The Vitess Documentation

Contents

Vitess is a database solution for deploying, scaling and managing large clusters of MySQL instances. It's built to run with equal effectiveness on public cloud architecture, private cloud architecture, and dedicated hardware.

Vitess and MySQL

Vitess combines and extends the important features of MySQL with the scalability of a NoSQL database. Vitess can help you with a variety of problems, including:

- Scaling a MySQL database, using sharding, while keeping application changes to a minimum
- Migrating your MySQL installation from bare metal to a private or public cloud
- Deploying and managing a large number of MySQL instances

Vitess database drivers

Vitess includes compliant JDBC and Go (Golang) database drivers using a native query protocol. Additionally, it implements the MySQL server protocol, which is compatible with virtually any other language.

Vitess in action

Vitess has been serving all YouTube database traffic since 2011, and has now been adopted by many enterprises for their production needs.

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.

Keyspace Graph

The *keyspace graph* allows Vitess to decide which set of shards to use for a given keyspace, cell, and tablet type.

Partitions

During horizontal resharding (splitting or merging shards), there can be shards with overlapping key ranges. For example, the source shard of a split may serve `c0-d0` while its destination shards serve `c0-c8` and `c8-d0` respectively.

Since these shards need to exist simultaneously during the migration, the keyspace graph maintains a list (called a *partitioning* or just a *partition*) of shards whose ranges cover all possible keyspace ID values, while being non-overlapping and contiguous. Shards can be moved in and out of this list to determine whether they are active.

The keyspace graph stores a separate partitioning for each `(cell, tablet type)` pair. This allows migrations to proceed in phases: first migrate *rdonly* and *replica* requests, one cell at a time, and finally migrate *master* requests.

Served From

During vertical resharding (moving tables out from one keyspace to form a new keyspace), there can be multiple keyspaces that contain the same table.

Since these multiple copies of the table need to exist simultaneously during the migration, the keyspace graph supports keyspace redirects, called `ServedFrom` records. That enables a migration flow like this:

1. Create `new_keyspace` and set its `ServedFrom` to point to `old_keyspace`.
2. Update the app to look for the tables to be moved in `new_keyspace`. Vitess will automatically redirect these requests to `old_keyspace`.
3. Perform a vertical split clone to copy data to the new keyspace and start filtered replication.
4. Remove the `ServedFrom` redirect to begin actually serving from `new_keyspace`.
5. Drop the now unused copies of the tables from `old_keyspace`.

There can be a different `ServedFrom` record for each `(cell, tablet type)` pair. This allows migrations to proceed in phases: first migrate *rdonly* and *replica* requests, one cell at a time, and finally migrate *master* requests.

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 without downtime.

Identifying Candidate Tables

It is recommended to keep tables that need to join on eachother 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 also be able to migrate to 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.

Related Vitess Documentation

- MoveTables User Guide

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 division within a keyspace. A shard typically contains one MySQL master and many MySQL slaves.

Each MySQL instance within a shard has the same data (excepting some replication lag). The slaves can serve read-only traffic (with eventual consistency guarantees), execute long-running data analysis tools, or perform administrative tasks (backup, restore, diff, etc.).

An unsharded keyspace has effectively one shard. Vitess names the shard `0` by convention. When sharded, a keyspace has `N` shards with non-overlapping data.

Shard Naming

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 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`

`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.

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 slave 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 slave 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. Since *cluster* is an overloaded term, and one Vitess cluster is distinguished from another by the fact that each has its own global topology service, we refer to each Vitess cluster as a **toposphere**.

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:

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 allows a human or application to easily interact with a Vitess implementation. Using vtctl, you can identify master and replica databases, create tables, initiate failovers, perform sharding (and resharding) operations, and so forth.

As vtctl performs operations, it updates the lockserver 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 lockserver. It is useful for troubleshooting or for getting a high-level overview of the servers and their current states.

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.

vtworker

vtworker hosts long-running processes. It supports a plugin architecture and offers libraries so that you can easily choose tablets to use. Plugins are available for the following types of jobs:

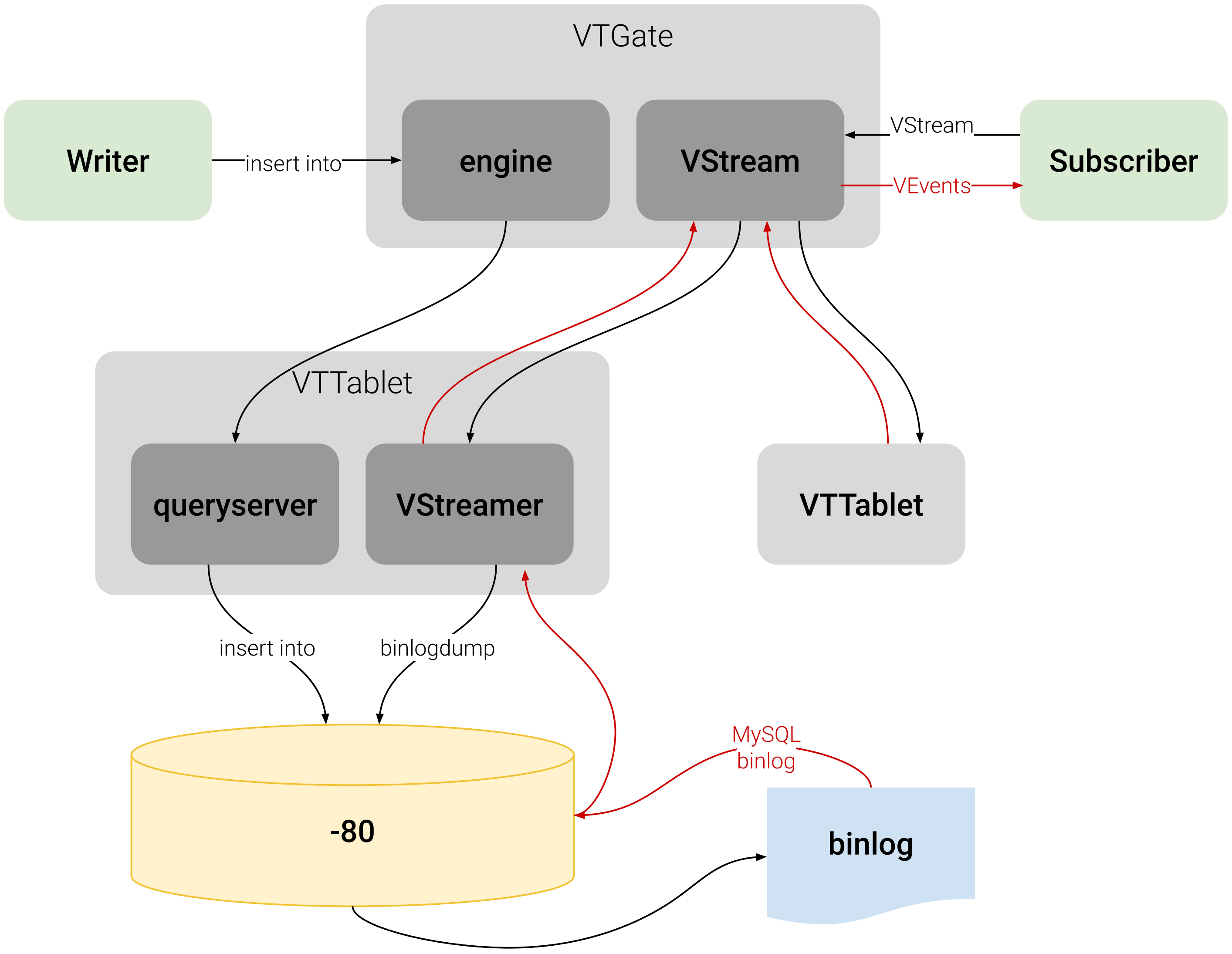

Figure 1: VStream diagram

- resharding differ jobs check data integrity during shard splits and joins
- vertical split differ jobs check data integrity during vertical splits and joins

vtworker also lets you easily add other validation procedures. You could do in-tablet integrity checks to verify foreign-key-like relationships or cross-shard integrity checks if, for example, an index table in one keyspace references data in another keyspace.

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 1.13+ Download and install Golang 1.13. For example, at writing:

```
curl -O https://dl.google.com/go/go1.13.9.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.13.9.linux-amd64.tar.gz
```

Make sure to add go to your bashrc:

```
export PATH=$PATH:/usr/local/go/bin
```

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

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

# 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@1.13 automake git curl wget mysql@5.7 etcd
```

Add `mysql@5.7` and `go@1.13` to your `PATH`:

```
echo 'export PATH="/usr/local/opt/mysql@5.7/bin:$PATH"' >> ~/.bash_profile
echo 'export PATH="/usr/local/opt/go@1.13/bin:$PATH"' >> ~/.bash_profile
```

Do not setup 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 cask install adoptopenjdk8
brew cask 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]
```

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 1.13+ Download and install Golang 1.13. For example, at writing:

```
curl -O https://dl.google.com/go/go1.13.9.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.13.9.linux-amd64.tar.gz
```

Make sure to add go to your bashrc:

```
export PATH=$PATH:/usr/local/go/bin
```

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

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

# 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
```

Code Reviews

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

What to look for in a Review

Both authors and reviewers need to answer these general questions:

- Does this change match an existing design / bug?
- Is there proper unit test coverage for this change? All changes should increase coverage. We need at least integration test coverage when unit test coverage is not possible.
- 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 change match our coding conventions / style? Linter was run and is happy?
- Does this match our current patterns? Example include RPC patterns, Retries / Waits / Timeouts patterns using Context, …

Additionally, we 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:

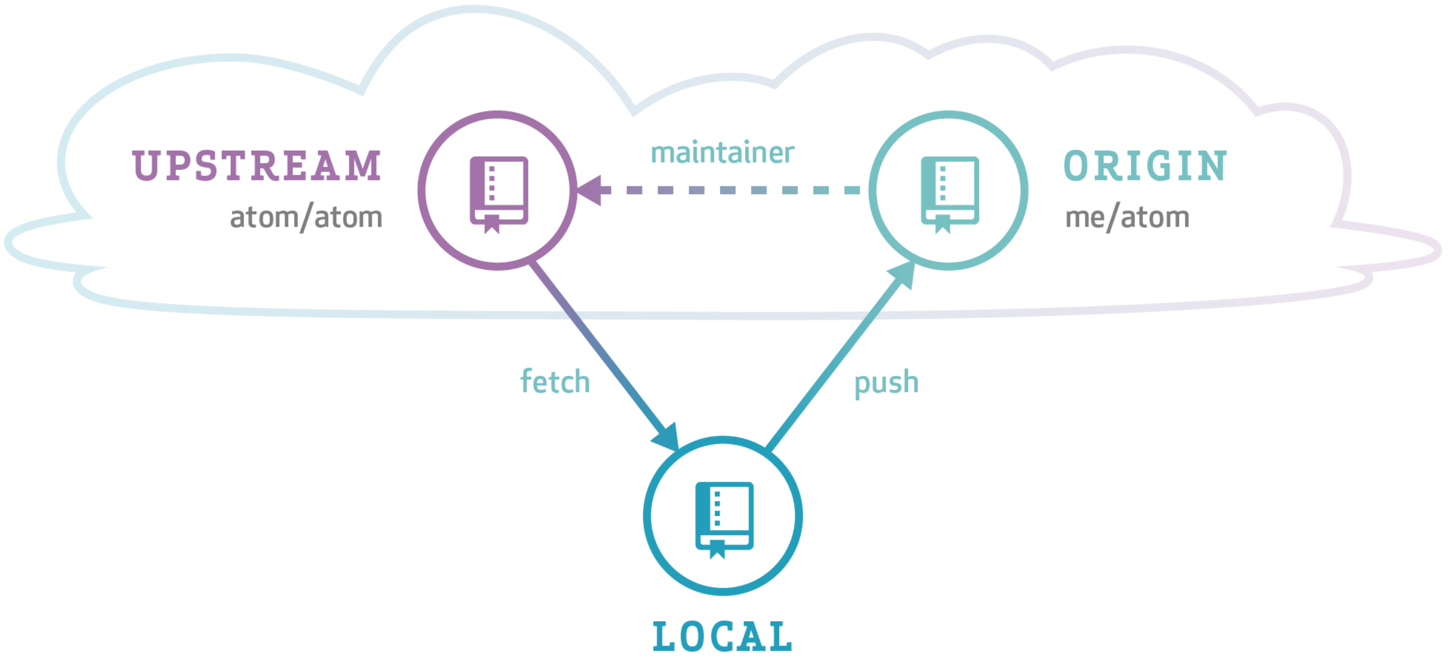

Image Source

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 `master` branch, do this:

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

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

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

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

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

```
(master) $ git pull
```

Topic Branches

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

```
$ git checkout master
(master) $ git pull
(master) $ 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 master 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`)

FAQ

description: Frequently Asked Questions about Vitess

Configuration

description: Frequently Asked Questions about Configuration

Does the application need to know about the sharding scheme underneath Vitess?

The application does not need to know about how the data is sharded. This information is stored in a VSchema which the VTGate servers use to automatically route your queries. This allows the application to connect to Vitess and use it as if it's a single giant database server.

