VITESS

Servers and Tools for MariaDB/MySQL Scaling

Editor:
T.J. Yang YANG

October 4, 2014

Summary Table of Contents

Summary Table of Contents 1

Detail Table of Contents i

1 Overview 2
- 1.1 Vitess . . . 2

2 Vitess Introduction 5
- 2.1 Helicopter overview . . . 6
- 2.2 Motivation and Vision . . . 9
- 2.3 Vitess Frequently Asked Questions . . . 11
- 2.4 Contributing to Vitess . . . 12

3 Using Vitess 14
- 3.1 Getting Started . . . 15
- 3.2 Tools and servers . . . 17
- 3.3 vttablet . . . 19
- 3.4 Reparenting . . . 20
- 3.5 Resharding . . . 22
- 3.6 Production setup . . . 24
- 3.7 Schema Management . . . 25

4 Reference 27
- 4.1 Concepts . . . 28
- 4.2 Zookeeper Data . . . 30
- 4.3 Serving Graph . . . 34
- 4.4 Replication Graph . . . 35
- 4.5 Testing On A Ramdisk . . . 36
- 4.6 Setup . . . 36
- 4.7 Teardown . . . 36

A Making of this book 37
- A.1 Where to download the book source . . . 37
- A.2 Book Revision History . . . 37
- A.3 Xymon Revision History . . . 37

B Open Publication License 39
- B.1 Requirements on both unmodified and modified versions . . . 39
- B.2 Copyright . . . 39
- B.3 Scope of license . . . 39
- B.4 Requirements on modified works . . . 40

B.5 Good-practice recommendations 40
B.6 License options 40

Bibliography **41**

Index **41**

Contents

Summary Table of Contents 1

Detail Table of Contents i

1 Overview 2
- 1.1 Vitess . . . 2
  - 1.1.1 Overview . . . 2
  - 1.1.2 Trying it out . . . 3
  - 1.1.3 Documentation . . . 3
    - Intro . . . 3
    - Using Vitess . . . 3
    - Reference . . . 4
  - 1.1.4 License . . . 4

2 Vitess Introduction 5
- 2.1 Helicopter overview . . . 6
  - 2.1.1 When do you need Vitess? . . . 6
  - 2.1.2 How Vitess can help? . . . 6
    - Connection pooling . . . 6
    - Handling queries . . . 6
    - Sharding . . . 7
    - Workflow management . . . 7
    - Limiting complexity . . . 7
  - 2.1.3 Starting easy: vtocc . . . 7
  - 2.1.4 Going all the way: vttablet . . . 7
    - Topology backend . . . 7
    - Preparing the data . . . 7
    - vttablets . . . 8
    - Clients . . . 8
- 2.2 Motivation and Vision . . . 9
  - Priorities . . . 9
  - Trade-offs . . . 9
  - Preserved MySQL features . . . 9
  - The Vitess spectrum . . . 9
- 2.3 Vitess Frequently Asked Questions . . . 11
  - 2.3.1 Is Vitess used at Google? . . . 11
  - 2.3.2 Why not do it inside MySQL? . . . 11
  - 2.3.3 Why is it written in Go? . . . 11
  - 2.3.4 Why not use X (where X is usually a NoSQL database) instead of putting effort into scaling MySQL? . . . 11

2.3.5 Whats up with the name, Vitess? . . . 11
2.3.6 Why are you using ZooKeeper instead of Etcd/Doozer/Consul/etc? . . . 11
2.3.7 Is the Vitess used at Google the same as the Open Source version? . . . 11
2.4 Contributing to Vitess . . . 12
Prerequisites . . . 12
2.4.1 Small changes . . . 12
2.4.2 Bigger changes . . . 12
Recommended Git flow: single contributor . . . 12
Recommended Git flow: multiple contributors . . . 12
Changes and code reviews . . . 13
2.4.3 Other Git setups . . . 13

