_event": {
      "column_vindexes": [{
        "column": "corder_id",
        "name": "corder_keyspace_idx"
      }],
      "auto_increment": {
        "column": "corder_event_id",
        "sequence": "product.corder_event_seq"
      }
    }
```

Alternate VSchema DDL:

```
alter vschema add sequence product.corder_event_seq;
alter vschema on customer.corder_event add vindex corder_keyspace_idx(corder_id);
alter vschema on customer.corder_event add auto_increment corder_event_id using
   product.corder_event_seq;
```

We can now insert rows in `corder_event` against rows in `corder`:

```
mysql> insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
Query OK, 5 rows affected (0.04 sec)

mysql> insert into corder_event(corder_id, ename) values(1, 'paid'), (5, 'delivered');
Query OK, 2 rows affected (0.01 sec)

mysql> insert into corder_event(corder_id, ename) values(6, 'expect failure');
ERROR 1105 (HY000): vtgate: http://sougou-lap1:12345/: execInsertSharded:
   getInsertShardedRoute: could not map [INT64(6)] to a keyspace id
```

As expected, inserting a row for a non-existent order results in an error.

Reversible Vindexes In Vitess, it is insufficient for a table to only have a Lookup Vindex. This is because it is not practical to reshard such a table. The overhead of performing a lookup before redirecting every row event to a new shard would be prohibitively expensive.

To overcome this limitation, we must add a column with a non-lookup vindex, also known as Functional Vindex to the table. By rule, the Primary Vindex computes the keyspace id of the row. This means that the value of the column should also be such that it yields the same keyspace id.

A Reversible Vindex is one that can back-compute the column value from a given keyspace id. If such a vindex is used for this new column, then Vitess will automatically perform this work and fill in the correct value for it. The list of vindex properties, like Functional, Reversible, etc. are listed in the Vindexes Reference.

In other words, adding a column with a vindex that is both Functional and Reversible allows Vitess to auto-fill the values, thereby avoiding any impact to the application logic.

The `binary` vindex is one that yields the input value itself as the `keyspace_id`, and is naturally reversible. Using this Vindex will generate the `keyspace_id` as the column value. The modified schema for the table will be as follows:

```
create table corder_event(corder_event_id bigint, corder_id bigint, ename varchar(128),
   keyspace_id varbinary(10), primary key(corder_id, corder_event_id));
```

We create a vindex instantiation for `binary`:

```
    "binary": {
      "type": "binary"
    }
```

Modify the table VSchema:

```
    "corder_event": {
      "column_vindexes": [{
        "column": "corder_id",
        "name": "corder_keyspace_idx"
      }, {
        "column": "keyspace_id",
        "name": "binary"
      }],
      "auto_increment": {
        "column": "corder_event_id",
        "sequence": "product.corder_event_seq"
      }
    }
```

Alternate VSchema DDL:

```
alter vschema on customer.corder_event add vindex `binary`(keyspace_id) using `binary`;
```

Note that `binary` needs to be backticked because it is a keyword.

After these modifications, we can now observe that the `keyspace_id` column is getting automatically populated:

```
mysql> insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
Query OK, 5 rows affected (0.01 sec)

mysql> insert into corder_event(corder_id, ename) values(1, 'paid'), (5, 'delivered');
Query OK, 2 rows affected (0.01 sec)

mysql> select corder_event_id, corder_id, ename, hex(keyspace_id) from corder_event;
+-----------------+-----------+-----------+------------------+
| corder_event_id | corder_id | ename     | hex(keyspace_id) |
+-----------------+-----------+-----------+------------------+
|               1 |         1 | paid      | 166B40B44ABA4BD6 |
|               2 |         5 | delivered | D2FD8867D50D2DFE |
+-----------------+-----------+-----------+------------------+
2 rows in set (0.00 sec)
```

There is no support for backfilling the reversible vindex column yet. This will be added soon.

{{< info >}} The original `keyspace_id` for all these rows came from `customer_id`. Since `hash` is also a reversible vindex, reversing the `keyspace_id` using `hash` will yield the `customer_id`. We could instead leverage this knowledge to replace `keyspace_id+binary` with `customer_id+hash`. Vitess will auto-populate the correct value. Using this approach may be more beneficial because `customer_id` is a value the application can understand and make use of. {{< /info >}}

Non-Unique Lookup Vindexes

The `oname` column in `corder` can contain duplicate values. There may be a need in the application to frequently search by this column:

```
select * from corder where oname='gift'
```

To prevent this query from resulting in a full scatter, we will need to create a lookup vindex for it. But this time, it will need to be non-unique. However, the fact that duplicates are allowed leads to a complication with the lookup table approach. Let us look at the insert query:

```
insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
```

We see that `customer_id 1` has two rows where the `oname` is `gift`. If we try to create entries for those two in a lookup table, they would be identical:

```
+-----------+--------------+
| oname | hex(keyspace_id) |
+-----------+--------------+
| gift  | 166B40B44ABA4BD6 | (corder_id=1)
| gift  | 166B40B44ABA4BD6 | (corder_id=2)
+-----------+--------------+
```

To disambiguate this situation, non-unique lookup vindexes require you to add additional columns to the lookup table. They are typically the Primary Key of the main table. For the sake of demonstration, let us create this as a sharded table in the `customer` keyspace:

```
create table oname_keyspace_idx(oname varchar(128), corder_id bigint, keyspace_id
   varbinary(10), primary key(oname, corder_id));
```

Note that the primary key includes the `oname` column as well as the `corder_id` column.

Because `oname` is a text column, the recommended Primary Vindex for it would be `unicode_loose_md5`, which also requires a vindex instantiation:

"vindexes" section:

```
    "unicode_loose_md5": {
      "type": "unicode_loose_md5"
    }
```

"tables" section:

```
    "oname_keyspace_idx": {
      "column_vindexes": [{
        "column": "oname",
        "name": "unicode_loose_md5"
      }]
    }
```

The lookup vindex should reference these new columns as follows:

```
    "oname_keyspace_idx": {
      "type": "consistent_lookup",
      "params": {
        "table": "customer.oname_keyspace_idx",
        "from": "oname,corder_id",
        "to": "keyspace_id"
      },
      "owner": "corder"
    }
```

{{< info >}} This Vindex could also be seen as a multi-column Unique Lookup Vindex: For a given pair of `oname,corder_id` as input, the result can only yield a single `keyspace_id`. However, the `consistent_lookup` vindex functionality only supports resolution using the first column `oname`. In the future, we may add the ability to use both columns as input if they are present in the `where` clause. This may result in the merger of `consistent_lookup` with a multi-column version of `consistent_lookup_unique` that can also perform non-unique lookups on a subset of the inputs. {{< /info >}}

Finally, we tie the associated columns in `corder` to the vindex:

```
    "corder": {
      "column_vindexes": [{
        "column": "customer_id",
        "name": "hash"
```

```
    }, {
      "column": "corder_id",
      "name": "corder_keyspace_idx"
    }, {
      "columns": ["oname", "corder_id"],
      "name": "oname_keyspace_idx"
    }],
    "auto_increment": {
      "column": "corder_id",
      "sequence": "product.corder_seq"
    }
  }
```

Alternate VSchema DDL:

```
alter vschema on customer.oname_keyspace_idx add vindex unicode_loose_md5(oname) using
   unicode_loose_md5;
alter vschema on customer.corder add vindex oname_keyspace_idx(oname,corder_id) using
   consistent_lookup with owner=`corder`, table=`customer.oname_keyspace_idx`,
   from=`oname,corder_id`, to=`keyspace_id`;
```

We can now look at the effects of this change:

```
mysql> insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
Query OK, 5 rows affected (0.03 sec)

mysql> use `customer:-80`;
Database changed
mysql> select oname, corder_id, hex(keyspace_id) from oname_keyspace_idx;
+-------+-----------+------------------+
| oname | corder_id | hex(keyspace_id) |
+-------+-----------+------------------+
| gift  |         1 | 166B40B44ABA4BD6 |
| gift  |         2 | 166B40B44ABA4BD6 |
| work  |         3 | 06E7EA22CE92708F |
+-------+-----------+------------------+
3 rows in set (0.00 sec)

mysql> use `customer:80-`;
Database changed
mysql> select oname, corder_id, hex(keyspace_id) from oname_keyspace_idx;
+----------+-----------+------------------+
| oname    | corder_id | hex(keyspace_id) |
+----------+-----------+------------------+
| personal |         4 | 4EB190C9A2FA169C |
| personal |         5 | D2FD8867D50D2DFE |
+----------+-----------+------------------+
2 rows in set (0.00 sec)
```

We can see that the lookup table is following its own sharding scheme and distributing its rows according to the value of the `oname` column.

Deleting one of the `corder` rows results in the corresponding lookup row being deleted:

```
mysql> delete from corder where corder_id=1;
Query OK, 1 row affected (0.00 sec)

mysql> select oname, corder_id, hex(keyspace_id) from oname_keyspace_idx where oname='gift';
```

```
+-------+-----------+------------------+
| oname | corder_id | hex(keyspace_id) |
+-------+-----------+------------------+
| gift  |         2 | 166B40B44ABA4BD6 |
+-------+-----------+------------------+
1 row in set (0.00 sec)
```

{{< info >}} You would typically have to create a MySQL non-unique index on `oname` for queries to work efficiently. While these vindexes and indexes improve read performance, the trade-off is that they also increase storage requirements and amplify writes when inserting rows. {{< /info >}}

CreateLookupVindex To create such a lookup vindex on a real Vitess cluster, you can use the following instructions:

Save the following json into a file, say `oname_keyspace_idx.json`:

```
{
  "sharded": true,
  "vindexes": {
    "oname_keyspace_idx": {
      "type": "consistent_lookup",
      "params": {
        "table": "customer.oname_keyspace_idx",
        "from": "oname,corder_id",
        "to": "keyspace_id"
      },
      "owner": "corder"
    }
  },
  "tables": {
    "corder": {
      "column_vindexes": [{
        "columns": ["oname", "corder_id"],
        "name": "oname_keyspace_idx"
      }]
    }
  }
}
```

And issue the vtctlclient command:

```
$ vtctlclient -server <vtctld_grpc_port> CreateLookupVindex -tablet_types=REPLICA customer
   "$(cat oname_keyspace_idx.json)"
```

The workflow will automatically create the necessary Primary Vindex entries for `oname_keyspace_idx` knowing that it is sharded.

After the backfill is done, you should clean up the workflow. More detailed instructions are available in the CreateLookupVindex Reference

Overview

One of the goals for Vitess is to provide a unified view for a large number of MySQL clusters distributed across multiple data centers and regions.

Vitess achieves this goal by allowing the application to connect to any VTGate server, and that server gives you the semblance of being connected to a single MySQL server. The metadata that maps the logical view to the physical MySQL servers is stored in the topology.

In this logical view, a Vitess keyspace is the equivalent of a MySQL database. In many cases, this is a one-to-one mapping where a keyspace directly corresponds to a physical MySQL server with a single database. However, a Vitess keyspace can also be sharded. If so, a single keyspace would map to multiple MySQL servers behind the scenes.

The topology is typically spread across multiple Topo Servers: The Global Topo server contains global information, like the list of keyspaces, shards and cells. This information gets deployed into cell-specific topo servers. Each cell-specific Topo Server contains additional information about vttablets and MySQL servers running in that cell. With this architecture, an outage in one cell does not affect other cells.

The topo also stores a VSchema for each keyspace. For an unsharded keyspace, the vschema is a simple list of table names. If a keyspace is sharded, then it must contain additional metadata about the sharding scheme for each table, and how they relate to each other. When a query is received by VTGate, the information in the vschema is used to make decisions about how to serve the query. In some cases, it will result in the query being routed to a single shard. In other cases, it could result in the query being sent to all shards, etc.

This guide explains how to build vschemas for Vitess keyspaces.

Demo To illustrate the various features of the VSchema, we will make use of the demo app. After installing Vitess, you can launch this demo by running `go run demo.go`. Following this, you can visit http://localhost:8000 to view the tables, issue arbitrary queries, and view their effects.

Alternatively, you can also connect to Vitess using a MySQL client: `mysql -h 127.0.0.1 -P 12348`.

The demo models a set of tables that are similar to those presented in the Getting Started guide, but with more focus on the VSchema.

Note that the demo brings up a test process called vtcombo (instead of a real Vitess cluster), which is functionally equivalent to all the components of Vitess, but within a single process.

You can also use the demo app to follow the steps of this user guide. If so, you can start by emptying out the files under `schema/product` and `schema/customer`, and incrementally making the changes presented in the steps that follow.

VSchema DDL The demo describes the VSchema JSON syntax. Many of the changes can be executed by issuing special DDL commands that Vitess understands. Wherever applicable, we have provided the equivalent DDL construct you could apply if you were running a live system. All the DDLs are also listed in the `vschema_ddls.sql` file.

It is generally recommended that you get familiar with the JSON syntax as it will be useful for troubleshooting if something does not work as intended.

Pictorial Summary

The following two diagrams highlight some of the relationships that exist between VSchema elements and the MySQL tables.

product and customer

corder

Sequences

The sharded `customer` table we created did not have an auto-increment column. The Vitess Sequence feature can be used to emulate the same behavior as MySQL's auto-increment. A Vitess sequence is a single row unsharded tablet that keeps track of ids issued so far. Additionally, a configurable number of values can be cached by vttablet to minimize round trips into MySQL.

We will create the sequence table in the unsharded `product` keyspace as follows:

```
create table customer_seq(id bigint, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
insert into customer_seq(id, next_id, cache) values(0, 1, 3);
```

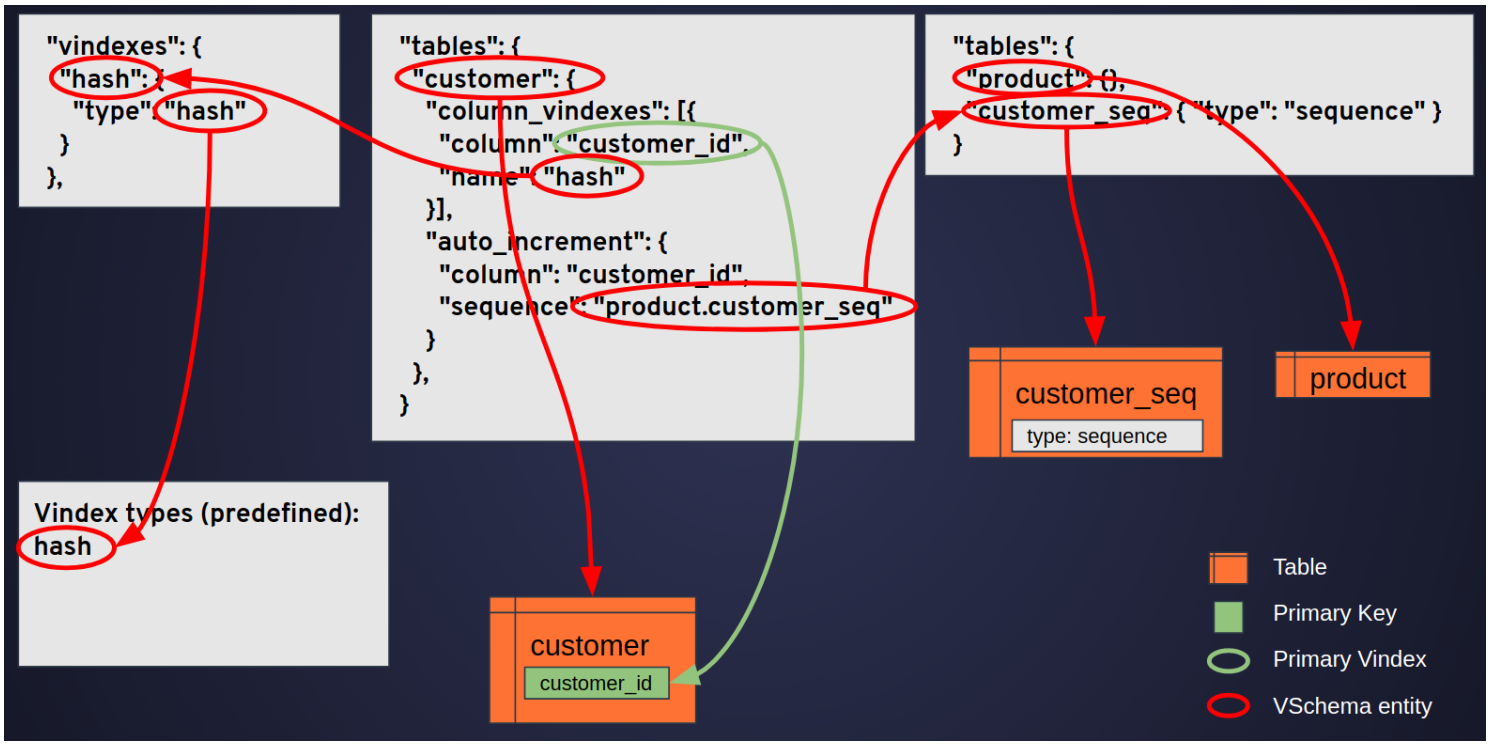


Figure 13: vschema1

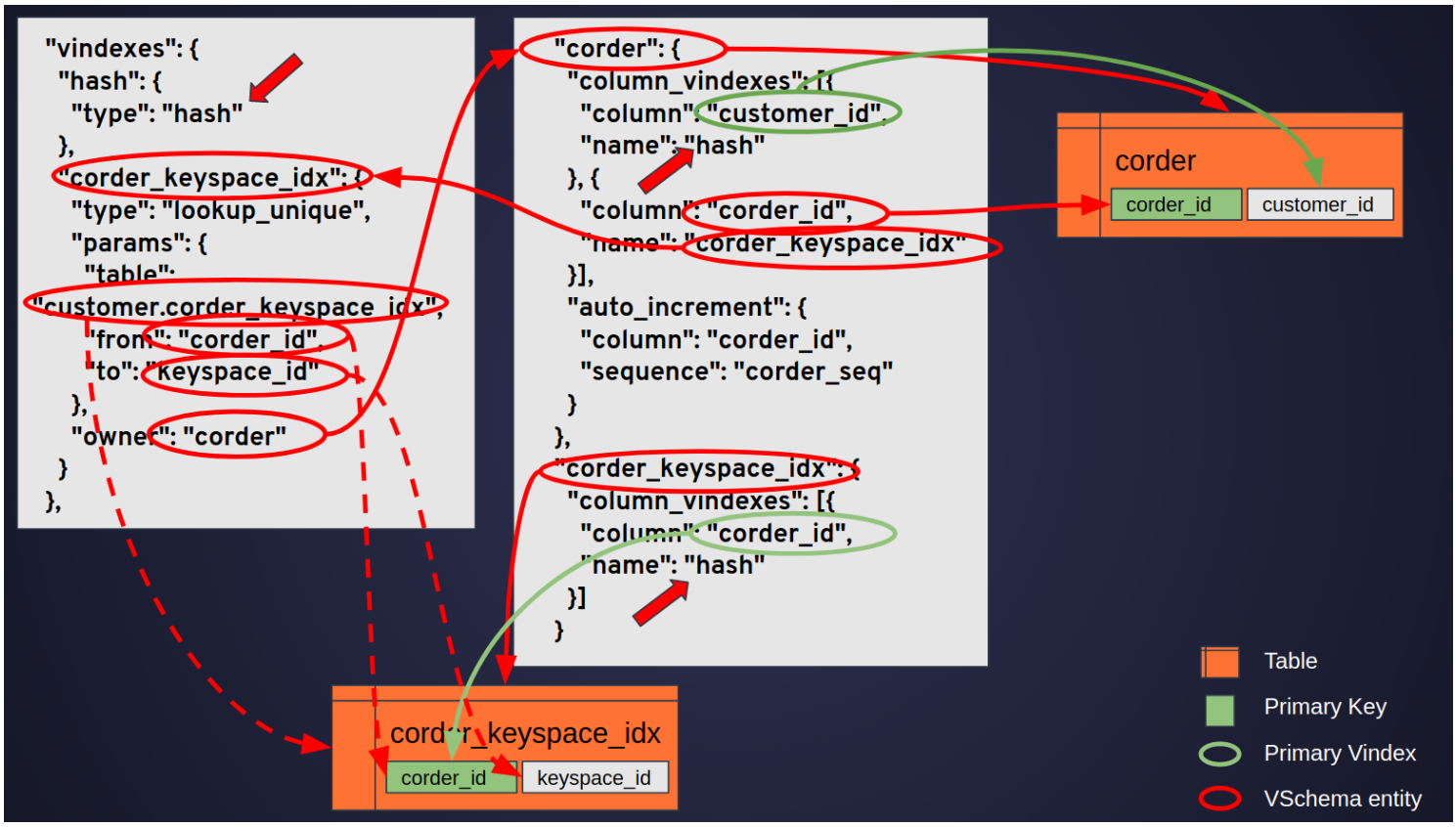


Figure 14: vschema2

Note the special comment `vitess_sequence`. This instructs vttablet that this is a special table.

The table needs to be pre-populated with a single row where:

- `id` must always be 0
- `next_id` should be set to the next (starting) value of the sequence
- `cache` is the number of values to cache before updating the table for the next value. This value should be set to a fairly large number like 1000. We have set the value to `3` mainly to demonstrate how the feature works.

Since this is a special table, we have to inform the vschema by giving it a `sequence` type.