Can I override the default db name from vt_xxx to my own? Yes. You can start vttablet with the `-init_db_name_override` command line option to specify a different db name. There is no downside to performing this override

How do I connect to vtgate using MySQL protocol? If you look at the example vtgate-up.sh script, you'll see the following lines:

```
-mysql_server_port $mysql_server_port \
-mysql_server_socket_path $mysql_server_socket_path \
-mysql_auth_server_static_file "./mysql_auth_server_static_creds.json" \
```

In this example, vtgate accepts MySQL connections on port 15306 and the authentication info is stored in the json file. So, you should be able to connect to it using the following command:

```
mysql -h 127.0.0.1 -P 15306 -u mysql_user --password=mysql_password
```

I cannot start a cluster, and see these errors in the logs: Could not open required defaults file: /path/to/my.cnf

Most likely this means that AppArmor is running on your server and is preventing Vitess processes from accessing the my.cnf file. The workaround is to uninstall AppArmor:

```
sudo service apparmor stop
sudo service apparmor teardown
sudo update-rc.d -f apparmor remove
```

You may also need to reboot the machine after this. Many programs automatically install AppArmor, so you may need to uninstall again.

Queries

description: Frequently Asked Questions about Queries

Can I address a specific shard if I want to?

If necessary, you can access a specific shard by connecting to it using the shard specific database name. For a keyspace ks and shard -80, you would connect to ks:-80.

How do I choose between master vs. replica for queries?

You can qualify the keyspace name with the desired tablet type using the @ suffix. This can be specified as part of the connection as the database name, or can be changed on the fly through the USE command.

For example, `ks@master` will select `ks` as the default keyspace with all queries being sent to the master. Consequently `ks@replica` will load balance requests across all `REPLICA` tablet types, and `ks@rdonly` will choose `RDONLY`.

You can also specify the database name as `@master`, etc, which instructs Vitess that no default keyspace was specified, but that the requests are for the specified tablet type.

If no tablet type was specified, then VTGate chooses its default, which can be overridden with the `-default_tablet_type` command line argument.

There seems to be a 10 000 row limit per query. What if I want to do a full table scan?

Vitess supports different modes. In OLTP mode, the result size is typically limited to a preset number (10 000 rows by default). This limit can be adjusted based on your needs.

However, OLAP mode has no limit to the number of rows returned. In order to change to this mode, you may issue the following command before executing your query:

```
set workload='olap'
```

You can also set the workload to `dba mode`, which allows you to override the implicit timeouts that exist in vttablet. However, this mode should be used judiciously as it supersedes shutdown and reparent commands.

The general convention is to send OLTP queries to `REPLICA` tablet types, and OLAP queries to `RDONLY`.

Is there a list of supported/unsupported queries?

Please see "SQL Syntax" under MySQL Compatibility.

If I have a log of all queries from my app. Is there a way I can try them against Vitess to see how they'll work?

Yes. The vtexplain tool can be used to preview how your queries will be executed by Vitess. It can also be used to try different sharding scenarios before deciding on one.

VIndexes

description: Frequently Asked Questions about VIndexes

Does the Primary Vindex for a tablet have to be the same as its Primary Key?

It is not necessary that a Primary Vindex be the same as the Primary Key. In fact, there are many use cases where you would not want this. For example, if there are tables with one-to-many relationships, the Primary Vindex of the main table is likely to be the same as the Primary Key. However, if you want the rows of the secondary table to reside in the same shard as the parent row, the Primary Vindex for that table must be the foreign key that points to the main table. A typical example is a user and order table. In this case, the order table has the `user_id` as a foreign key to the `id` of the user table. The `order_id` may be the primary key for `order`, but you may still want to choose `user_id` as Primary Vindex, which will make a user's orders live in the same shard as the user.

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.

Run Vitess on Kubernetes

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

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 --cpus=4 --memory=8000
   ```

   Note the additional resource requirements. In order to go through all the use cases, many vttablet and MySQL instances will be launched. These require more resources than the defaults used by Minikube.

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-v3.1.2-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:

```
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-2pbzn          0/1     Completed   0          2m44s
pod/commerce-apply-vschema-initial-mfhvl         0/1     Completed   0          2m44s
pod/vtctld-6f955957bb-67bq7                      1/1     Running     0          2m44s
pod/vtgate-zone1-86b7cb87d6-vckzw                1/1     Running     3          2m44s
pod/zone1-commerce-0-init-shard-master-dh727     0/1     Completed   0          2m44s
pod/zone1-commerce-0-replica-0                   5/6     Running     0          2m44s
pod/zone1-commerce-0-replica-1                   5/6     Running     0          2m44s
pod/zone1-commerce-0-replica-2                   5/6     Running     0          2m44s

NAME                                             COMPLETIONS   DURATION   AGE
job.batch/commerce-apply-schema-initial          1/1           118s       2m44s
job.batch/commerce-apply-vschema-initial         1/1           109s       2m44s
job.batch/zone1-commerce-0-init-shard-master     1/1           115s       2m44s
```

Setup Aliases

For ease-of-use, Vitess provides aliases for `mysql` and `vtctlclient`. This script also sets up all the required networking:

```
source alias.source
```

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)
```

You can also browse to the vtctld console using the following command (Ubuntu):

```
./kvtctld.sh
```

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
```

You will need to delete the persistent volume claims too

```
kubectl delete pvc $(kubectl get pvc | grep vtdataroot-zone1 | awk '{print $1}')
```

Congratulations on completing this exercise!— ## Run Vitess Locally 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.

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:

```
tar -xzf vitess-6.0.20-20200429-f7fa695.tar.gz
cd vitess-6.0.20-20200429-f7fa695
sudo mkdir -p /usr/local/vitess
sudo mv * /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
```

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:

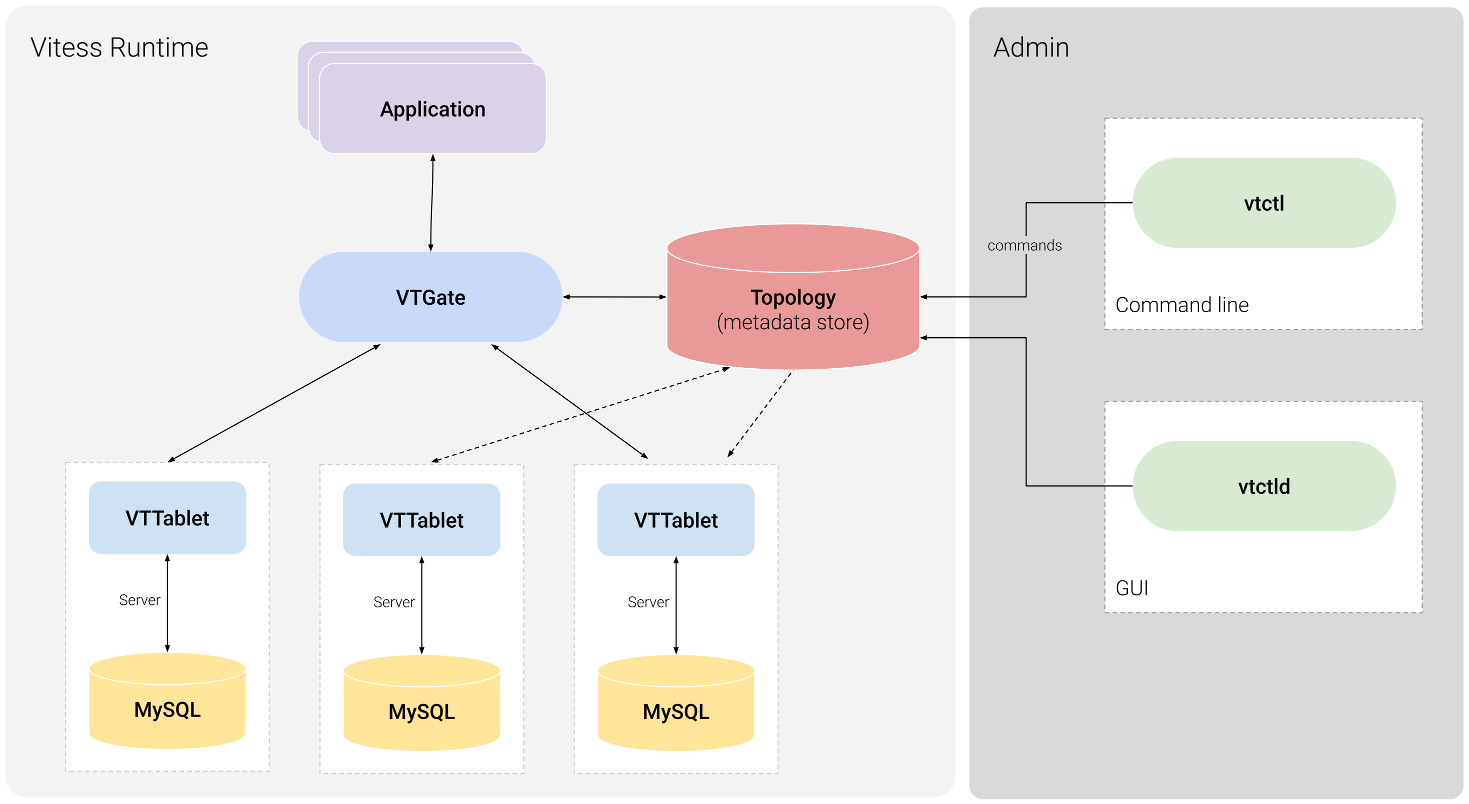

Figure 2: Architecture Diagram

For additonal 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.

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 has been serving all YouTube database traffic since 2011, and has now been adopted by many enterprises 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 deadlines 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.
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.

Reference

description: Detailed information about specific Vitess functionality

Vitess 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.

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 long as there are not expressions that compare columns on the outer table to the inner table in sharded keyspaces.

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 Vitess does not yet support MySQL Stored Procedures.

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'
```

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.

Session Scope

Vitess uses a connection pool to fan-in connections from VTGate servers to Tablet servers. VTGate servers will refuse statements that make changes to the connection's session scope. This includes:

- `SET SESSION var=x`
- `CREATE TEMPORARY TABLE`
- `SET @var=x`

The exception to this, is that Vitess maintains a whitelist statements that MySQL connectors may use when they first connect to Vitess. Vitess will ignore these noisy statements when it knows it is safe to do so.

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.
- `SHOW vitess_tablets` – Information about the current Vitess tablets such as the keyspace, key ranges, tablet type, hostname, and status.
- `SHOW vitess_shards` – A list of shards that are available.
- `SHOW vschema tables` – A list of tables available in the current keyspace's vschema.
- `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.

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.

MySQL 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.

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 slave 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 slave 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 slaves via replication. 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_file=<sql 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.

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 slaves. The master handles write operations, while slaves 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.
- The MapReduce framework fully utilizes key ranges to read data as quickly as possible, concurrently from all shards and all replicas.

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 also uses RPCs. These RPCs may use secure transport options. This document explains how to use these features.

Overview

The following diagram represents all the RPCs we use in a Vitess cluster:

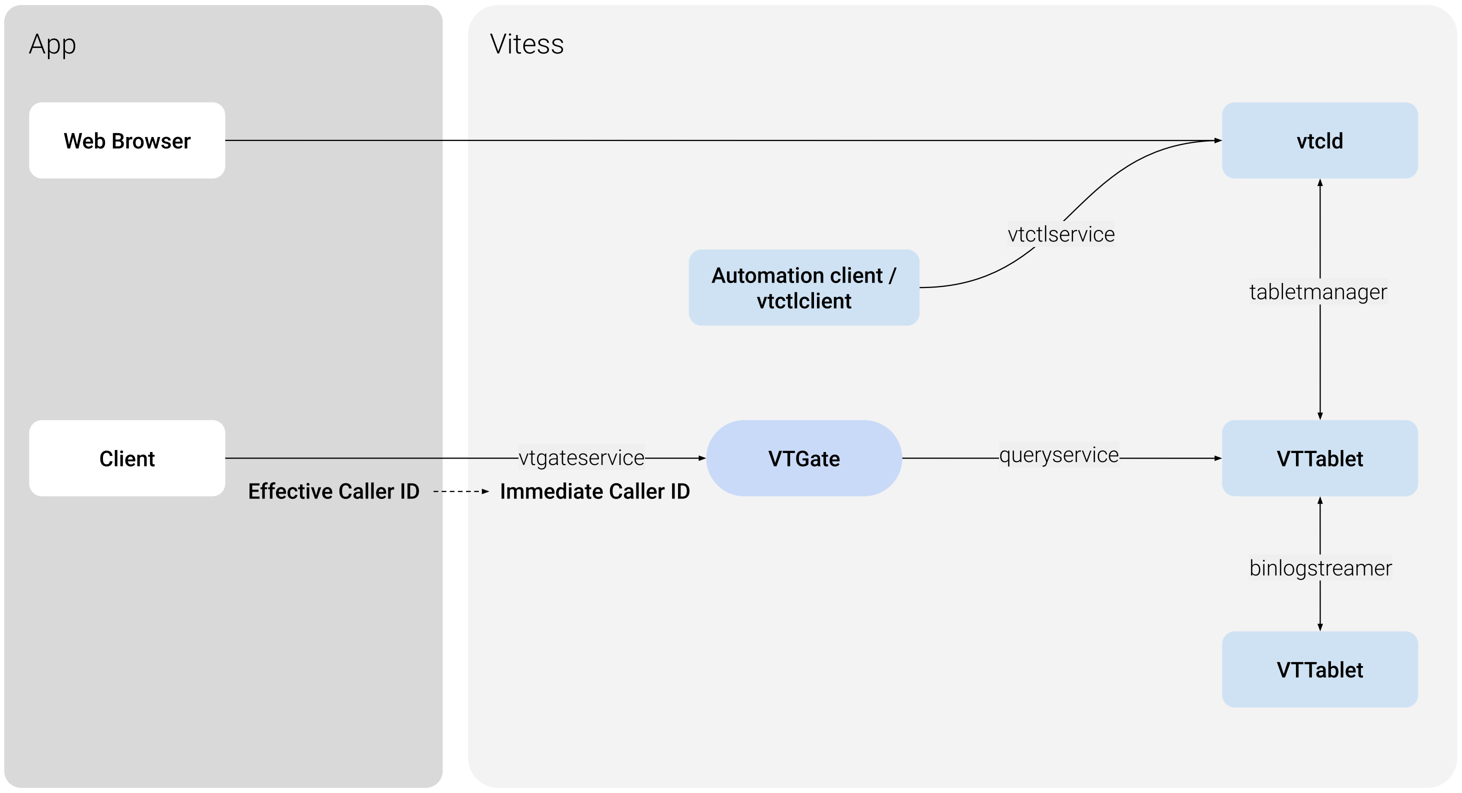

Figure 3: Vitess Transport Security Model Diagram

There are two main categories:

- Internal RPCs: they are used to connect Vitess components.
- Externally visible RPCs: they are use by the app to talk to Vitess.