3 Using Vitess **14**
3.1 Getting Started . . . 15
3.1.1 Dependencies . . . 15
3.1.2 Building . . . 15
3.1.3 Testing . . . 15
Common Test Issues . . . 15
Node already exists, port in use, etc. . . . 15
Too many connections to MySQL, or other timeouts . . . 16
Connection refused to tablet, MySQL socket not found, etc. . . . 16
Connection refused in zkctl test . . . 16
Running out of disk space . . . 16
3.1.4 Setting up a cluster . . . 16
3.1.5 TODO . . . 16
3.2 Tools and servers . . . 17
vtctl . . . 17
vttablet . . . 17
vtocc . . . 17
vtgate . . . 17
vtctld . . . 17
vtworker . . . 17
vtprimecache . . . 18
Other support tools . . . 18
3.2.1 Vitess components block diagram . . . 18
3.3 vttablet . . . 19
3.4 Reparenting . . . 20
3.4.1 Active Reparents . . . 20
3.4.2 External Reparents . . . 20
3.4.3 Reparenting And Serving Graph . . . 20
3.5 Resharding . . . 22
3.5.1 Process . . . 22
3.5.2 Applications . . . 22
3.5.3 Scaling Up and Down . . . 23
3.5.4 Filtered Replication . . . 23
3.6 Production setup . . . 24
3.6.1 Setting up Zookeeper . . . 24
3.6.2 Launch vttablets . . . 24
3.6.3 Launch vtgate(s) . . . 24
3.7 Schema Management . . . 25
3.7.1 Looking at the Schema . . . 25
3.7.2 Changing the Schema . . . 25

Use case 1: Single tablet update: . . . 25
Use case 2: Single Shard update: . . . 25
Use case 3: Keyspace update: . . . 26

4 Reference **27**
4.1 Concepts . . . 28
Shard . . . 28
Tablet . . . 28
Keyspace id . . . 28
Shard graph . . . 28
Replication graph . . . 29
Serving graph . . . 29
Topology Server . . . 29
Cell (Data Center) . . . 29
4.2 Zookeeper Data . . . 30
4.2.1 Keyspace / Shard / Tablet Data . . . 30
Keyspace . . . 30
Shard . . . 30
Tablet . . . 31
4.2.2 Replication Graph . . . 32
4.2.3 Serving Graph . . . 32
SrvKeyspace . . . 32
SrvShard . . . 33
EndPoints . . . 33
4.3 Serving Graph . . . 34
4.3.1 SrvKeyspace . . . 34
4.3.2 SrvShard . . . 34
4.3.3 EndPoints . . . 34
4.3.4 Rebuilding the Serving Graph . . . 34
4.4 Replication Graph . . . 35
4.4.1 Master . . . 35
4.4.2 Slaves . . . 35
4.4.3 Discovery . . . 35
4.4.4 Reparenting . . . 35
4.5 Testing On A Ramdisk . . . 36
4.6 Setup . . . 36
4.7 Teardown . . . 36

A Making of this book **37**
A.1 Where to download the book source . . . 37
A.2 Book Revision History . . . 37
A.3 Xymon Revision History . . . 37

B Open Publication License **39**
B.1 Requirements on both unmodified and modified versions . . . 39
B.2 Copyright . . . 39
B.3 Scope of license . . . 39
B.4 Requirements on modified works . . . 40
B.5 Good-practice recommendations . . . 40
B.6 License options . . . 40