```
    "customer_seq": { "type": "sequence" }
```

Once setup this way, you can use the special `select next` syntax to generate values from this sequence:

```
mysql> select next 2 values from customer_seq;
+---------+
| nextval |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)

mysql> select next 1 values from customer_seq;
+---------+
| nextval |
+---------+
|       3 |
+---------+
1 row in set (0.00 sec)
```

The construct returns the first of the N values generated.

However, this is insufficient to emulate MySQL's auto-increment behavior. To achieve this, we have to inform the VSchema that the `customer_id` column should use this sequence to generate values if no value is specified. This is done by adding the following section to the `customer` table:

```
      "auto_increment": {
        "column": "customer_id",
        "sequence": "product.customer_seq"
      }
```

Alternate VSchema DDL:

```
alter vschema add sequence product.customer_seq;
alter vschema on customer.customer add auto_increment customer_id using
   product.customer_seq;
```

With this, you can insert into `customer` without specifying the `customer_id`:

```
mysql> insert into customer(uname) values('alice'),('bob'),('charlie'),('dan'),('eve');
Query OK, 5 rows affected (0.03 sec)

mysql> use `customer:-80`;
Database changed
mysql> select * from customer;
+-------------+---------+
| customer_id | uname   |
+-------------+---------+
|           1 | alice   |
```

```
|           2 | bob     |
|           3 | charlie |
|           5 | eve     |
+-------------+---------+
4 rows in set (0.00 sec)

mysql> use `customer:80-`;
Database changed
mysql> select * from customer;
+-------------+-------+
| customer_id | uname |
+-------------+-------+
|           4 | dan   |
+-------------+-------+
1 row in set (0.00 sec)
```

Sharded Keyspace

A sharded keyspace allows you to split a large database into smaller parts by distributing the rows of each table into different shards. In Vitess, each shard is assigned a `keyrange`. Every row has a keyspace id, and this value decides the shard in which the row lives. For key-value stores, the keyspace id is dictated by the value of the key, also known as the sharding key. In Vitess, this is known as the Primary Vindex. But it differs from a sharding key in the following ways:

- Any column or a set of columns can be chosen to be the primary vindex.
- The Vindex also decides the sharding function that controls how the data is distributed.
- The sharding function is pluggable, allowing for user-defined sharding schemes.

Vitess provides many predefined vindex types. The most popular ones are:

- `hash`: for numbers
- `unicode_loose_md5`: for text columns
- `binary_md5`: for binary columns

In our example, we are going to designate `customer` as a sharded keyspace, and create a `customer` table in it. The schema for the table is as follows:

```
create table customer(customer_id bigint, uname varchar(128), primary key(customer_id));
```

In the VSchema, we need to designate which column should be the Primary Vindex, and choose the vindex type for it. The `customer_id` column seems to be the natural choice. Since it is a number, we will choose `hash` as the vindex type:

```
{
  "sharded": true,
  "vindexes": {
    "hash": {
      "type": "hash"
    }
  },
  "tables": {
    "customer": {
      "column_vindexes": [{
        "column": "customer_id",
        "name": "hash"
      }]
    }
  }
}
```

In the above section, we are instantiating a vindex named `hash` from the vindex type `hash`. Such instantiations are listed in the `vindexes` section of the vschema. The tables are expected to refer to the instantiated name. There are a few reasons why this additional level of indirection is necessary:

- As we will see later, vindexes can be instantiated with different input parameters. In such cases, they have to have their own distinct names.
- Vindexes can be shared by tables, and this has special meaning. We will cover this in a later section.
- Vindexes can also be referenced as if they were tables and can be used to compute the keyspace id for a given input.

The `column_vindexes` section is a list. This is because a table can have multiple vindexes. If so, the first vindex in the list must be the Primary Vindex. More information about vindexes can be found in the Vindex Reference.

Alternate VSchema DDL:

```
alter vschema on customer.customer add vindex hash(customer_id) using hash;
```

The DDL creates the `hash` vindex under the `vindexes` section, the `customer` table under the `tables` section, and associates the `customer_id` column to `hash`. For sharded keyspaces, the only way to create a table is using the above construct. This is because a primary vindex is mandatory for sharded tables.

{{< info >}} Every sharded table must have a Primary Vindex. A Primary Vindex must be instantiated from a vindex type that is Unique. `hash`, `unicode_loose_md5` and `binary_md5` are unique vindex types. {{< /info >}}

The demo brings up the `customer` table as two shards: `-80` and `80-`. For a `hash` vindex, input values of 1, 2 and 3 fall in the `-80` range, and 4 falls in the `80-` range. Restarting the demo with the updated configs should allow you to perform the following:

```
mysql> insert into customer(customer_id,uname) values(1,'alice'),(4,'dan');
Query OK, 2 rows affected (0.00 sec)

mysql> use `customer:-80`;
Database changed
mysql> select * from customer;
+-------------+-------+
| customer_id | uname |
+-------------+-------+
|           1 | alice |
+-------------+-------+
1 row in set (0.00 sec)

mysql> use `customer:80-`;
Database changed
mysql> select * from customer;
+-------------+-------+
| customer_id | uname |
+-------------+-------+
|           4 | dan   |
+-------------+-------+
1 row in set (0.00 sec)
```

You will notice that we used a special shard targeting construct: `use customer:-80`. Vitess allows you to use this hidden database name to bypass its routing logic and directly send queries to a specific shard. Using this construct, we are able to verify that the rows went to different shards.

At the time of insert, the Primary Vindex is used to compute and assign a keyspace id to each row. This keyspace id gets used to decide where the row will be stored. Although a keyspace id is not explicitly stored anywhere, it must be seen as an unchanging property of that row; as if there was an invisible column for it.

Consequently, you cannot make changes to a row that can cause the keyspace id to change. Such a change will be supported in the future through a shard move operation. Trying to change the value of a Primary Vindex results in an error:

```
mysql> update customer set customer_id=2 where customer_id=1;
ERROR 1235 (HY000): vtgate: http://sougou-lap1:12345/: unsupported: You can't update
   primary vindex columns. Invalid update on vindex: hash
```

A Primary Vindex can also be used to find rows if referenced in a where clause:

```
mysql> select * from customer where customer_id=1;
+-------------+-------+
| customer_id | uname |
+-------------+-------+
|           1 | alice |
+-------------+-------+
1 row in set (0.00 sec)
```

If you run the above query in the demo app, the panel on the bottom right will show that the query was executed only on one shard.

On the other hand, the query below will get sent to all shards because there is no where clause:

```
mysql> select * from customer;
+-------------+-------+
| customer_id | uname |
+-------------+-------+
|           4 | dan   |
|           1 | alice |
+-------------+-------+
2 rows in set (0.01 sec)
```

{{< info >}} There is no implicit or predictable ordering for rows that are gathered from multiple shards. If a specific order is required, the query must include an `order by` clause. {{< /info >}}

Sharding Guidelines

The following guidelines are not set in stone. They mainly establish a framework for making decisions.

Why There was a time when sharding used to be a line that one should avoid crossing for as long as possible. However, with Vitess considerably reducing the pain of sharding, we can look at leveraging some of its benefits much sooner than when a machine runs out of capacity:

- Smaller blast radius: If a shard goes down, the outage affects a smaller percentage of the users.
- Improved resource utilization: It is difficult to pack large instances of servers efficiently across machines. It is much easier to utilize the capacity of the existing hardware if the shard sizes are relatively small. Orchestration systems like Kubernetes further facilitate such utilization.
- Reduced contention: MySQL itself runs a lot better when instance sizes are small. There is less internal contention, replicas tend to keep up more easily with their master, etc.
- Improved maintenance: Operations like schema deployment can happen in parallel across all shards and finish much sooner than usual.

There is a general worry that the complexity of deployment increases with the number of servers. However, this becomes a static cost once the necessary automation and monitoring is put in place.

Why not There are also reasons why you may want to avoid sharding. The main reason is that you may introduce inefficiencies by splitting data that would have been better off if it stayed together. Or, if your database is extremely small.

However, if you reach a point where the data is starting to grow, sharding may become inevitable.

Moving Tables Typically, the first step you may perform is to split your database by moving some tables on to other databases. In Vitess parlance, we call this as splitting off keyspaces. The MoveTables workflow allows you to perform this with minimal impact to the application.

Resharding Beyond a certain point, it may not make sense to separate tables that are strongly related to each other. This is when resharding comes into play. Choosing the "sharding key" is often intuitively obvious.

If you analyze the query pattern in the application, the query with the highest QPS will dictate the sharding key (or Primary Vindex). In our example below, we will be choosing `customer_id` as the Primary Vindex for the `customer` table.

If there are queries with other where clauses on the same table, those would be candidates for secondary lookup vindexes.

Joins The next criteria to take into account are joins. If you are performing joins across tables, it will be beneficial to keep those rows together. In our example, we will be keeping the rows of the order table along with their customer. This grouping will allow us to efficiently perform operations like reading all orders placed by a customer.

Transactions It is important to keep transactions to be within one shard. Vitess currently does not guarantee atomicity for transactions that go across shards. Grouping related rows together usually results in transactions also falling within the same shard.

But there are situations where this may not be possible. If so, you can see if it will be possible to avoid this problem by grouping data differently. However, this may not be possible either. A well known use case is one where customers send each other money.

In such situations, you can look at refactoring the application such that transactions are broken into smaller single shard transactions.

Vitess will be adding support for distributed transactions soon.

Large Tenants If your application is tenant-based, it is possible that a single tenant may grow so big that they may not fit in one shard. If so, it is likely that the application is using a different key that has a higher cardinality than the tenant id.

The question to ask oneself is: if the tenant were a single application by themselves, what would be their sharding key, and then shard by that key instead of the tenant id.

Vitess has started rolling out support for multi-column Vindexes. Once this feature is fully done, you will be able to shard by the tenant id and a secondary key. The two-column sharding approach will allow you to group all data for a given tenant into a smaller set of shards rather than a random distribution. This may be beneficial for security or compliance reasons, in case the tenant would want their data to be physically isolated from other tenants.

Region Sharding The Vitess multi-column Vindex feature also allows you to locate data associated with regions in different geographical locations. For more details, see Region-based Sharding.

Many-to-Many relationships Sharding works well only if the relationship between data is hierarchical (one-to-one or one-to-many). If a table has foreign keys into multiple other tables, you have to choose one based on the strongest relationship and group the rows by that foreign key. The rest of the relationships will incur cross-shard overheads.

If more than one relationship is critically strong, you can look at the Materialization feature which allows you to create a materialized view of the table that is sharded using the other relationship. The application will write to the source, and the other view will be automatically updated.

Course Correction It may happen that the original sharding decision is not working out. It may also be possible that the application evolves in such a way that the sharding decision you previously made does not make sense any more.

In such cases, the MoveTables feature can be used to change the sharding key of existing tables.

Essentially, Vitess removes the fear of making the wrong sharding decision because you can always change your mind later.

Rule of Thumb Although a Vitess shard can grow to many terabytes, we have seen that 250GB is the sweet spot. If your data size approaches this limit, it is time to think about splitting your data.

Shared Vindexes and Foreign Keys

Let us now look at creating the `corder` table that will contain orders placed by the customers. It will be beneficial to group the rows of the orders in the same shard as that of the customer that placed the orders. Doing things this way will allow for simpler join queries between `customer` and `corder`. There will also be transactional benefits: any transaction that also updates the customer row along with an order will be a single shard transaction.

To make this happen in Vitess, all you have to do is specify that `corder.customer_id` uses the `hash` vindex, which is the same one used by `customer.customer_id`.

This is one situation where a Primary Vindex conceptually differs from a traditional database Primary Key. Whereas a Primary Key makes a row unique, a Vitess Primary Vindex only yields a Unique value. But multiple rows with the same Primary Vindex value can exist.

In other words, the Primary Vindex column need not be the primary key, or unique within MySQL. This is convenient for the `corder` table because we want customers to place multiple orders. In this case, all orders placed by a customer will have the same `customer_id`. The Primary Vindex for those will yield the same keyspace id as that of the customer. Therefore, all the rows for that customer's orders will end up in the same shard along with the customer row.

Since `corder` rows will need to have their own unique identifier, we also need to create a separate sequence for it in the product keyspace.