A few features in the Vitess ecosystem depend on authentication, like Caller ID and table ACLs. We'll explore the Caller ID feature first.

The encryption and authentication scheme used depends on the transport used. With gRPC (the default for Vitess), TLS can be used to secure both internal and external RPCs. We'll detail what the options are.

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, represents the user connecting to Vitess (vtgate).

    - It is authenticated by the transport layer used.
    - It is used by the Vitess TableACL feature.

- Effective Caller ID: It provides detailed information on who the individual caller process is:
    - It contains more information about the caller: principal, component, sub-component.
    - It is provided by the application layer.
    - It is not authenticated.
    - It is exposed in query logs to be able to debug the source of a slow query, for instance.

gRPC Transport

gRPC Encrypted Transport When using gRPC transport, Vitess can use the usual TLS security features (familiarity with SSL / 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 use a certificate signed by one ca in the provided file.
- A Vitess go client can be configured with symmetrical parameters to enable TLS:
    - `..._grpc_ca`: list of server cert signers to trust.
    - `..._grpc_server_name`: name of the server cert to trust, instead of the hostname used to connect.
    - `..._grpc_cert`, `..._grpc_key`: client side cert and key to use (when the server requires client authentication)
- Other clients can take similar parameters, in various ways, see each client for more information.

With these options, it is possible to use TLS-secured connections for all parts of the system. This enables the server side to authenticate the client, and / or the client to authenticate the server.

Note this is not enabled by default, as usually the different Vitess servers will run on a private network (in a Cloud environment, usually all local traffic is already secured over a VPN, for instance).

Certificates and Caller ID Additionally, if a client uses a certificate to connect to Vitess (vtgate), 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 SSL 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 test/encrypted_transport.py in the source tree. It first sets up all the certificates, and some table ACLs, then uses the python client to connect with SSL. It also exercises the `grpc_use_effective_callerid` flag, by connecting without SSL.

MySQL Transport

To get `vtgate` to support SSL/TLS use `-mysql_server_ssl_cert` and `-mysql_server_ssl_key`.

To require client certificates set `-mysql_server_ssl_ca`. If there is no CA specified then TLS is optional.

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`. This mapping can be used to identify the location of 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 there is only one access path to the data, which is the Key. However, relational databases are more versatile about the data and their relationships. So, sharding a database by only designating a 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 you to define 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

Vindexes offer many flexibilities:

- A table can have multiple Vindexes.
- Vindexes could be NonUnique, which allows a column value to yield multiple keyspace IDs.
- They could be a simple function or be based on a lookup table.
- They could be shared across multiple tables.
- Custom Vindexes can be plugged in, and Vitess will still know how to reshard using such Vindexes.

The Primary Vindex The Primary Vindex 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. This unique mapping will be used at the time of insert to decide the target shard for a row. Conceptually, this is also equivalent to the NoSQL Sharding Key, and we often refer to the Primary Vindex as the Sharding Key.

Uniqueness for a Primary Vindex does not mean that the column has to be a primary key or unique in the MySQL schema. You can have multiple rows that map to the same keyspace ID. The Vindex uniqueness constraint is only used to make sure that all rows for a keyspace ID live in the same shard.

However, there is a subtle difference: NoSQL datastores let you choose the Sharding Key, but the Sharding Scheme is generally hardcoded in the engine. In Vitess, the choice of Vindex lets you control how a column value maps to a keyspace ID. In other words, a Primary Vindex in Vitess not only defines the Sharding Key, it also decides the Sharding Scheme.

Vindexes come in many varieties. Some of them can be used as Primary Vindex, and others have different purposes. The following sections will describe their properties.

Secondary Vindexes Secondary Vindexes are additional vindexes you can define against other columns of a table offering you 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.

Secondary Vindexes are also commonly known as cross-shard indexes. It is important to note that Secondary Vindexes are only for making routing decisions. The underlying database shards will most likely need traditional indexes on those same columns.

Unique and NonUnique Vindex A Unique Vindex is one that yields at most one keyspace ID for a given input. Knowing that a Vindex is Unique is useful because VTGate can push down some complex queries into VTTablet if it knows that the scope of that query cannot exceed a 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 let you efficiently search for users that match a certain `name`.

Functional and Lookup Vindex A Functional Vindex is one where the column value to keyspace ID mapping is pre-established, typically through an algorithmic function. In contrast, a Lookup Vindex is one that gives you the ability to create an association between a value and a keyspace ID, and recall it later when needed.

Typically, the Primary Vindex is Functional. In some cases, it is the identity function where the input value yields itself as the keyspace id. However, one could also choose other algorithms like hashing or mod functions.

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. 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 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.

Shared Vindexes 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 are currently two vindex types for consistent lookup:

Consistent lookup vindexes use an alternate approach that makes use of careful locking and transaction sequences to guarantee consistency without using 2PC. This gives you the best of both worlds, where you get a consistent cross-shard vindex without paying the price of 2PC.

Relational databases encourage normalization, which lets you split 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, aka `Foreign Key`.

- `consistent_lookup_unique`
- `consistent_lookup`

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, you would find that the backing tables would actually be identical. In such cases, Vitess lets you share a single Lookup Vindex for multiple tables. Of these, one of them is designated as the owner, which 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`, you can change 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, it's recommended that the lookup table 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 `(user_id, email)`, where `user_id` was the primary key and `email` needed a non-unique lookup vindex, the lookup table would have the following columns `(email, user_id, keyspace_id)`.

Orthogonality The previously described properties are mostly orthogonal. Combining them gives rise to the following valid categories:

- **Functional Unique**: This is 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**: This gets used for optimizing high QPS queries that do not use the Primary Vindex columns in their WHERE clause. There is a price to pay: You incur an extra write to the lookup table for insert and delete operations, and an extra lookup for read operations. However, it is worth it if you do not want these high QPS queries to be sent to all shards.
- **Lookup Unique Unowned**: This category is used as an optimization as described in the Shared Vindexes section.
- **Lookup NonUnique Owned**: This gets 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 Primary. 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 Primary because it is too slow for resharding. If absolutely unavoidable, you can use a Lookup Vindex as Primary. In such cases, it is recommended that you add a `keyspace ID` column to such tables. While resharding, Vitess can use that column to efficiently compute the target shard. You can even configure Vitess to auto-populate that column on inserts. This is done using the reverse map feature explained below.

How vindexes are used

Cost Vindexes have costs. For routing a query, the Vindex with the lowest cost is chosen. The current costs are:

Vindex Type	Cost
Indentity	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, you can refer to the design doc 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 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.

Predefined Vindexes Vitess provides the following predefined Vindexes:

Name	Type	Description	Primary	Reversible	Cost
binary	Functional Unique	Identity	Yes	Yes	0
binary_md5	Functional Unique	md5 hash	Yes	No	1
hash	Functional Unique	3DES null-key hash	Yes	Yes	1
lookup	Lookup NonUnique	Lookup table non-unique values	No	Yes	20
lookup_unique	Lookup Unique	Lookup table unique values	If unowned	Yes	10
consistent_lookup	Lookup NonUnique	Lookup table non-unique values	No	No	20
consistent_lookup_unique	Lookup Unique	Lookup table unique values	unowned	No	10
numeric	Functional Unique	Identity	Yes	Yes	0
numeric_static_map	Functional Unique	A JSON file that maps input values to keyspace IDs	Yes	No	1
unicode_loose_md5	Functional Unique	Case-insensitive (UCA level 1) md5 hash	Yes	No	1
reverse_bits	Functional Unique	Bit Reversal	Yes	Yes	1

Consistent lookup vindexes are a new category of vindexes that are meant to replace the existing lookup vindexes. For the time being, they have a different name to allow for users to switch back and forth.

Custom vindexes can also be plugged in as needed.

Vitess API Reference

noToc: true

This document describes Vitess API methods that enable your client application to more easily talk to your storage system to query data. API methods are grouped into the following categories:

- Range-based Sharding
- Transactions
- Custom Sharding
- Map Reduce
- Topology
- v3 API (alpha)

The following table lists the methods in each group and links to more detail about each method:

Range-based Sharding

ExecuteBatchKeyspaceIds

ExecuteBatchKeyspaceIds executes the list of queries based on the specified keyspace ids.

ExecuteEntityIds

ExecuteEntityIds executes the query based on the specified external id to keyspace id map.

ExecuteKeyRanges

ExecuteKeyRanges executes the query based on the specified key ranges.

ExecuteKeyspaceIds

ExecuteKeyspaceIds executes the query based on the specified keyspace ids.

StreamExecuteKeyRanges

StreamExecuteKeyRanges executes a streaming query based on key ranges. Use this method if the query returns a large number of rows.

StreamExecuteKeyspaceIds

StreamExecuteKeyspaceIds executes a streaming query based on keyspace ids. Use this method if the query returns a large number of rows.

Transactions

Begin

Begin a transaction.

Commit

Commit a transaction.

ResolveTransaction

ResolveTransaction resolves a transaction.

Rollback

Rollback a transaction.

Custom Sharding

ExecuteBatchShards

ExecuteBatchShards executes the list of queries on the specified shards.

ExecuteShards

ExecuteShards executes the query on the specified shards.

StreamExecuteShards

StreamExecuteShards executes a streaming query based on shards. Use this method if the query returns a large number of rows.

Map Reduce

SplitQuery

Split a query into non-overlapping sub queries

Topology

GetSrvKeyspace

GetSrvKeyspace returns a SrvKeyspace object (as seen by this vtgate). This method is provided as a convenient way for clients to take a look at the sharding configuration for a Keyspace. Looking at the sharding information should not be used for routing queries (as the information may change, use the Execute calls for that). It is convenient for monitoring applications for instance, or if using custom sharding.

v3 API (alpha)

Execute

Execute tries to route the query to the right shard. It depends on the query and bind variables to provide enough information in conjonction with the vindexes to route the query.

StreamExecute

StreamExecute executes a streaming query based on shards. It depends on the query and bind variables to provide enough information in conjonction with the vindexes to route the query. Use this method if the query returns a large number of rows.

Range-based Sharding

ExecuteBatchKeyspaceIds ExecuteBatchKeyspaceIds executes the list of queries based on the specified keyspace ids.

Request ExecuteBatchKeyspaceIdsRequest is the payload to ExecuteBatchKeyspaceId.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
queries list <BoundKeyspaceIdQuery>	BoundKeyspaceIdQuery represents a single query request for the specified list of keyspace ids. This is used in a list for ExecuteBatchKeyspaceIdsRequest.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
as_transaction bool	as_transaction will execute the queries in this batch in a single transaction per shard, created for this purpose. (this can be seen as adding a 'begin' before and 'commit' after the queries). Only makes sense if tablet_type is master. If set, the Session is ignored.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteBatchKeyspaceIdsResponse is the returned value from ExecuteBatchKeyspaceId.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
results list <query.QueryResult>	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

ExecuteEntityIds ExecuteEntityIds executes the query based on the specified external id to keyspace id map.

Request ExecuteEntityIdsRequest is the payload to ExecuteEntityIds.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
entity_column_name string	entity_column_name is the column name to use.
entity_keyspace_ids list <EntityId>	entity_keyspace_ids are pairs of entity_column_name values associated with its corresponding keyspace_id.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
not_in_transaction bool	not_in_transaction is deprecated and should not be used.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Messages ExecuteEntityIdsRequest.EntityId

Properties

Name	Description
type query.Type	Type defines the various supported data types in bind vars and query results.
value bytes	value is the value for the entity. Not set if type is NULL_TYPE.

Name	Description
keyspace_id bytes	keyspace_id is the associated keyspace_id for the entity.

Response ExecuteEntityIdsResponse is the returned value from ExecuteEntityIds.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

ExecuteKeyRanges ExecuteKeyRanges executes the query based on the specified key ranges.

Request ExecuteKeyRangesRequest is the payload to ExecuteKeyRanges.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to
key_ranges list <topodata.KeyRange>	KeyRange describes a range of sharding keys, when range-based sharding is used.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
not_in_transaction bool	not_in_transaction is deprecated and should not be used.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteKeyRangesResponse is the returned value from ExecuteKeyRanges.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

ExecuteKeyspaceIds ExecuteKeyspaceIds executes the query based on the specified keyspace ids.

Request ExecuteKeyspaceIdsRequest is the payload to ExecuteKeyspaceIds.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
keyspace_ids list <bytes>	keyspace_ids contains the list of keyspace_ids affected by this query. Will be used to find the shards to send the query to.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
not_in_transaction bool	not_in_transaction is deprecated and should not be used.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteKeyspaceIdsResponse is the returned value from ExecuteKeyspaceIds.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.

Name	Description
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

StreamExecuteKeyRanges StreamExecuteKeyRanges executes a streaming query based on key ranges. Use this method if the query returns a large number of rows.