Bibliography **41**

Index **41**

List of Figures

1.1 Vitess Overview 2

2.1 Vitess Spectrum 10

3.1 components block diagram 18

List of Tables

A.1 Book Revision History. 38
A.2 Xymon Revision History. 38

Chapter 1

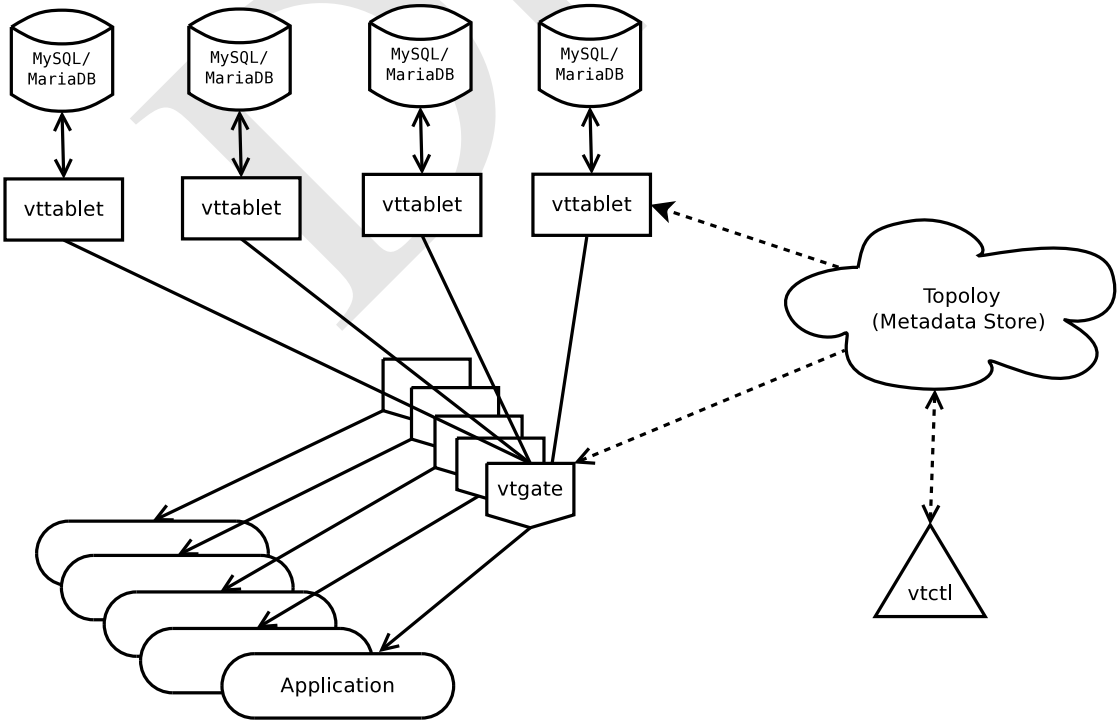
Overview

1.1 Vitess

Vitess is a set of servers and tools meant to facilitate scaling of MySQL databases for the web. It's been developed since 2011, and is currently used as a fundamental component of YouTube's MySQL infrastructure, serving thousands of QPS (per server). If you want to find out whether Vitess is a good fit for your project, please read our helicopter overview.

1.1.1 Overview

Figure 1.1: Vitess Overview



Vitess consists of a number servers, command line utilities, and a consistent metadata store. Taken together, they allow you to serve more database traffic, and add features like sharding, which normally you would have to implement in your application.

vttablet is a server that sits in front of a MySQL database, making it more robust and available in the face of high traffic. Among other things, it adds a connection pool, has a row based cache, and it rewrites SQL queries to be safer and nicer to the underlying database.

vtgate is a very light proxy that routes database traffic from your app to the right vttablet, basing on the sharding scheme, latency required, and health of the vttablets. This allows the client to be very simple, as all it needs to be concerned about is finding the closest vtgate.

The **topology** is a metadata store that contains information about running servers, the sharding scheme, and replication graph. It is backed by a consistent data store, like Apache ZooKeeper. The topology backends are plugin based, allowing you to write your own if ZooKeeper doesn't fit your needs. You can explore the topology through **vtctld**, a webserver (not shown in the diagram).

vtctl is a command line utility that allows a human or a script to easily interact with the system.

All components communicate using a lightweight RPC system based on BSON. The RPC system is plugin based, so you can easily write your own backend (at Google we use a Protocol Buffers based protocol). We provide a client implementation for three languages: Python, Go, and Java. Writing a client for your language should not be difficult, as it's a matter of implementing only a few API calls (please send us a pull request if you do!).

To learn more, please click on the documentation links below. You can also watch a 9 minute introduction to Vitess sougou prepared for Google I/O 2014. There's also a longer presentation from the Fosdem '14 go devroom (slides, video).

1.1.2 Trying it out

Vitess is not entirely ready for unsupervised use yet. Some functionality is still under development, APIs may change, and parts of the code are undocumented. However, if you feel adventurous, you're more than welcome to try it. We know that there are some rough edges, so please don't hesitate to reach out to us through our mailing list if you run into any issues. Warnings aside, please take a look at our Getting Started guide.

1.1.3 Documentation

Intro

- Helicopter overview: high level overview of Vitess that should tell you whether Vitess is for you.
- Frequently Asked Questions.
- Vision: principles guiding the design of Vitess.

Using Vitess

- Getting Started: how to set your environment to work with Vitess.
- Tools: all Vitess tools and servers.
- Vttablet: information about the most important Vitess server.
- Reparenting: performing master failover.
- Resharding: adding more shards to your cluster.
- Preparing for production (wip).
- Schema management: managing your database schema using Vitess.