```
create table corder_seq(id bigint, next_id bigint, cache bigint, primary key(id)) comment
   'vitess_sequence';
insert into corder_seq(id, next_id, cache) values(0, 1, 3);
```

VSchema:

```
    "corder_seq": { "type": "sequence" }
```

We create the `corder` table as follows:

```
create table corder(corder_id bigint, customer_id bigint, product_id bigint, oname
   varchar(128), primary key(corder_id));
```

VSchema:

```
    "corder": {
      "column_vindexes": [{
          "column": "customer_id",
          "name": "hash"
        }],
      "auto_increment": {
        "column": "corder_id",
        "sequence": "product.corder_seq"
      }
    }
```

Alternate VSchema DDL:

```
alter vschema on customer.corder add vindex hash(customer_id);
alter vschema add sequence product.corder_seq;
alter vschema on customer.corder add auto_increment corder_id using product.corder_seq;
```

Inserting into `corder` yields the following results:

```
mysql> insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
Query OK, 5 rows affected (0.03 sec)

mysql> use `customer:-80`;
Database changed
mysql> select * from corder;
+-----------+-------------+------------+----------+
| corder_id | customer_id | product_id | oname    |
+-----------+-------------+------------+----------+
|         1 |           1 |          1 | gift     |
|         2 |           1 |          2 | gift     |
|         3 |           2 |          1 | work     |
|         4 |           3 |          2 | personal |
+-----------+-------------+------------+----------+
4 rows in set (0.00 sec)

mysql> use `customer:80-`;
Database changed
mysql> select * from corder;
+-----------+-------------+------------+----------+
| corder_id | customer_id | product_id | oname    |
+-----------+-------------+------------+----------+
|         5 |           4 |          1 | personal |
+-----------+-------------+------------+----------+
1 row in set (0.00 sec)
```

As expected, orders are created in the same shard as their customer. Selecting orders by their customer id goes to a single shard:

```
mysql> select * from corder where customer_id=1;
+-----------+-------------+------------+-------+
| corder_id | customer_id | product_id | oname |
+-----------+-------------+------------+-------+
|         1 |           1 |          1 | gift  |
|         2 |           1 |          2 | gift  |
+-----------+-------------+------------+-------+
2 rows in set (0.00 sec)
```

Joining `corder` with `customer` also goes to a single shard. This is also referred to as a local join:

```
mysql> select c.uname, o.oname, o.product_id from customer c join corder o on c.customer_id
   = o.customer_id where c.customer_id=1;
+-------+-------+------------+
| uname | oname | product_id |
+-------+-------+------------+
| alice | gift  |          1 |
| alice | gift  |          2 |
+-------+-------+------------+
2 rows in set (0.01 sec)
```

Performing the join without a `customer_id` constraint still results in a local join, but the query is scattered across all shards:

```
mysql> select c.uname, o.oname, o.product_id from customer c join corder o on c.customer_id
   = o.customer_id;
+---------+----------+------------+
| uname   | oname    | product_id |
+---------+----------+------------+
| alice   | gift     |          1 |
```

```
| alice   | gift     |          2 |
| bob     | work     |          1 |
| charlie | personal |          2 |
| dan     | personal |          1 |
+---------+----------+------------+
5 rows in set (0.00 sec)
```

However, adding a join with `product` results in a cross-shard join for the product part ot the query:

```
mysql> select c.uname, o.oname, p.pname from customer c join corder o on c.customer_id =
   o.customer_id join product p on o.product_id = p.product_id;
+---------+----------+----------+
| uname   | oname    | pname    |
+---------+----------+----------+
| alice   | gift     | monitor  |
| alice   | gift     | keyboard |
| bob     | work     | monitor  |
| charlie | personal | keyboard |
| dan     | personal | monitor  |
+---------+----------+----------+
5 rows in set (0.01 sec)
```

Although the underlying work performed by Vitess is not visible here, you can see it in the bottom right panel if using the demo app. Alternatively, you can also stream this information with the following command:

```
curl localhost:12345/debug/querylog
[verbose output not shown]
```

Foreign Keys More generically stated: If a table has a foreign key into another table, then Vitess can ensure that the related rows live in the same shard by making them share a common Unique Vindex.

In cases where you choose to group rows based on their foreign key relationships, you have the option to enforce those constraints within each shard at the MySQL level. You can also configure cascade deletes as needed. However, overuse of foreign key constraints is generally discouraged in MySQL.

Foreign key constraints across shards or keyspaces are not supported in Vitess. For example, you cannot specify a foreign key between `corder.product_id` and `product.product_id`.

Many-to-Many relationships In the case where a table has relationships with multiple other tables, you can only choose one of those relationships for shard grouping. All other relationships will end up being cross-shard, and will incur cross-shard penalties.

If a table has strong relationships with multiple other tables, and if performance becomes a challenge choosing either way, you can explore the VReplication Materialization feature that allows you to materialize a table both ways.

Enforcing Uniqueness To enforce global uniqueness for a row in a sharded table, you have to have:

- A Unique Vindex on the column
- A Unique constraint at the MySQL level

A Primary Vindex coupled with a Primary Key constraint makes a row globally unique.

A Unique Vindex can also be specified for a non-unique column. In such cases, it is likely that you will be using that column in a where clause, and will require a secondary non-unique index on it at the MySQL level.

Unique Lookup Vindexes

Certain application features may require you to point-select orders by their id with a query like this:

```
select * from corder where corder_id=1;
```

However, issuing this query to Vitess will cause it to scatter this query across all shards because there is no way to know which shard contains that order id. This would be inefficient if the QPS of this query or the number of shards is too high.

Vitess supports the concept of lookup vindexes, also known as cross-shard indexes. You can instruct Vitess to create and manage a lookup vindex for the `corder_id` column. Such a vindex needs to maintain a mapping from `corder_id` to the `keyspace_id` of the row, which will be stored in a lookup table.

This lookup table can be created in any keyspace, and it may or may not be sharded. In this particular case, we are going to create the table in the unsharded product keyspace even though the lookup vindex itself is going to be in the `customer` keyspace:

```
create table corder_keyspace_idx(corder_id bigint not null auto_increment, keyspace_id
   varbinary(10), primary key(corder_id));
```

The primary key is `corder_id`. The unique constraint on `corder_id` makes the Lookup Vindex unique: for a given `corder_id` as input, at most one `keyspace_id` can be produced. It is not necessary to name the column as `corder_id`, but it is less confusing to do so.

Since the table is not sharded, we have a trivial VSchema addition:

```
    "corder_keyspace_idx": {}
```

We can now instantiate the Lookup Vindex in the VSchema of the `customer` keyspace:

```
    "corder_keyspace_idx": {
      "type": "consistent_lookup_unique",
      "params": {
        "table": "product.corder_keyspace_idx",
        "from": "corder_id",
        "to": "keyspace_id"
      },
      "owner": "corder"
    }
```

- The vindex is given a distinctive name `corder_keyspace_idx` because of its specific input parameters.
- The vindex type is `consistent_lookup_unique`. We expect this lookup vindex to yield at most one keyspace id for a given input. The `consistent` qualifier is explained below.
- The `params` section of a Vindex is a set of key-value strings. Each vindex expects a different set of parameters depending on the implementation. A lookup vindex requires the following three parameters:
  - `table` should be the name of the lookup table. It is recommended that it is fully qualified.
  - The `from` and `to` fields must reference the column names of the lookup table.
  - An optional fourth parameter will be available in Vitess 12.0:
    - `batch_lookup`: Set this to `"true"` if you want lookups for key values in the lookup vindex table to be batched (i.e. all values that match to a shard of the lookup backing table will be done in a single query). This can be important if you have queries with multiple point lookup values in the `WHERE` clause (e.g. using `IN`). This is not enabled by default, since it only supports binary equality, i.e. does not take collation into account. You should therefore only use it with lookup vindexes where the `from` key is binary or binary equivalent (e.g. integer or a character field where you are sure that collation matching is not required). Without this option, lookups of multiple values in the shards backing the lookup table will be performed one-by-one, and can have a significant latency overhead if you have a large number of values to lookup per query, relative to your number of shards.
- The `owner` field indicates that `corder` is responsible for populating the lookup table and keeping it up-to-date. This means that an insert into `corder` will result in a corresponding lookup row being inserted in the lookup table, etc. Lookup vindexes can also be shared, but they can have only one owner each. We will later see an example about how to share lookup vindexes.

{{< info >}} Since `corder_keyspace_idx` and `corder` are in different keyspaces, any change that affects the lookup column results in a distributed transaction between the `customer` shard and the `product` keyspace. Usually, a two-phase commit (2PC) protocol would need to be used for the distributed transaction. However, the `consistent` lookup vindex utilizes a special algorithm that orders the commits in such a way that a commit failure resulting in partial commits does not result in inconsistent data. This avoids the extra overheads associated with 2PC. {{< /info >}}

Finally, we must associate `customer.corder_id` with the lookup vindex:

```
      "column_vindexes": [{
          "column": "customer_id",
          "name": "hash"
        }, {
          "column": "corder_id",
          "name": "corder_keyspace_idx"
        }]
```

Note that `corder_id` comes after `customer_id` implying that `customer_id` is the Primary Vindex for this table.

Alternate VSchema DDL:

```
alter vschema add table product.corder_keyspace_idx;
alter vschema on customer.corder add vindex corder_keyspace_idx(corder_id) using
   consistent_lookup_unique with owner=`corder`, table=`product.corder_keyspace_idx`,
   from=`corder_id`, to=`keyspace_id`;
```

{{< info >}} An owned lookup vindex (even if unique) cannot be a Primary Vindex because it creates an association against a keyspace id after one has been assigned to the row. The job of computing the keyspace id must therefore be performed by a different unique vindex. {{< /info >}}

Bringing up the demo application again, you can now see the lookup table being automatically populated when rows are inserted in `corder`:

```
mysql> insert into corder(customer_id, product_id, oname) values
   (1,1,'gift'),(1,2,'gift'),(2,1,'work'),(3,2,'personal'),(4,1,'personal');
Query OK, 5 rows affected (0.00 sec)

mysql> select corder_id, hex(keyspace_id) from corder_keyspace_idx;
+-----------+------------------+
| corder_id | hex(keyspace_id) |
+-----------+------------------+
|         1 | 166B40B44ABA4BD6 |
|         2 | 166B40B44ABA4BD6 |
|         3 | 06E7EA22CE92708F |
|         4 | 4EB190C9A2FA169C |
|         5 | D2FD8867D50D2DFE |
+-----------+------------------+
5 rows in set (0.01 sec)
```

And then, issuing a query like `select * from corder where corder_id=1` results in two single-shard round-trips instead of a full scatter.

Reversible Vindexes Looking at the rows in `corder_keyspace_idx` reveals a few things. We get to now see actual keyspace id values that were previously invisible. We can also notice that two different inputs `1` and `2` map to the same keyspace id `166B40B44ABA4BD6`. In other words, a unique vindex does not necessarily guarantee that two different values yield different keyspace ids. In fact, this is derived from the fact that there are two order rows for customer id `1`.

Vindexes that do have a one-to-one correspondence between the input value and keyspace id , like `hash`, are known as reversible vindexes: Given a keyspace id, the input value can be back-computed. This property will be used in a later example.

Backfill Creating a lookup vindex after the main table already contains rows does not automatically backfill the lookup table for the existing entries. Only newer inserts cause automatic population of the lookup table. This backfill can be set up using the `CreateLookupVindex` command covered below.

Checklist Creating a unique lookup Vindex is an elaborate process. It is good to use the following checklist if this is done manually:

- Create the lookup table as sharded or unsharded. Make the `from` column the primary key.
- Create a VSchema entry for the lookup table. If sharded, assign a Primary Vindex for the `from` column.
- Create the lookup vindex in the VSchema of the sharded keyspace:
  - Give it a distinct name
  - Specify the type as `consistent_lookup_unique`
  - Under `params`: specify the properties of the lookup table
  - Specify the `Owner` as the main table
- Associate the column of the owner table with the new Vindex.

CreateLookupVindex command vtctld supports the CreateLookupVindex command that can perform all the above steps as well as the backfill.

{{< warning >}} This will not work against the `vtcombo` based demo app because it does not support vreplication. You can only try this against a real Vitess cluster. {{< /warning >}}

Save the following json into a file, say `corder_keyspace_idx.json`:

```
{
  "sharded": true,
  "vindexes": {
    "corder_keyspace_idx": {
      "type": "consistent_lookup_unique",
      "params": {
        "table": "product.corder_keyspace_idx",
        "from": "corder_id",
        "to": "keyspace_id"
      },
      "owner": "corder"
    }
  },
  "tables": {
    "corder": {
      "column_vindexes": [{
          "column": "corder_id",
          "name": "corder_keyspace_idx"
      }],
    }
  }
}
```

And issue the vtctlclient command:

```
$ vtctlclient -server <vtctld_grpc_address> CreateLookupVindex -tablet_types=REPLICA
   customer "$(cat corder_keyspace_idx.json)"
```

The workflow automatically infers the schema and vschema for the lookup table and creates it. It also sets up the necessary VReplication streams to backfill the lookup table.

After the backfill is done, you should clean up the workflow. More detailed instructions are available in the CreateLookupVindex Reference

Unsharded Keyspace

We are going to start with configuring the `product` table in the unsharded keyspace `product`. The schema file should be as follows:

```
create table product(product_id bigint auto_increment, pname varchar(128), primary
   key(product_id));
```

`product_id` is the primary key for product, and it is also configured to use MySQL's `auto_increment` feature that allows you to automatically generate unique values for it.

We also need to create a VSchema for the `product` keyspace and specify that `product` is a table in the keyspace:

```
{
  "sharded": false,
  "tables": {
    "product": {}
  }
}
```

The json states that the keyspace is not sharded. The product table is specified in the "tables" section of the json. This is because there are other sections that we will introduce later.

For unsharded keyspaces, no additional metadata is needed for regular tables. So, their entry is empty.

Alternate VSchema DDL:

```
alter vschema add table product.product;
```

{{< info >}} If `product` is the only keyspace in the cluster, a vschema is unnecessary. Vitess treats single keyspace clusters as a special case and optimistically forwards all queries to that keyspace even if there is no table metadata present in the vschema. But it is a best practice to provide a full vschema to avoid future complications. {{< /info >}}

Bringing up the cluster will allow you to access the `product` table. You can now insert rows into the table:

```
$ mysql -h 127.0.0.1 -P 12348
[snip]
mysql> insert into product(pname) values ('monitor'), ('keyboard');
Query OK, 2 rows affected (0.00 sec)

mysql> select * from product;
+------------+----------+
| product_id | pname    |
+------------+----------+
|          1 | monitor  |
|          2 | keyboard |
+------------+----------+
2 rows in set (0.00 sec)
```

The insert does not specify values for `product_id`, because we are relying on MySQL's `auto_increment` feature to populate it.

You will notice that we did not connect to the `product` database or issue a `use` statement to select it. This is the 'unspecified' mode supported by Vitess. As long as a table name can be uniquely identified from the vschemas, Vitess will automatically direct the query to the correct keyspace.

You can also connect or specify keyspaces as if they were MySQL databases. The following constructs are valid:

```
mysql> select * from product.product;
+------------+----------+
| product_id | pname    |
+------------+----------+
|          1 | monitor  |
```

```
|          2 | keyboard |
+------------+----------+
2 rows in set (0.00 sec)

mysql> use product;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from product;
+------------+----------+
| product_id | pname    |
+------------+----------+
|          1 | monitor  |
|          2 | keyboard |
+------------+----------+
2 rows in set (0.01 sec)
```