Request StreamExecuteKeyRangesRequest is the payload to StreamExecuteKeyRanges.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
key_ranges list <topodata.KeyRange>	KeyRange describes a range of sharding keys, when range-based sharding is used.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response StreamExecuteKeyRangesResponse is the returned value from StreamExecuteKeyRanges.

Properties

Name	Description
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

StreamExecuteKeyspaceIds StreamExecuteKeyspaceIds executes a streaming query based on keyspace ids. Use this method if the query returns a large number of rows.

Request StreamExecuteKeyspaceIdsRequest is the payload to StreamExecuteKeyspaceIds.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
keyspace_ids list <bytes>	keyspace_ids contains the list of keyspace_ids affected by this query. Will be used to find the shards to send the query to.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response StreamExecuteKeyspaceIdsResponse is the returned value from StreamExecuteKeyspaceIds.

Properties

Name	Description
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

Transactions

Begin Begin a transaction.

Request BeginRequest is the payload to Begin.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
single_db bool	single_db specifies if the transaction should be restricted to a single database.

Response BeginResponse is the returned value from Begin.

Properties

Name	Description
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.

Commit Commit a transaction.

Request CommitRequest is the payload to Commit.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
atomic bool	atomic specifies if the commit should go through the 2PC workflow to ensure atomicity.

Response CommitResponse is the returned value from Commit.

Properties

Name	Description

ResolveTransaction ResolveTransaction resolves a transaction.

Request ResolveTransactionRequest is the payload to ResolveTransaction.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
dtid string	dtid is the dtid of the transaction to be resolved.

Response ResolveTransactionResponse is the returned value from Rollback.

Properties

Name	Description

Rollback Rollback a transaction.

Request RollbackRequest is the payload to Rollback.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.

Response RollbackResponse is the returned value from Rollback.

Properties

Name	Description

Custom Sharding

ExecuteBatchShards ExecuteBatchShards executes the list of queries on the specified shards.

Request ExecuteBatchShardsRequest is the payload to ExecuteBatchShards

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.

Name	Description
queries list <BoundShardQuery>	BoundShardQuery represents a single query request for the specified list of shards. This is used in a list for ExecuteBatchShardsRequest.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
as_transaction bool	as_transaction will execute the queries in this batch in a single transaction per shard, created for this purpose. (this can be seen as adding a 'begin' before and 'commit' after the queries). Only makes sense if tablet_type is master. If set, the Session is ignored.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteBatchShardsResponse is the returned value from ExecuteBatchShards.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
results list <query.QueryResult>	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

ExecuteShards ExecuteShards executes the query on the specified shards.

Request ExecuteShardsRequest is the payload to ExecuteShards.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
shards list <string>	shards to target the query to. A DML can only target one shard.

Name	Description
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
not_in_transaction bool	not_in_transaction is deprecated and should not be used.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteShardsResponse is the returned value from ExecuteShards.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

StreamExecuteShards StreamExecuteShards executes a streaming query based on shards. Use this method if the query returns a large number of rows.

Request StreamExecuteShardsRequest is the payload to StreamExecuteShards.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
shards list <string>	shards to target the query to.
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response StreamExecuteShardsResponse is the returned value from StreamExecuteShards.

Properties

Name	Description
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

Map Reduce

SplitQuery Split a query into non-overlapping sub queries

Request SplitQueryRequest is the payload to SplitQuery. SplitQuery takes a “SELECT” query and generates a list of queries called “query-parts”. Each query-part consists of the original query with an added WHERE clause that restricts the query-part to operate only on rows whose values in the the columns listed in the “split_column” field of the request (see below) are in a particular range. It is guaranteed that the set of rows obtained from executing each query-part on a database snapshot and merging (without deduping) the results is equal to the set of rows obtained from executing the original query on the same snapshot with the rows containing NULL values in any of the split_column’s excluded. This is typically called by the MapReduce master when reading from Vitess. There it’s desirable that the sets of rows returned by the query-parts have roughly the same size.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
keyspace string	keyspace to target the query to.
query query.BoundQuery	BoundQuery is a query with its bind variables
split_column list <string>	Each generated query-part will be restricted to rows whose values in the columns listed in this field are in a particular range. The list of columns named here must be a prefix of the list of columns defining some index or primary key of the table referenced in ‘query’. For many tables using the primary key columns (in order) is sufficient and this is the default if this field is omitted. See the comment on the ‘algorithm’ field for more restrictions and information.
split_count int64	You can specify either an estimate of the number of query-parts to generate or an estimate of the number of rows each query-part should return. Thus, exactly one of split_count or num_rows_per_query_part should be nonzero. The non-given parameter is calculated from the given parameter using the formula: split_count * num_rows_per_query_pary = table_size, where table_size is an approximation of the number of rows in the table. Note that if “split_count” is given it is regarded as an estimate. The number of query-parts returned may differ slightly (in particular, if it’s not a whole multiple of the number of vitess shards).

Name	Description
num_rows_per_query_part int64	
algorithm query.SplitQueryRequest.Algorithm	The algorithm to use to split the query. The split algorithm is performed on each database shard in parallel. The lists of query-parts generated by the shards are merged and returned to the caller. Two algorithms are supported: EQUAL_SPLITS If this algorithm is selected then only the first 'split_column' given is used (or the first primary key column if the 'split_column' field is empty). In the rest of this algorithm's description, we refer to this column as "the split column". The split column must have numeric type (integral or floating point). The algorithm works by taking the interval [min, max], where min and max are the minimum and maximum values of the split column in the table-shard, respectively, and partitioning it into 'split_count' sub-intervals of equal size. The added WHERE clause of each query-part restricts that part to rows whose value in the split column belongs to a particular sub-interval. This is fast, but requires that the distribution of values of the split column be uniform in [min, max] for the number of rows returned by each query part to be roughly the same. FULL_SCAN If this algorithm is used then the split_column must be the primary key columns (in order). This algorithm performs a full-scan of the table-shard referenced in 'query' to get "boundary" rows that are num_rows_per_query_part apart when the table is ordered by the columns listed in 'split_column'. It then restricts each query-part to the rows located between two successive boundary rows. This algorithm supports multiple split_column's of any type, but is slower than EQUAL_SPLITS.
use_split_query_v2 bool	Remove this field after this new server code is released to prod. We must keep it for now, so that clients can still send it to the old server code currently in production.

Response SplitQueryResponse is the returned value from SplitQuery.

Properties

Name	Description
splits list <Part>	splits contains the queries to run to fetch the entire data set.

Messages SplitQueryResponse.KeyRangePart

Properties

Name	Description
keyspace string	keyspace to target the query to.
key_ranges list <topodata.KeyRange>	KeyRange describes a range of sharding keys, when range-based sharding is used.

SplitQueryResponse.Part

Properties

Name	Description
query query.BoundQuery	BoundQuery is a query with its bind variables
key_range_part KeyRangePart	key_range_part is set if the query should be executed by ExecuteKeyRanges.
shard_part ShardPart	shard_part is set if the query should be executed by ExecuteShards.
size int64	size is the approximate number of rows this query will return.

SplitQueryResponse.ShardPart

Properties

Name	Description
keyspace string	keyspace to target the query to.
shards list <string>	shards to target the query to.

Topology

GetSrvKeyspace GetSrvKeyspace returns a SrvKeyspace object (as seen by this vtgate). This method is provided as a convenient way for clients to take a look at the sharding configuration for a Keyspace. Looking at the sharding information should not be used for routing queries (as the information may change, use the Execute calls for that). It is convenient for monitoring applications for instance, or if using custom sharding.

Request GetSrvKeyspaceRequest is the payload to GetSrvKeyspace.

Parameters

Name	Description
keyspace string	keyspace name to fetch.

Response GetSrvKeyspaceResponse is the returned value from GetSrvKeyspace.

Properties

Name	Description
srv_keyspace topodata.SrvKeyspace	SrvKeyspace is a rollup node for the keyspace itself.

v3 API (alpha)

Execute Execute tries to route the query to the right shard. It depends on the query and bind variables to provide enough information in conjonction with the vindexes to route the query.

Request ExecuteRequest is the payload to Execute.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
query query.BoundQuery	BoundQuery is a query with its bind variables
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
not_in_transaction bool	not_in_transaction is deprecated and should not be used.
keyspace string	keyspace to target the query to.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response ExecuteResponse is the returned value from Execute.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
session Session	Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

StreamExecute StreamExecute executes a streaming query based on shards. It depends on the query and bind variables to provide enough information in conjonction with the vindexes to route the query. Use this method if the query returns a large number of rows.

Request StreamExecuteRequest is the payload to StreamExecute.

Parameters

Name	Description
caller_id vtrpc.CallerID	CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.
query query.BoundQuery	BoundQuery is a query with its bind variables
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.
keyspace string	keyspace to target the query to.
options query.ExecuteOptions	ExecuteOptions is passed around for all Execute calls.

Response StreamExecuteResponse is the returned value from StreamExecute.

Properties

Name	Description
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

Enums

query.Type Type defines the various supported data types in bind vars and query results.

Name	Value	Description
NULL_TYPE	0	NULL_TYPE specifies a NULL type.
INT8	257	INT8 specifies a TINYINT type. Properties: 1, IsNumber.
UINT8	770	UINT8 specifies a TINYINT UNSIGNED type. Properties: 2, IsNumber, IsUnsigned.
INT16	259	INT16 specifies a SMALLINT type. Properties: 3, IsNumber.
UINT16	772	UINT16 specifies a SMALLINT UNSIGNED type. Properties: 4, IsNumber, IsUnsigned.
INT24	261	INT24 specifies a MEDIUMINT type. Properties: 5, IsNumber.
UINT24	774	UINT24 specifies a MEDIUMINT UNSIGNED type. Properties: 6, IsNumber, IsUnsigned.
INT32	263	INT32 specifies a INTEGER type. Properties: 7, IsNumber.
UINT32	776	UINT32 specifies a INTEGER UNSIGNED type. Properties: 8, IsNumber, IsUnsigned.

Name	Value	Description
INT64	265	INT64 specifies a BIGINT type. Properties: 9, IsNumber.
UINT64	778	UINT64 specifies a BIGINT UNSIGNED type. Properties: 10, IsNumber, IsUnsigned.
FLOAT32	1035	FLOAT32 specifies a FLOAT type. Properties: 11, IsFloat.
FLOAT64	1036	FLOAT64 specifies a DOUBLE or REAL type. Properties: 12, IsFloat.
TIMESTAMP	2061	TIMESTAMP specifies a TIMESTAMP type. Properties: 13, IsQuoted.
DATE	2062	DATE specifies a DATE type. Properties: 14, IsQuoted.
TIME	2063	TIME specifies a TIME type. Properties: 15, IsQuoted.
DATETIME	2064	DATETIME specifies a DATETIME type. Properties: 16, IsQuoted.
YEAR	785	YEAR specifies a YEAR type. Properties: 17, IsNumber, IsUnsigned.
DECIMAL	18	DECIMAL specifies a DECIMAL or NUMERIC type. Properties: 18, None.
TEXT	6163	TEXT specifies a TEXT type. Properties: 19, IsQuoted, IsText.
BLOB	10260	BLOB specifies a BLOB type. Properties: 20, IsQuoted, IsBinary.
VARCHAR	6165	VARCHAR specifies a VARCHAR type. Properties: 21, IsQuoted, IsText.
VARBINARY	10262	VARBINARY specifies a VARBINARY type. Properties: 22, IsQuoted, IsBinary.
CHAR	6167	CHAR specifies a CHAR type. Properties: 23, IsQuoted, IsText.
BINARY	10264	BINARY specifies a BINARY type. Properties: 24, IsQuoted, IsBinary.
BIT	2073	BIT specifies a BIT type. Properties: 25, IsQuoted.
ENUM	2074	ENUM specifies an ENUM type. Properties: 26, IsQuoted.
SET	2075	SET specifies a SET type. Properties: 27, IsQuoted.
TUPLE	28	TUPLE specifies a a tuple. This cannot be returned in a QueryResult, but it can be sent as a bind var. Properties: 28, None.
GEOMETRY	2077	GEOMETRY specifies a GEOMETRY type. Properties: 29, IsQuoted.
JSON	2078	JSON specified a JSON type. Properties: 30, IsQuoted.

topodata.KeyspaceIdType KeyspaceIdType describes the type of the sharding key for a range-based sharded keyspace.

Name	Value	Description
UNSET	0	UNSET is the default value, when range-based sharding is not used.
UINT64	1	UINT64 is when uint64 value is used. This is represented as `UNSIGNED BIGINT` in MySQL
BYTES	2	BYTES is when an array of bytes is used. This is represented as `VARBINARY` in MySQL

topodata.TabletType TabletType represents the type of a given tablet.

Name	Value	Description
UNKNOWN	0	UNKNOWN is not a valid value.
MASTER	1	MASTER is the master server for the shard. Only MASTER allows DMLs.
REPLICA	2	REPLICA is a slave type. It is used to serve live traffic. A REPLICA can be promoted to MASTER. A demoted MASTER will go to REPLICA.
RDONLY	3	RDONLY (old name) / BATCH (new name) is used to serve traffic for long-running jobs. It is a separate type from REPLICA so long-running queries don't affect web-like traffic.
BATCH	3	
SPARE	4	SPARE is a type of servers that cannot serve queries, but is available in case an extra server is needed.
EXPERIMENTAL	5	EXPERIMENTAL is like SPARE, except it can serve queries. This type can be used for usages not planned by Vitess, like online export to another storage engine.
BACKUP	6	BACKUP is the type a server goes to when taking a backup. No queries can be served in BACKUP mode.
RESTORE	7	RESTORE is the type a server uses when restoring a backup, at startup time. No queries can be served in RESTORE mode.
DRAINED	8	DRAINED is the type a server goes into when used by Vitess tools to perform an offline action. It is a serving type (as the tools processes may need to run queries), but it's not used to route queries from Vitess users. In this state, this tablet is dedicated to the process that uses it.

vtrpc.ErrorCode ErrorCode is the enum values for Errors. Internally, errors should be created with one of these codes. These will then be translated over the wire by various RPC frameworks.