Reference

- General Concepts
- Zookeeper data
- Serving graph
- Replication Graph

1.1.4 License

Unless otherwise noted, the vitess source files are distributed under the BSD-style license found in the LICENSE file.

Chapter 2

Vitess Introduction

2.1 Helicopter overview

This is a very high level overview of Vitess. It doesnt get into the technical details its goal is to help you understand if Vitess can solve your problems, and what kind of effort it would require for you to start using Vitess.

2.1.1 When do you need Vitess?

- You store all your data in a MySQL database, and have a significant number of clients. At some point, you start getting Too many connections errors from MySQL, so you have to change the max_connections system variable. Every MySQL connection has a memory overhead, which is just below 3 MB in the default configuration. If you want 1500 additional connections, you will need over 4 GB of additional RAM and this is not going to be contributing to faster queries.
- From time to time, your developers make mistakes. For example, they make your app issue a query without setting a LIMIT, which makes the database slow for all users. Or maybe they issue updates that break statement based replication. Whenever you see such a query, you react, but it usually takes some time and effort to get the story straight.
- You store your data in a MySQL database, and your database has grown uncomfortably big. You are planning to do some horizontal sharding. MySQL doesnt support sharding, so you will have write the code to perform the sharding, and then bake all the sharding logic into your app.
- You run a MySQL cluster, and use replication for availability: you have a master database and a few replicas, and in the case of a master failure some replica should become the new master. You have to manage the lifecycle of the databases, and communicate the current state of the system to the application.
- You run a MySQL cluster, and have custom database configurations for different workloads. Theres the master where all the writes go, fast read-only replicas for web clients, slower read-only replicas for batch jobs, and another kind of slower replicas for backups. If you have horizontal sharding, this setup is repeated for every shard. The code that your app uses to find the right database is rather complicated, and you have to deal with constantly updating the configuration.

If your MySQL installation is similar to any of the three scenarios described above, you may benefit from taking a closer look at Vitess. YouTube suffered from an extreme case of all of these problems we serve a lot of traffic and Vitess helped us alleviate them.

2.1.2 How Vitess can help?

Connection pooling

Instead of using MySQLs protocol, clients connect to Vitess uses its almost stateless, BSON based protocol. These connections are very lightweight (around 32 KB per connections), which means our servers can handle thousands of them without breaking a sweat. These connections are efficiently mapped to a pool of MySQL connections thanks to Gos awesome concurrency support.

Handling queries

When Vitess gets a query, it goes through a SQL parser, which informs the decision how to proceed with the query. If the query looks like it could have an avoidable negative impact on the performance, it will be rewritten according to a configurable set of rules. For example, queries without a limit will get a default limit of 10 thousand rows. If at the time a client issues a query the exact same query is already in flight, the database will do the work only once, returning the same result to both clients.

Sharding

Vitess has sharding built in, and it facilitates sharding with minimal downtime. For example, it supports split replication, in which the replication stream can be divided in such a way that a future shard master will get only statements that could possibly affect rows in its new shard. This allows us to perform a resharding with only a few minutes of read-only downtime for the shard.

What is more, if you already have your own, custom sharding scheme in place, Vitess is fine with that: you can easily keep using it for its other features.

Workflow management

Vitess can help you manage the lifecycle of your database instances. It supports various scenarios, like master failover, backups, etc. and all of them are handled automatically, minimizing any necessary downtime.

Limiting complexity

All the metadata about the Vitess cluster (sharding scheme, running instances, their health, work profile, etc.) is stored in the Topology, which is backed by a consistent data store, like ZooKeeper. This means that the apps cluster view is always up to date and consistent for different clients. What is more, your app doesnt need to know anything about all this. It connects to Vitess servers through vtgate, a lightweight proxy which handles routing the queries to the appropriate instances. This allows the Vitess client to be extremely simple its just an implementation of a few API calls.

2.1.3 Starting easy: vtocc

If you are only interested in connection pooling and handling queries, you are in luck: it should be simple to start using those features without making significant changes to your infrastructure. All you need is to run vtocc in front of your MySQL database, and change your app to use the Vitess client instead of your MySQL driver. This is how we launched Vitess into production at YouTube

2.1.4 Going all the way: vttablet

Getting a fully functional Vitess installation is a quite involved affair, and it requires some planning.