Name	Value	Description
SUCCESS	0	SUCCESS is returned from a successful call.
CANCELLED	1	CANCELLED means that the context was cancelled (and noticed in the app layer, as opposed to the RPC layer).
UNKNOWN_ERROR	2	UNKNOWN_ERROR includes: 1. MySQL error codes that we don't explicitly handle. 2. MySQL response that wasn't as expected. For example, we might expect a MySQL timestamp to be returned in a particular way, but it wasn't. 3. Anything else that doesn't fall into a different bucket.
BAD_INPUT	3	BAD_INPUT is returned when an end-user either sends SQL that couldn't be parsed correctly, or tries a query that isn't supported by Vitess.
DEADLINE_EXCEEDED	4	DEADLINE_EXCEEDED is returned when an action is taking longer than a given timeout.
INTEGRITY_ERROR	5	INTEGRITY_ERROR is returned on integrity error from MySQL, usually due to duplicate primary keys.
PERMISSION_DENIED	6	PERMISSION_DENIED errors are returned when a user requests access to something that they don't have permissions for.
RESOURCE_EXHAUSTED	7	RESOURCE_EXHAUSTED is returned when a query exceeds its quota in some dimension and can't be completed due to that. Queries that return RESOURCE_EXHAUSTED should not be retried, as it could be detrimental to the server's health. Examples of errors that will cause the RESOURCE_EXHAUSTED code: 1. TxPoolFull: this is retried server-side, and is only returned as an error if the server-side retries failed. 2. Query is killed due to it taking too long.
QUERY_NOT_SERVED	8	QUERY_NOT_SERVED means that a query could not be served right now. Client can interpret it as: "the tablet that you sent this query to cannot serve the query right now, try a different tablet or try again later." This could be due to various reasons: QueryService is not serving, should not be serving, wrong shard, wrong tablet type, blacklisted table, etc. Clients that receive this error should usually retry the query, but after taking the appropriate steps to make sure that the query will get sent to the correct tablet.

Name	Value	Description
NOT_IN_TX	9	NOT_IN_TX means that we're not currently in a transaction, but we should be.
INTERNAL_ERROR	10	INTERNAL_ERRORs are problems that only the server can fix, not the client. These errors are not due to a query itself, but rather due to the state of the system. Generally, we don't expect the errors to go away by themselves, but they may go away after human intervention. Examples of scenarios where INTERNAL_ERROR is returned: 1. Something is not configured correctly internally. 2. A necessary resource is not available, and we don't expect it to become available by itself. 3. A sanity check fails. 4. Some other internal error occurs. Clients should not retry immediately, as there is little chance of success. However, it's acceptable for retries to happen internally, for example to multiple backends, in case only a subset of backend are not functional.
TRANSIENT_ERROR	11	TRANSIENT_ERROR is used for when there is some error that we expect we can recover from automatically - often due to a resource limit temporarily being reached. Retrying this error, with an exponential backoff, should succeed. Clients should be able to successfully retry the query on the same backends. Examples of things that can trigger this error: 1. Query has been throttled 2. VtGate could have request backlog
UNAUTHENTICATED	12	UNAUTHENTICATED errors are returned when a user requests access to something, and we're unable to verify the user's authentication.

Messages

BoundKeyspaceIdQuery BoundKeyspaceIdQuery represents a single query request for the specified list of keyspace ids. This is used in a list for ExecuteBatchKeyspaceIdsRequest.

Properties

Name	Description
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.

Name	Description
keyspace_ids list <bytes>	keyspace_ids contains the list of keyspace_ids affected by this query. Will be used to find the shards to send the query to.

BoundShardQuery BoundShardQuery represents a single query request for the specified list of shards. This is used in a list for ExecuteBatchShardsRequest.

Properties

Name	Description
query query.BoundQuery	BoundQuery is a query with its bind variables
keyspace string	keyspace to target the query to.
shards list <string>	shards to target the query to. A DML can only target one shard.

Session Session objects are session cookies and are invalidated on use. Query results will contain updated session values. Their content should be opaque to the user.

Properties

Name	Description
in_transaction bool	
shard_sessions list <ShardSession>	
single_db bool	single_db specifies if the transaction should be restricted to a single database.

Messages Session.ShardSession

Properties

Name	Description
target query.Target	Target describes what the client expects the tablet is. If the tablet does not match, an error is returned.
transaction_id int64	

query.BindVariable BindVariable represents a single bind variable in a Query.

Properties

Name	Description
type Type	
value bytes	
values list <Value>	Value represents a typed value.

query.BoundQuery BoundQuery is a query with its bind variables

Properties

Name	Description
sql string	sql is the SQL query to execute
bind_variables map <string, BindVariable>	bind_variables is a map of all bind variables to expand in the query

query.EventToken EventToken is a structure that describes a point in time in a replication stream on one shard. The most recent known replication position can be retrieved from vttablet when executing a query. It is also sent with the replication streams from the binlog service.

Properties

Name	Description
timestamp int64	timestamp is the MySQL timestamp of the statements. Seconds since Epoch.
shard string	The shard name that applied the statements. Note this is not set when streaming from a vttablet. It is only used on the client -> vtgate link.
position string	The position on the replication stream after this statement was applied. It is not the transaction ID / GTID, but the position / GTIDSet.

query.ExecuteOptions ExecuteOptions is passed around for all Execute calls.

Properties

Name	Description
include_event_token bool	This used to be exclude_field_names, which was replaced by IncludedFields enum below If set, we will try to include an EventToken with the responses.
compare_event_token EventToken	EventToken is a structure that describes a point in time in a replication stream on one shard. The most recent known replication position can be retrieved from vttablet when executing a query. It is also sent with the replication streams from the binlog service.
included_fields IncludedFields	Controls what fields are returned in Field message responses from MySQL, i.e. field name, table name, etc. This is an optimization for high-QPS queries where the client knows what it's getting

Enums ExecuteOptions.IncludedFields

Name	Value	Description
TYPE_AND_NAME	0	
TYPE_ONLY	1	
ALL	2	

query.Field Field describes a single column returned by a query

Properties

Name	Description
name string	name of the field as returned by mysql C API
type Type	vitess-defined type. Conversion function is in sqltypes package.
table string	Remaining fields from mysql C API. These fields are only populated when ExecuteOptions.included_fields is set to IncludedFields.ALL.
org_table string	
database string	
org_name string	
column_length uint32	column_length is really a uint32. All 32 bits can be used.
charset uint32	charset is actually a uint16. Only the lower 16 bits are used.
decimals uint32	decimals is actualy a uint8. Only the lower 8 bits are used.
flags uint32	flags is actually a uint16. Only the lower 16 bits are used.

query.QueryResult QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

Properties

Name	Description
fields list <Field>	Field describes a single column returned by a query
rows_affected uint64	
insert_id uint64	
rows list <Row>	Row is a database row.
extras ResultExtras	ResultExtras contains optional out-of-band information. Usually the extras are requested by adding ExecuteOptions flags.

query.ResultExtras ResultExtras contains optional out-of-band information. Usually the extras are requested by adding ExecuteOptions flags.

Properties

Name	Description
event_token EventToken	EventToken is a structure that describes a point in time in a replication stream on one shard. The most recent known replication position can be retrieved from vttablet when executing a query. It is also sent with the replication streams from the binlog service.
fresher bool	If set, it means the data returned with this result is fresher than the compare_token passed in the ExecuteOptions.

query.ResultWithError ResultWithError represents a query response in the form of result or error but not both.

Properties

Name	Description
error vtrpc.RPCError	RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.
result query.QueryResult	QueryResult is returned by Execute and ExecuteStream. As returned by Execute, len(fields) is always equal to len(row) (for each row in rows). As returned by StreamExecute, the first QueryResult has the fields set, and subsequent QueryResult have rows set. And as Execute, len(QueryResult[0].fields) is always equal to len(row) (for each row in rows for each QueryResult in QueryResult[1:]).

query.Row Row is a database row.

Properties

Name	Description
lengths list <sint64>	lengths contains the length of each value in values. A length of -1 means that the field is NULL. While reading values, you have to accummulate the length to know the offset where the next value begins in values.
values bytes	values contains a concatenation of all values in the row.

query.StreamEvent StreamEvent describes a set of transformations that happened as a single transactional unit on a server. It is streamed back by the Update Stream calls.

Properties

Name	Description
statements list <Statement>	The statements in this transaction.
event_token EventToken	EventToken is a structure that describes a point in time in a replication stream on one shard. The most recent known replication position can be retrieved from vttablet when executing a query. It is also sent with the replication streams from the binlog service.

Messages StreamEvent.Statement

One individual Statement in a transaction.

Properties

Name	Description
category Category	

Name	Description
table_name string	table_name, primary_key_fields and primary_key_values are set for DML.
primary_key_fields list <Field>	Field describes a single column returned by a query
primary_key_values list <Row>	Row is a database row.
sql bytes	sql is set for all queries. FIXME(alainjobart) we may not need it for DMLs.

Enums StreamEvent.Statement.Category

One individual Statement in a transaction. The category of one statement.

Name	Value	Description
Error	0	
DML	1	
DDL	2	

query.Target Target describes what the client expects the tablet is. If the tablet does not match, an error is returned.

Properties

Name	Description
keyspace string	
shard string	
tablet_type topodata.TabletType	TabletType represents the type of a given tablet.

query.Value Value represents a typed value.

Properties

Name	Description
type Type	
value bytes	

topodata.KeyRange KeyRange describes a range of sharding keys, when range-based sharding is used.

Properties

Name	Description
start bytes	
end bytes	

topodata.ShardReference ShardReference is used as a pointer from a SrvKeyspace to a Shard

Properties

Name	Description
name string	Copied from Shard.
key_range KeyRange	KeyRange describes a range of sharding keys, when range-based sharding is used.

topodata.SrvKeyspace SrvKeyspace is a rollup node for the keyspace itself.

Properties

Name	Description
partitions list <KeyspacePartition>	The partitions this keyspace is serving, per tablet type.
sharding_column_name string	copied from Keyspace
sharding_column_type KeyspaceIdType	
served_from list <ServedFrom>	

Messages SrvKeyspace.KeyspacePartition

Properties

Name	Description
served_type TabletType	The type this partition applies to.
shard_references list <ShardReference>	ShardReference is used as a pointer from a SrvKeyspace to a Shard

SrvKeyspace.ServedFrom

ServedFrom indicates a relationship between a TabletType and the keyspace name that's serving it.

Properties

Name	Description
tablet_type TabletType	ServedFrom indicates a relationship between a TabletType and the keyspace name that's serving it. the tablet type
keyspace string	the keyspace name that's serving it

vtrpc.CallerID CallerID is passed along RPCs to identify the originating client for a request. It is not meant to be secure, but only informational. The client can put whatever info they want in these fields, and they will be trusted by the servers. The fields will just be used for logging purposes, and to easily find a client. VtGate propagates it to VtTablet, and VtTablet may use this information for monitoring purposes, to display on dashboards, or for blacklisting purposes.

Properties

Name	Description
principal string	principal is the effective user identifier. It is usually filled in with whoever made the request to the appserver, if the request came from an automated job or another system component. If the request comes directly from the Internet, or if the Vitess client takes action on its own accord, it is okay for this field to be absent.

Name	Description
component string	component describes the running process of the effective caller. It can for instance be the hostname:port of the servlet initiating the database call, or the container engine ID used by the servlet.
subcomponent string	subcomponent describes a component inisde the immediate caller which is responsible for generating is request. Suggested values are a servlet name or an API endpoint name.

vtrpc.RPCError RPCError is an application-level error structure returned by VtTablet (and passed along by VtGate if appropriate). We use this so the clients don't have to parse the error messages, but instead can depend on the value of the code.

Properties

Name	Description
code ErrorCode	
message string	

Vitess 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:

```sql
/* 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;
```

VReplication

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 manage the switching from a backfill mode to the vindex itself keeping it up-to-date.
- **Schema deployment**: We can use VReplication 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 if something went wrong.
- **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's still possible to directly issue commands to do some of the activities.

Feature description

VReplication works as a stream or combination 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 necessary goals. This is because there will be multiple source shards as well as 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 large data, 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's 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 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 (to be implemented): 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 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 `vreplication` table in the `vt` database. VReplication uses the 'pull' model. This means that a stream is created on the target side, and the target pulls the data by finding the appropriate source.

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.

```
lvtctl.sh 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 = [
  './lvtctl.sh',
  '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.

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.
- `time_updated`: 0, reserved.
- `transaction_timestamp`: 0, reserved.
- `state`: 'Running' or 'Stopped'.
- `cell`: is an optional parameter that specifies the cell from which the stream can be sourced.

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: replicate all columns and rows of product from `lookup/0.product` into 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: replicate all columns of `user/-80.uorder` where `unicode_loose_md5(mname)` is within `-80` keyrange, into `morder`.

This particular stream generally wouldn't make sense in isolation. This would typically be one of four streams that combine together 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 are not supported. Among aggregate expressions, only `count(*)` and `sum(col)` are supported.
- The where clause can only contain the `in_keyrange` construct. It has two forms:
  - `in_keyrange('-80')`: The row's source keyrange matched against `-80`.
  - `in_keyrange(col, 'hash', '-80')`: The keyrange is computed using `hash(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`, joins, 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.

It's a shared row The vreplication 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 is updated as `Init` or `Copying`.

Updating a stream You can change any field of the stream by issuing a `VReplicationExec` with an `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.

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.

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.

VSchema

VSchemas describe how to shard data

VSchema stands for Vitess Schema. In contrast to a traditional database schema that contains metadata about tables, a VSchema contains metadata about how tables are organized across keyspaces and shards. Simply put, it contains the information needed to make Vitess look like a single database server.

For example, the 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.

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, 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.

Configuration

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.

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 showcase 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": "user_seq"
      }
    }
  }
}
```

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.

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`.

vtctl Reference

noToc: true

This reference guide explains the commands that the vtctl tool supports. **vtctl** is a command-line tool used to administer a Vitess cluster, and it allows a human or application to easily interact with a Vitess implementation.

Commands are listed in the following groups:

- Cells
- Generic
- Keyspaces
- Queries
- Replication Graph
- Resharding Throttler

- Schema, Version, Permissions
- Serving Graph
- Shards
- Tablets
- Topo
- Workflows

Cells

- AddCellInfo
- DeleteCellInfo
- GetCellInfo
- GetCellInfoNames
- UpdateCellInfo
- AddCellsAlias
- DeleteCellsAlias
- UpdateCellsAlias
- GetCellsAliases

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.

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.

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.

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.

GetCellsAliases Fetches in json format all the existent cells alias groups.

Example

Generic

- ListAllTablets
- ListTablets
- Validate

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.

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

Keyspaces

- CreateKeyspace
- DeleteKeyspace
- FindAllShardsInKeyspace
- GetKeyspace
- GetKeyspaces
- MigrateServedFrom
- MigrateServedTypes
- CancelResharding
- ShowResharding
- RebuildKeyspaceGraph
- RemoveKeyspaceCell
- SetKeyspaceServedFrom
- SetKeyspaceShardingInfo
- ValidateKeyspace
- WaitForDrain

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.

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.

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.

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
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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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.

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
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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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

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>.

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.

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.

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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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.

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

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.

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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.

    - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
      Use this command to exit this mode:
    - spare – A slaved 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.

Queries

- VtGateExecute
- VtGateExecuteKeyspaceIds
- VtGateExecuteShards
- VtGateSplitQuery
- VtTabletBegin
- VtTabletCommit
- VtTabletExecute
- VtTabletRollback
- VtTabletStreamHealth
- VtTabletUpdateStream

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

VtGateExecuteKeyspaceIds Executes the given SQL query with the provided bound variables against the vtgate server. It is routed to the shards that contain the provided keyspace ids.

Example

Flags

Name	Type	Definition
json	Boolean	Output JSON instead of human-readable table
keyspace	string	keyspace to send query to
keyspace_ids	string	comma-separated list of keyspace ids (in hex) that will map into shards to send query to
options	string	execute options values as a text encoded proto of the ExecuteOptions structure
server	string	VtGate server to connect to
tablet_type	string	tablet type to query

Arguments

- <vtgate> – Required.
- <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.
- <ks1 in hex> – Required. To specify multiple values for this argument, separate individual values with a comma.
- <sql> – Required.

Errors

- the <sql> argument is required for the <VtGateExecuteKeyspaceIds> 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 hex-decode value %v '%v': %v
- error connecting to vtgate '%v': %v
- Execute failed: %v

VtGateExecuteShards Executes the given SQL query with the provided bound variables against the vtgate server. It is routed to the provided shards.

Example

Flags

Name	Type	Definition
json	Boolean	Output JSON instead of human-readable table
keyspace	string	keyspace to send query to

Name	Type	Definition
options	string	execute options values as a text encoded proto of the ExecuteOptions structure
server	string	VtGate server to connect to
shards	string	comma-separated list of shards to send query to
tablet_type	string	tablet type to query

Arguments

- <vtgate> – Required.
- <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.
- <shard> – Required. The name of a shard. The argument value is typically in the format <range start>-<range end>. To specify multiple values for this argument, separate individual values with a comma.
- <sql> – Required.

Errors

- the <sql> argument is required for the <VtGateExecuteShards> 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

VtGateSplitQuery Executes the SplitQuery computation for the given SQL query with the provided bound variables against the vtgate server (this is the base query for Map-Reduce workloads, and is provided here for debug / test purposes).

Example

Flags

Name	Type	Definition
algorithm	string	The algorithm to
keyspace	string	keyspace to send query to
server	string	VtGate server to connect to
split_count	Int64	number of splits to generate.

Arguments

- <vtgate> – Required.
- <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.
- <split_count> – Required.
- <sql> – Required.

Errors

- the <sql> argument is required for the <VtGateSplitQuery> 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)
- Exactly one of <split_count> or num_rows_per_query_part
- Unknown split-query <algorithm>: %v
- error connecting to vtgate '%v': %v
- Execute failed: %v
- SplitQuery 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

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

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

VtTabletUpdateStream Executes the UpdateStream 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
position	string	position to start the stream from
timestamp	Int	timestamp to start the stream from

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 <VtTabletUpdateStream> 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

Replication Graph

- GetShardReplication

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.

Resharding Throttler

- GetThrottlerConfiguration
- ResetThrottlerConfiguration
- ThrottlerMaxRates
- ThrottlerSetMaxRate
- UpdateThrottlerConfiguration

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

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

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

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

Schema, Version, Permissions

- ApplySchema
- ApplyVSchema
- CopySchemaShard
- GetPermissions
- GetSchema
- GetVSchema
- RebuildVSchemaGraph
- ReloadSchema
- ReloadSchemaKeyspace
- ReloadSchemaShard
- ValidatePermissionsKeyspace
- ValidatePermissionsShard
- ValidateSchemaKeyspace
- ValidateSchemaShard
- ValidateVersionKeyspace
- ValidateVersionShard

ApplySchema Applies the schema change to the specified keyspace on every master, running in parallel on all shards. The changes are then propagated to slaves 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

Name	Type	Definition
wait_slave_timeout	Duration	The amount of time to wait for slaves 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.

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.
skip_rebuild	Boolean	If set, do no rebuild the SrvSchema objects.
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

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/
wait_slave_timeout	Duration	The amount of time to wait for slaves 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.

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.

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.

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.

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.

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.

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.

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.

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.

ValidatePermissionsShard Validates that the master permissions match all the slaves.

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.

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.

ValidateSchemaShard Validates that the master schema matches all of the slaves.

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.

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.

ValidateVersionShard Validates that the master version matches all of the slaves.

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.

Serving Graph

- GetSrvKeyspace
- GetSrvKeyspaceNames
- GetSrvVSchema

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.

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.

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.

Shards

- CreateShard
- DeleteShard
- EmergencyReparentShard
- GetShard
- InitShardMaster
- ListBackups
- ListShardTablets
- PlannedReparentShard
- RemoveBackup
- RemoveShardCell
- SetShardServedTypes
- SetShardTabletControl
- ShardReplicationFix
- ShardReplicationPositions
- SourceShardAdd
- SourceShardDelete
- TabletExternallyReparented
- ValidateShard
- WaitForFilteredReplication

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.

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.

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_slave_timeout	Duration	time to wait for slaves 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

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.

InitShardMaster Sets the initial master for a shard. Will make all other tablets in the shard slaves 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_slave_timeout	Duration	time to wait for slaves 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)

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.

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.

PlannedReparentShard Reparents the shard to the new master, or away from old master. Both old and new master need to be up and running.

Example

Flags

Name	Type	Definition
avoid_master	string	alias of a tablet that should not be the master, i.e. reparent to any other tablet if this one is the master
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_slave_timeout	Duration	time to wait for slaves 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

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.

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.

SetShardServedTypes Add or remove served type to/from a shard. This is meant as an emergency function. It does not rebuild any serving graph i.e. does not run 'RebuildKeyspaceGraph'.

Example

Flags

Name	Type	Definition
cells	string	Specifies a comma-separated list of cells to update
remove	Boolean	Removes the served tablet type

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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved copy of data that is ready but not serving query traffic. The data could be a potential master tablet.

Errors

- the <keyspace/shard> and <served tablet type> arguments are both required for the <SetShardServedTypes> command This error occurs if the command is not called with exactly 2 arguments.

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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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.

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.

ShardReplicationPositions Shows the replication status of each slave machine in the shard graph. In this case, the status refers to the replication lag between the master vttablet and the slave vttablet. In Vitess, data is always written to the master vttablet first and then replicated to all slave 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.

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
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.

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.

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.

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.

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.

Tablets

- Backup
- ChangeSlaveType
- DeleteTablet
- ExecuteFetchAsDba
- ExecuteHook

- GetTablet
- IgnoreHealthError
- InitTablet
- Ping
- RefreshState
- RefreshStateByShard
- ReparentTablet
- RestoreFromBackup
- RunHealthCheck
- SetReadOnly
- SetReadWrite
- Sleep
- StartSlave
- StopSlave
- UpdateTabletAddrs

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

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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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 <ChangeSlaveType> 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

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.

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.

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.

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.

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

Name	Type	Definition
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 slaved copy of data that is offline to queries other than for backup purposes
  - batch – A slaved 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 slaved 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 slaved copy of data for OLAP load patterns
  - replica – A slaved 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.
  - schema_apply – A slaved copy of data that had been serving query traffic but that is now applying a schema change. Following the change, the tablet will revert to its serving type.
  - snapshot_source – A slaved copy of data where mysqld is not running and where Vitess is serving data files to clone slaves. Use this command to enter this mode:
    Use this command to exit this mode:
  - spare – A slaved 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.

Ping Checks 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.

ReparentTablet Reparent a tablet to the current master in the shard. This only works if the current slave 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)

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.

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.

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

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.

StartSlave Starts replication on the specified slave.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- action <StartSlave> requires <tablet alias> This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

StopSlave Stops replication on the specified slave.

Example

Arguments

- <tablet alias> – Required. A Tablet Alias uniquely identifies a vttablet. The argument value is in the format <cell name>-<uid>.

Errors

- action <StopSlave> requires <tablet alias> This error occurs if the command is not called with exactly one argument.
- failed reading tablet %v: %v

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.

Topo

- TopoCat

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

Workflows

- WorkflowAction
- WorkflowCreate
- WorkflowDelete
- WorkflowStart
- WorkflowStop
- WorkflowTree
- WorkflowWait

WorkflowAction Sends the provided action name on the specified path.

Example

Arguments

- <name> – Required.

Errors

- the <path> and <name> arguments are required for the <WorkflowAction> command This error occurs if the command is not called with exactly 2 arguments.
- no workflow.Manager registered

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

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

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

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

WorkflowWait Waits for the workflow to finish.

Example

Errors

- the <uuid> argument is required for the <WorkflowWait> command This error occurs if the command is not called with exactly one argument.
- no workflow.Manager registered

VTExplain command line tool

Overview

This document provides information about the options and syntax of the `VTExplain` tool.

For a user guide that describes how to use the `VTExplain` tool to explain how Vitess executes a particular SQL statement, see Explaining how Vitess executes a SQL statement.

About VTExplain

The `VTExplain` tool provides information about how Vitess will execute a particular SQL statement. `VTExplain` is analagous to the MySQL `EXPLAIN` tool.

Syntax

```
> vtexplain {-vschema|vschema-file} {-schema|-schema-file} -sql
```

Options

The `vtexplain` command takes the following options:

-dbname string Optional database target to override normal routing (default ””)
-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 to simulate per keyspace (default 2).`vtexplain` will always allocate an evenly divided key range to each.
-replication-mode string The replication mode to simulate: either ROW or STATEMENT (default “ROW”).
-schema string The SQL table schema (default ””). Either `schema` or `schema-file` is required.
-schema-file string Identifies the file that contains the SQL table schema (default ””). Either `schema` or `schema-file` is required.
-sql string A list of semicolon-delimited SQL commands to analyze (default ””). Required.
-sql-file string Identifies the file that contains the SQL commands to analyze (default ””)
-vschema string Identifies the VTGate routing schema (default ””). Either `-vschema` or `-vschema-file` is required.
-vschema-file string Identifies the VTGate routing schema file (default ””)
-queryserver-config-passthrough-dmls query server pass through all dml statements without rewriting (default false)

To view a list of these options, execute the following command:

```
vtexplain --help
```

Examples

```
vtexplain -vschema-file vschema.json -schema-file schema.sql  -sql "SELECT * FROM users"
```

The example above explains how Vitess would execute the query `SELECT * FROM users` using the VSchema contained in `vschema.json` and the database schema contained in `schema.sql`.

```
vtexplain -shards 128 -vschema-file /tmp/vschema.json -schema-file /tmp/schema.sql
   -replication-mode "ROW" -output-mode text -sql "INSERT INTO users (user_id, name)
   VALUES(1, 'john')"
```

The example above explains how Vitess would execute the query `INSERT INTO users (user_id, name)VALUES(1, 'john')`, simulating 128 shards and row-based replication, and specifying text-based output.