Topology backend

Vitess needs to store its metadata in some place. This data storage needs to have different properties than a regular database. It doesnt need to support a lot of traffic or data, but the data should be consistent and be able to survive failures in the machines that host it (so, we are interested in Consistency and Partition Tolerance from the CAP theorem). This means that the ideal candidates for this role implement Paxos or something equivalent.

Out of the box, Vitess supports Apache ZooKeeper, which is a well battle tested distributed lock service. If you have a reason not to use ZooKeeper, it should be relatively easy to write a Vitess plugin to use something like Etcd, Consul, or Doozer. At Google, for example, we use Chubby through the plugin mechanism.

Preparing the data

Vitess has the notion of a keyspace a logical database that may consist of many shards. If you handle sharding yourself or dont need it, this is just a matter of choosing a name (your database name is a reasonable choice). If you want Vitess to handle sharding, or think you may need that in the future, theres a bit more work that you should do.

First, you have to think about your sharding scheme. This is quite a big topic by itself, but the general idea is that data that is used together should be kept on the same shard. For example, if your app deals with users and objects created by those users, the user id is a good candidate to build your sharding schema around. The choices you make

about your sharding scheme are crucial. One of the limitations of Vitess (and any other sharding solution for MySQL) is that transactions will not work across shard boundaries.

Once you know what you want to base your sharding key on, you have to add this data to the database. You should create an additional column in all your sharded tables thats big enough to contain a 64 bit integer and populate it.

vttablets

Now that the data is ready for Vitess, you can start launching vttablets. There should be exactly one vttablet for every MySQL database, and you can tag different vttablets for different kind of jobs. After you start an instance that connects to its database, you let the system know about it by using vtctl, a command line tool.

Clients

Your app wont connect to the tablets directly it will always go through vtgate. There are many ways to setup vtgate. You can have a pool of them, or you can start one per process or group of processes (it doesnt consume much resources). All your app needs to do is to use the Vitess client and somehow let it know how to connect to a running vtgate instance.

2.2 Motivation and Vision

MySQL is an easy relational database to get started with. It's easy to setup and has a short learning curve. However, as your system starts to scale, it begins to run out of steam. This is mainly because it's non-trivial to shard a MySQL database after the fact. Among other problems, the growing number of connections also becomes an unbearable overhead.

On the other end of the spectrum, there are NoSQL databases. However, they suffer from problems that mainly stem from the fact that they're new. Those who have adopted them have struggled with the lack of secondary indexes, table joins and transactions.

Vitess tries to bring the best of both worlds by trading off some of MySQL's consistency features in order to achieve the kind of scalability that NoSQL databases provide.

Priorities

- *Scalability*: This is achieved by replication and sharding.
- *Efficiency*: This is achieved by a proxy server (vttablet) that mediates all queries and connections. It also utilizes a more efficient rowcache to short-cut some of the queries. This effectively increases a typical MySQL's serving capacity.
- *Manageability*: As soon as you add replication and sharding that span across multiple data centers, the number of servers spirals out of control. Vitess provides a set of tools backed by a lockserver (zookeeper) to track and administer them.
- *Simplicity*: As the complexity grows, it's important to hide this from the application. The vtgate servers give you a unified view of the fleet that makes it feel like you're just interacting with one database.

Trade-offs

Scalability and availability require some trade-offs: * *Consistency*: In a typical web application, not all reads have to be fully consistent. Vitess lets you specify the kind of consistency you want on your read. It's generally recommended that you use replica reads as they're easier to scale. You can always request for master reads if you want up-to-date data. You can also additionally perform 'for update' reads that ensure that a row will not change until you've committed your changes. * *Transactions*: Relational transactions are prohibitively expensive across distributed systems. Vitess eases this constraint and guarantees transactional integrity 'per keyspace id', which is restricted to one shard. Heuristically, this tends to cover most of an application's transactions. For the few cases that don't, you can sequence your changes in such a way that the system looks consistent even if a distributed transaction fails in the middle. * *Latency*: There is some negligible latency introduced by the proxy servers. However, they make up for the fact that you can extract more throughput from MySQL than you would otherwise be able to without them.