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
```

See also

- Explaining how Vitess executes a SQL statement

VTTablet Modes

VTTablet can be configured to control MySQL at many levels. At the level with the most control, VTTablet can perform backups and restores, respond to reparenting commands coming through vtctld, automatically fix replication, and enforce semi-sync settings.

At the level with the least control, vttablet just sends the application's queries to MySQL. The level of desired control is achieved through various command line arguments, explained below.

Managed MySQL

In the mode with the highest control, VTTablet can take backups. It can also automatically restore from an existing backup to prime a new replica. For this mode, vttablet needs to run on the same host as MySQL, and must be given access to MySQL's `my.cnf` file. Additionally, the flags must not contain any connectivity flags like -db_host or -db_socket; VTTablet will fetch the socket information from `my.cnf` and use that to connect to the local MySQL.

It will also load other information from the `my.cnf`, like the location of data files, etc. When it receives a request to take a backup, it will shut down MySQL, copy the MySQL data files to the backup store, and restart MySQL.

The `my.cnf` file can be specified in the following ways:

- Implicit: If MySQL was initialized by the `mysqlctl` tool, then vttablet can find it based on just the `-tablet-path`. The default location for this file is `$VTDATAROOT/vt_<tablet-path>/my.cnf`.
- `-mycnf-file`: This option can be used if the file is not present in the default location.

- `-mycnf_server_id` and other flags: You can specify all values of the `my.cnf` file from the command line, and vttablet will behave as it read this information from a physical file.

Specifying a `-db_host` or a `-db_socket` flag will cause vttablet to skip the loading of the `my.cnf` file, and will disable its ability to perform backups or restores.

-restore_from_backup

The default value for this flag is false (i.e. the flag is not present). If set to true (i.e. the flag is present), 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 present, 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 (default is set to 4). This is typically useful in cloud environments to prevent the restore process from becoming a 'noisy' neighbor by consuming all available disk IOPS.

Unmanaged or remote MySQL

You can start a vttablet against a remote MySQL instance by simply specifying the connection flags `-db_host` and `-db_port` on the command line. In this mode, backup and restore operations will be disabled. If you start vttablet against a local MySQL, you can specify a `-db_socket` instead, which will still make vttablet treat the MySQL instance as if it was remote.

Specifically, the absence of a `my.cnf` file indicates to vttablet that it's connecting to a remote MySQL instance.

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 slave commands will not be allowed. These are InitShardMaster, 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 given to MySQL when it connects a slave 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 replication on new replicas as well as on any tablet that transitions to 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.

Typical vttablet command line flags

```
$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.

Example `TOPOLOGY_FLAGS` for a Topology Service like zookeeper:

```
-topo_implementation zk2 -topo_global_server_address localhost:21811,localhost:21812,localhost:21813
-topo_global_root /vitess/global
```

```
-enable_semi_sync
-enable_replication_reporter
-backup_storage_implementation file
-file_backup_storage_root $BACKUP_MOUNT
-restore_from_backup
-vtctld_addr http://$hostname:$vtctld_web_port/
```

```
-queryserver-config-pool-size 24
-queryserver-config-stream-pool-size 24
-queryserver-config-transaction-cap 300
```

More tuning flags are available, but the above overrides are definitely needed for serving reasonable production traffic.

```
-db_host $MYSQL_HOST
-db_port $MYSQL_PORT
-db_app_user $USER
-db_app_password $PASSWORD
```

```
-db_allprivs_user
-db_allprivs_password
-db_appdebug_user
-db_appdebug_password
-db_dba_user
-db_dba_password
-db_filtered_user
-db_filtered_password
```

Other flags exist for finer control.

Resources

description: Additional resources including Presentations and Roadmap

Presentations and Videos

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.

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.

{{< 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 >}}

Short Term

- Support for Point in Time Recovery
- Improve Documentation
- Improve Usability
- Support more MySQL Syntax (improve compatibility as a drop-in replacement)
- VReplication
  - Support "Dry Run"
- Componentize Tablet Server (lift restriction on one-tablet per MySQL schema)

Medium Term

- VReplication
  - Support for Schema Changes
  - Backfill lookup indexes
  - Support for Data Migration
- 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.

- Support for PostgreSQL: Vitess should be able to support PostgreSQL for both storing data, and speaking the protocol in VTGate.

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

Below are a few possible scenarios.

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

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

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. — ## 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 contents 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>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>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>xbstream_restore_flags</code></td>
  <td>For the <code>xtrabackup</code> backup engine, 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_stream_mode</code></td>
  <td>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>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>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</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 back-up 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

Run the following vtctl command to create a backup:

```
vtctl Backup <tablet-alias>
```

If the engine is `builtin`, in response to this command, the designated tablet performs the following sequence of actions:

1. Switches its type to `BACKUP`. After this step, the tablet is no longer used by vtgate to serve any query.
2. Stops replication, get the current replication position (to be saved in the backup along with the data).
3. Shuts down its mysqld process.
4. Copies the necessary files to the Backup Storage implementation that was specified when the tablet was started. Note if this fails, we still keep going, so the tablet is not left in an unstable state because of a storage failure.
5. Restarts mysqld.
6. Restarts replication (with the right semi-sync flags corresponding to its original type, if applicable).
7. Switches its type back to its original type. After this, it will most likely be behind on replication, and not used by vtgate for serving until it catches up.

If the engine is `xtrabackup`, we do not do any of the above. The tablet can continue to serve traffic while the backup is running.

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 Prerequisites 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>
  ```

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.

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

Configuring Components

Managed MySQL

The following describes the requirements for Vitess when fully managing MySQL with `mysqlctl` (see VTTablet Modes).

When using Unmanaged or Remote MySQL instead, the requirement is only that the server speak the MySQL protocol.

Version and Flavor `mysqlctl` supports MySQL/Percona Server 5.6 to 8.0, and MariaDB 10.0 to 10.3. MariaDB 10.4 is currently known to have installation issues (#5362).

Base Configuration Starting with Vitess 4.0, `mysqlctl` will auto-detect the version and flavor of MySQL you are using, and automatically-include a base configuration file in `config/mycnf/*`.

Auto-dection works by searching for `mysqld` in the `$PATH`, as well as in the environment variable `$VT_MYSQL_ROOT`. If auto-detection fails, `mysqlctl` will apply version detection based on the `$MYSQL_FLAVOR` environment variable. Auto-detection will always take precedence over `$MYSQL_FLAVOR`.

Specifying Additional Configuration The automatically-included base configuration makes only the required settings changes for Vitess to operate correctly. It is recommended to configure InnoDB settings such as `innodb_buffer_pool_size` and `innodb_log_file_size` according to your available system resources.

`mysqlctl` **will not** read configuration files from common locations such as `/etc/my.cnf` or `/etc/mysql/my.cnf`. To include a custom `my.cnf` file as part of the initialization of tablets, set the `$EXTRA_MY_CNF` environment variable to a list of colon-separated files. Each file must be an absolute path.

In Kubernetes, you can use a ConfigMap to overwrite the entire `$VTROOT/config/mycnf` directory with your custom versions, rather than baking them into a custom container image.

Unsupported Configuration Changes 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.

Setting	Reason
`sql-mode`	Vitess can operate with non-default SQL modes, but VTGate will not allow you to change the sql-mode on a per-session basis. This can create compatibility issues for applications that require changes to this setting.

init_db.sql When a new instance is initialized with mysqlctl init (as opposed to restarting in a previously initialized data dir with mysqlctl start), the `init_db.sql` file is applied to the server immediately after running the bootstrap procedure (either `mysqld --initialize-insecure` or `mysql_install_db`, depending on the MySQL version). This file is also responsible for removing unprivileged users, as well as adding the necessary tables and grants for Vitess.

Note that changes to this file will not be reflected in shards that have already been initialized and had at least one backup taken. New instances in such shards will automatically restore the latest backup upon vttablet startup, overwriting the data dir created by `mysqlctl`.

Vitess Servers

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 come in handy for troubleshooting. You can limit the disk usage by running cron jobs that periodically purge or archive them.

Vitess supports both MySQL protocol and gRPC for communication between client and Vitess and uses gRPC for communication between Vitess servers. By default, Vitess does not use SSL.

Also, even without using SSL, we allow the use of an application-provided CallerID object. It allows unsecure but easy to use authorization using Table ACLs.

See the Transport Security Model document for more information on how to setup both of these features, and what command line parameters exist.

Topology Service configuration Vttablet, vtgate and vtctld need the right command line parameters to find the topology service. First the `topo_implementation` flag needs to be set to one of zk2, etcd2, or consul. Then they're all configured as follows:

- The `topo_global_server_address` contains the server address / addresses of the global topology service.
- The `topo_global_root` contains the directory / path to use.

Note that the local cell for the tablet must exist and be configured properly in the Topology Service for vttablet to start. Local cells are configured inside the topology service, by using the `vtctl AddCellInfo` command. See the Topology Service documentation for more information.

VTTablet

VTTablet has a large number of command line options. Some important ones will be covered here. In terms of provisioning these are the recommended values

- 2-4 cores (in proportion to MySQL cores)
- 2-4 GB RAM

Directory Configuration vttablet supports a number of command line options and environment variables to facilitate its setup.

The VTDATAROOT environment variable specifies the toplevel directory for all data files. If not set, it defaults to /vt.

By default, a vttablet will use a subdirectory in VTDATAROOT named vt_NNNNNNNNNN where NNNNNNNNNN is the tablet id. The tablet_dir command-line parameter allows overriding this relative path. This is useful in containers where the filesystem only contains one vttablet, in order to have a fixed root directory.

When starting up and using mysqlctl to manage MySQL, the MySQL files will be in subdirectories of the tablet root. For instance, bin-logs for the binary logs, data for the data files, and relay-logs for the relay logs.

It is possible to host different parts of a MySQL server files on different partitions. For instance, the data file may reside in flash, while the bin logs and relay logs are on spindle. To achieve this, create a symlink from $VTDATAROOT/<dir name> to the proper location on disk. When MySQL is configured by mysqlctl, it will realize this directory exists, and use it for the files it would otherwise have put in the tablet directory. For instance, to host the binlogs in /mnt/bin-logs:

- Create a symlink from $VTDATAROOT/bin-logs to /mnt/bin-logs.
- When starting up a tablet: * /mnt/bin-logs/vt_NNNNNNNNNN will be created. * $VTDATAROOT/vt_NNNNNNNNNN/bin-logs will be a symlink to /mnt/bin-logs/vt_NNNNNNNNNN

Initialization

- Init_keyspace, init_shard, init_tablet_type: These parameters should be set at startup with the keyspace / shard / tablet type to start the tablet as. Note 'master' is not allowed here, instead use 'replica', as the tablet when starting will figure out if it is the master (this way, all replica tablets start with the same command line parameters, independently of which one is the master).

Query server parameters

- **queryserver-config-pool-size**: This value should typically 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.
- **queryserver-config-stream-pool-size**: This value is relevant only if you plan to run streaming queries against the database. It's recommended that you use rdonly instances for such streaming queries. This value depends on how many simultaneous streaming queries you plan to run. Typical values are in the low 100s.
- **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 QPS and transaction length. Typical values are in the low 100s.
- **queryserver-config-query-timeout**: This value should be set to the upper limit you're willing to allow a 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.

DB config parameters VTTablet requires multiple user credentials to perform its tasks. Since it's required to run on the same machine as MySQL, it's most beneficial to use the more efficient unix socket connections.

connection parameters

- `db_socket`: The unix socket to connect on. If this is specified, host and port will not be used.
- `db_host`: The host name for the tcp connection.
- `db_port`: The tcp port to be used with the `db_host`.
- `db_charset`: Character set. Only utf8 or latin1 based character sets are supported.
- `db_flags`: Flag values as defined by MySQL.

- `db_ssl_ca`, `db_ssl_ca_path`, `db_ssl_cert`, `db_ssl_key`: SSL flags.

app credentials are for serving app queries:

- `db_app_user`: App username.
- `db_app_password`: Password for the app username. If you need a more secure way of managing and supplying passwords, VTTablet does allow you to plug into a "password server" that can securely supply and refresh usernames and passwords. Please contact the Vitess team for help if you'd like to write such a custom plugin.
- `db_app_use_ssl`: Set this flag to false if you don't want to use SSL for this connection. This will allow you to turn off SSL for all users except for `repl`, which may have to be turned on for replication that goes over open networks.

appdebug credentials are for the appdebug user:

- `db_appdebug_user`
- `db_appdebug_password`
- `db_appdebug_use_ssl`

dba credentials will be used for housekeeping work like loading the schema or killing runaway queries:

- `db_dba_user`
- `db_dba_password`
- `db_dba_use_ssl`

repl credentials are for managing replication.

- `db_repl_user`
- `db_repl_password`
- `db_repl_use_ssl`

filtered credentials are for performing resharding:

- `db_filtered_user`
- `db_filtered_password`
- `db_filtered_use_ssl`

Monitoring VTTablet exports a wealth of real-time information about itself. This section will explain the essential ones:

/debug/status This page has a variety of human-readable information about the current VTTablet. You can look at this page to get a general overview of what's going on. It also has links to various other diagnostic URLs below.

/debug/vars This is the most important source of information for monitoring. There are other URLs below that can be used to further drill down.

Queries (as described in /debug/vars section) 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": {
      "PASS\_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, "`PASS\_SELECT`" is the only category. An entry like `"500000": 1133195` means that `1133195` queries took under `500000` nanoseconds to execute.

`Queries.Histograms.PASS\_SELECT.Count` is the total count in the `PASS\_SELECT` category.

`Queries.Histograms.PASS\_SELECT.Time` is the total time in the `PASS\_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 PASS_SELECT, but there will be more for masters.
- Per-category latency
- Per-category tail latency

```
  "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.

Mysql Mysql is a histogram variable like Queries, except that it reports MySQL execution times. The categories are "Exec" and "ExecStream".

In the past, the exec time difference between VTTablet and MySQL used to be substantial. With the newer versions of Go, the VTTablet exec time has been predominantly been equal to the mysql exec time, conn pool wait time and consolidations waits. In other words, this variable has not shown much value recently. However, it's good to track this variable initially, until it's determined that there are no other factors causing a big difference between MySQL performance and VTTablet performance.

Transactions Transactions is a histogram variable that tracks transactions. The categories are "Completed" and "Aborted".

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.

Plotting errors/query can sometimes be useful for troubleshooting.

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 be escalated for root cause analysis.

```
"Kills": {
  "Queries": 2,
  "Transactions": 0
},
```

Kills reports the queries and transactions killed by VTTablet due to timeout. It's a very important variable to look at during outages.

TransactionPool* There are a few variables with the above prefix:

```
"TransactionPoolAvailable": 300,
"TransactionPoolCapacity": 300,
"TransactionPoolIdleTimeout": 600000000000,
"TransactionPoolMaxCap": 300,
"TransactionPoolTimeout": 30000000000,
"TransactionPoolWaitCount": 0,
"TransactionPoolWaitTime": 0,
```

- `WaitCount` will give you how often the transaction pool gets full that 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 table acl stats 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.

`DataFree`, `DataLength`, `IndexLength`, `TableRows` These variables are updated periodically from information_schema.tables. They represent statistical information as reported by MySQL about each table. They can be used for planning purposes, or to track unusual changes in table stats.

- `DataFree` represents `data_free`
- `DataLength` represents `data_length`
- `IndexLength` represents `index_length`
- `TableRows` represents `table_rows`

/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 master.

/queryz, /debug/query__stats, /debug/query__plans, /streamqueryz

- /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.
- /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 at to see if a specific query is the root cause.
- /debug/query__plans is a more static view of the query cache. It just shows how VTTablet will process or rewrite the input query.
- /streamqueryz lists the currently running streaming queries. You have the option to kill any of them from this page.

/querylogz, /debug/querylog, /txlogz, /debug/txlog

- /debug/querylog is a never-ending stream of currently executing queries with verbose information about each query. 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/tabletserver/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.

/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

VTGate

A typical VTGate should be provisioned as follows.

- 2-4 cores
- 2-4 GB RAM

Since VTGate is stateless, you can scale it linearly by just adding more servers as needed. Beyond the recommended values, it's better to add more VTGates than giving more resources to existing servers, as recommended in the philosophy section.

Load-balancer in front of vtgate to scale up (not covered by Vitess). Stateless, can use the health URL for health check.

Parameters

- `cells_to_watch`: which cell vtgate is in and will monitor tablets from. Cross-cell master access needs multiple cells here.
- `tablet_types_to_wait`: VTGate waits for at least one serving tablet per tablet type specified here during startup, before listening to the serving port. So VTGate does not serve error. It should match the available tablet types VTGate connects to (master, replica, rdonly).
- `discovery_low_replication_lag`: when replication lags of all VTTablet in a particular shard and tablet type are less than or equal the flag (in seconds), VTGate does not filter them by replication lag and uses all to balance traffic.
- `degraded_threshold (30s)`: a tablet will publish itself as degraded if replication lag exceeds this threshold. This will cause VTGates to choose more up-to-date servers over this one. If all servers are degraded, VTGate resorts to serving from all of them.
- `unhealthy_threshold (2h)`: a tablet will publish itself as unhealthy if replication lag exceeds this threshold.
- `transaction_mode (multi)`: single: disallow multi-db transactions, multi: allow multi-db transactions with best effort commit, twopc: allow multi-db transactions with 2pc commit.
- `normalize_queries (false)`: Turning this flag on will cause vtgate to rewrite queries with bind vars. This is beneficial if the app doesn't itself send normalized queries.

Monitoring

/debug/status This is the landing page for a VTGate, which can gives you a status on 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.

/debug/query_plans

This URL gives you all the query plans for queries going through VTGate.

/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

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`. For `VARBINARY`, use `binary_md5`.

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
```

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.

Making Schema Changes

For applying schema changes for MySQL instances managed by Vitess, there are a few options.

ApplySchema

`ApplySchema` is a `vtctlclient` command that can be used to apply a schema to a keyspace. The main advantage of using this tool is that it performs some sanity checks about the schema before applying it. For example, if the schema change affects too many rows of a table, it will reject it.

However, the downside is that it is a little too strict, and may not work for all use cases.

VTGate

You can send a DDL statement directly to a VTGate just like you would send to a MySQL instance. If the target is a sharded keyspace, then the DDL would be scattered to all shards.

If a specific shard fails you can target it directly using the `keyspace/shard` syntax to retry the apply just to that shard.

If VTGate does not recognize a DDL syntax, the statement will get rejected.

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 and defaults to 1800 seconds). You can also explicitly issue the `vtctlclient ReloadSchema` command to make it reload immediately.

This approach can be extended to use schema deployment tools like `gh-ost` or `pt-online-schema-change`. Using these schema deployment tools is the recommended approach for large tables, because they incur no downtime.

MoveTables

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have either a Kubernetes (helm) or local installation ready. {{< /info >}}

MoveTables is a new VReplication workflow in Vitess 6, 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.

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 ecommerce system we know that `customer` and `corder` tables are closely related and growing at a high rate just by themselves.

Let's start by simulating this situation by loading sample data:

```
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.

Planning to Move Tables

In this scenario, we are going to split the `commerce` keyspace into `commerce` and `customer` keyspaces. The tables `Customer` and `COrder` will be moved into the newly created keyspace, and the `Product` table will remain in the `commerce` keyspace. This operation is 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.

Create new tablets

The first step in our MoveTables operation is to deploy new tablets for our `customer` keyspace. By convention, we are going to use the UIDs 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 `-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-6f955957bb-jp2t2                    1/1     Running     0          18m
pod/vtgate-zone1-86b7cb87d6-nsmw4              1/1     Running     3          18m
pod/zone1-commerce-0-init-shard-master-d5vj4   0/1     Completed   0          18m
```

```
pod/zone1-commerce-0-replica-0                   6/6     Running     0          18m
pod/zone1-commerce-0-replica-1                   6/6     Running     0          18m
pod/zone1-commerce-0-replica-2                   6/6     Running     0          18m
pod/zone1-customer-0-init-shard-master-xhzsr     0/1     Completed   0          89s
pod/zone1-customer-0-replica-0                   6/6     Running     0          89s
pod/zone1-customer-0-replica-1                   6/6     Running     0          89s
pod/zone1-customer-0-replica-2                   6/6     Running     0          89s

NAME                                             COMPLETIONS   DURATION   AGE
job.batch/zone1-commerce-0-init-shard-master     1/1           100s       18m
job.batch/zone1-customer-0-init-shard-master     1/1           17s        89s
```

```
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
```

Note: This change does not change the 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 -workflow=commerce2customer commerce customer '{"customer":{},
   "corder":{}}'
```

Phase 1: Switch 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 SwitchReads -tablet_type=rdonly customer.commerce2customer
vtctlclient SwitchReads -tablet_type=replica customer.commerce2customer
```

Phase 2: Switch Writes

After the reads have been *switched*, and you have verified that the system is operating as expected, it is time to *switch* the write operations. The command to execute the switch is very similar to switching reads:

```
vtctlclient SwitchWrites customer.commerce2customer
```

We can then verify that both reads and writes go to the new keyspace:

```
# Works
mysql --table < ../common/select_customer0_data.sql

# Expected to Fail!
mysql --table < ../common/select_commerce_data.sql
```

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 confusions. 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 SetShardTabletControl -blacklisted_tables=customer,corder -remove commerce/0
   rdonly
vtctlclient SetShardTabletControl -blacklisted_tables=customer,corder -remove commerce/0
   replica
vtctlclient SetShardTabletControl -blacklisted_tables=customer,corder -remove commerce/0
   master
vtctlclient ApplySchema -sql-file drop_commerce_tables.sql commerce
vtctlclient ApplyRoutingRules -rules='{}'
```

After this step is complete, you should see the following error:

```
# Expected to fail!
mysql --table < ../common/select_commerce_data.sql
```

This confirms that the data has been correctly cleaned up.

Next Steps

Congratulations! You've sucessfully moved tables between keyspaces. The next step to try out is to shard one of your keyspaces in Resharding.

Production Planning

Provisioning

Minimum Topology A highly available Vitess cluster requires the following components:

- 2 VTGate Servers
- A redundant Topology Service (e.g. 3 etcd servers)
- 3 MySQL Servers with semi-sync replication enabled
- 3 VTTablet processes
- A Vtctld process

It is common practice to locate the VTTablet process and MySQL Servers on the same host, and Vitess uses the terminology *tablet* to refer to both. The topology service in Vitess is pluggable, and you can use an existing etcd, ZooKeeper or Consul cluster to reduce the footprint required to deploy Vitess.

For development environments, it is possible to deploy with a lower number of these components. See `101_initial_cluster.sh` from the Run Vitess Locally guide for an example.

General Recommendations Vitess components (excluding the `mysqld` server) tend to be CPU-bound processes. They use disk space for storing their logs, but do not store any intermediate results on disk, and tend not to be disk IO bound. It is recommended to allocate 2-4 CPU cores for each VTGate server, and the same number of cores for VTTablet as with `mysqld`. If you are provisioning for a new workload, we recommend projecting that `mysqld` will require 1 core per 1500 QPS. Workloads with well optimized queries should be able to achieve greater than this.

The memory requirements for VTGate and VTTablet servers will depend on QPS and result set sizes, but a typical rule of thumb is to provision a baseline of 1GB per core.

The impact of network latency can be a factor when migrating from MySQL to Vitess. A simple rule of thumb is to estimate 2ms of round trip latency added to each query. Application code paths that make large numbers of database round-trips in a sequential code path will be most affected. To compensate, you may have to optimize or parallelize some code paths; or run additional threads or workers, which may result in additional memory requirements.

Planning Shard Size Vitess recommends provisioning shard sizes to approximately 250GB. This is not a hard-limit, and is driven primarily by the recovery time should an instance fail. With 250GB a full-recovery from backup is expected within less than 15 minutes. For most workloads this results in shards instances with relatively few CPU cores and lighter memory requirements, which tend to be more economical than running large instance sizes.

Running Multiple Tablets Per Server If you are using physical servers, Vitess encourages running multiple tablets (shards) per server. Typically the best way to do this is with Kubernetes, but `mysqlctl` also supports launching and managing multiple tablet servers if required.

Assuming tablets are kept to the recommended size of 250GB, they can start with a baseline CPU requirement of 2-4 cores for `mysqld` plus 2-4 cores for the VTTablet process, but this is obviously very workload-dependent.

Topology Service Provisioning By design, Vitess tries to contact the topology service as little as possible, and stores very little data in the topology server. For estimating CPU/memory/disk requirements, you can use the minimum requirements recommended by your preferred Topology Service.

Production testing

Before running Vitess in production, you should become comfortable with the different administrative operations. We recommend to go through the following scenarios on a non-production system.

Here is a short list of all the basic workflows Vitess supports:

- Reparenting
- Backup/Restore
- Schema Management
- Resharding / Horizontal Sharding Tutorial
- Upgrading

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 StartSlave` command.

Resharding

{{< info >}} This guide follows on from the Get Started guides. Please make sure that you have either a Kubernetes (helm) or local installation ready. {{< /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`.
- For `VARBINARY`, use `binary_md5`.
- 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 temporaily 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-file create_commerce_seq.sql commerce
vtctlclient ApplyVSchema -vschema_file vschema_commerce_seq.json commerce
vtctlclient ApplySchema -sql-file create_customer_sharded.sql customer
vtctlclient ApplyVSchema -vschema_file vschema_customer_sharded.json 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
```

```
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
```

Sanity Check

Applying the above change should result in the creation of six more vttablet instances; one master, one replica and one rdonly tablet for each of the two shards.

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
```

Start the Reshard

This process starts the reshard opration. It occurs online, and will not block any read or write operations to your database:

```
vtctlclient Reshard customer.cust2cust '0' '-80,80-'
```

Switch Reads

Once the reshard is complete, the first 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 SwitchReads -tablet_type=rdonly customer.cust2cust
vtctlclient SwitchReads -tablet_type=replica customer.cust2cust
```

Switch Writes

After 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 SwitchWrites customer.cust2cust
```

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 |
+----------+-------------+----------+-------+
```

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
```

```
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. But the shard metadata is still present. After Vitess brings down all vttablets, we can clean that up with this command:

```
vtctlclient DeleteShard -recursive customer/0
```

Beyond this, you will also need to manually delete the disk associated with this shard.— ## Upgrading Vitess

This document highlights things to look after 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 in YouTube we follow 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

User and Permission Management

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 you 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:

```
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'
```

Authorization

Authorization in Vitess is enforced on a table-level by the underlying VTTablets, and not by VTGate, as with authentication. As an example, say we have two services that want to run in their own keyspace and do not want any other service to have access to their keyspace.

Building on the authentication setup above and assuming your Vitess cluster already has 2 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 fine:

```
$ mysql -h 127.0.0.1 -u myuser1 -ppassword1 -D keyspace1 -e "select * from t"
$
```

Note the use above of the following parameters: * `-enforce-tableacl-config`: Fail to start VTTablet if there are no ACLs successfully configured. This ensures ACL misconfigurations are caught at startup. * `-queryserver-config-strict-table-acl`: only allow queries that pass table acl checks to ensure we do not allow other users. You will typically need to pass this parameter for the ACLs to be successfully enforced, unless you strictly limit the universe of potential users at the VTGate authentication level.

The following option may be useful: * `-queryserver-config-enable-table-acl-dry-run`: Only emit metrics when an ACL denies a request, but let the actual request pass through successfully. This allows you to test and verify ACL changes without running the risk of breakage.

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. {{< /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
```

Explaining how Vitess executes a SQL statement

Introduction

This document explains how to learn more about 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 explain how Vitess executes a SQL 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

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 will use. 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 `11-24` 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.

See also

- For detailed configuration options for VTExplain, see the VTExplain syntax reference.