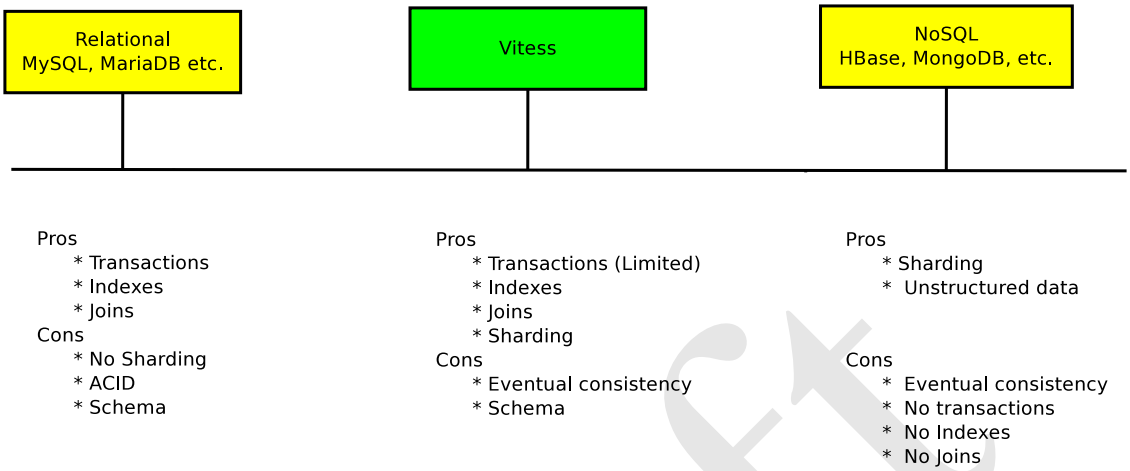
Preserved MySQL features

Since the underlying storage layer is still MySQL, we still get to preserve its other important features: * *Indexes*: You can create secondary indexes on your tables. This allows you to efficiently query rows using more than one key. * *Joins*: MySQL allows you to split one-to-many and many-to-many relational data into separate tables, and lets you join them on demand. This flexibility generally results in more efficient storage as each piece of data is stored only once, and fetched only if needed.

The Vitess spectrum

The following diagram illustrates where vitess fits in the spectrum of storage solutions:

Figure 2.1: Vitess Spectrum



2.3 Vitess Frequently Asked Questions

2.3.1 Is Vitess used at Google?

Yes, Vitess has been used to serve all YouTube database traffic since 2011.

2.3.2 Why not do it inside MySQL?

MySQL is really good at what it does: serving queries with a low latency, on a single server. However, it was not designed to do a lot of the things at the scale that we need at YouTube. Including the changes we needed inside MySQL itself would have been a huge project. Additionally, its a large C++ codebase thats almost 20 years old, which means that making changes is more complicated than we would like. We decided to correct or enforce the behaviors we needed from the outside.

2.3.3 Why is it written in Go?

Go is a language that hits the sweet spot in terms of expressiveness and performance: its almost as expressive as Python, very maintainable, but its performance is in the same range as Java (close to C++ in certain cases). What is more, the language is extremely well suited for concurrent programming, and it has a very high quality standard library.

2.3.4 Why not use X (where X is usually a NoSQL database) instead of putting effort into scaling MySQL?

Every database has its performance profile, meaning that some types of queries usually run faster than other. Because of that, migrating from one datastore to another involves a lot more work than just copying the data and writing equivalent queries. If you need high performance, you need to rethink how your app accesses its data. MySQLs low latency is very hard to match. Moreover, even if it has some quirks, those quirks are usually pretty well understood.

2.3.5 Whats up with the name, Vitess?

Vitess was originally named `Voltron`, but we decided to change it before making it Open Source to avoid infringing any trademarks. Still, we wanted a name that would allow us to keep using the prefix VT in our code. *Vitesse* means speed in French and we dropped the final E to make it easier to find using Google (also, it looks much cooler without the E!).

2.3.6 Why are you using ZooKeeper instead of Etcd/Doozer/Consul/etc?

The topology is a crucial part of our infrastructure, so we didnt want to take any risks with it. ZooKeeper may have some rough edges and not be very easy to set up, but its battle tested, so we know we can rely on it. Of course, you are free to write a plugin to use your lock service of choice we would appreciate if you contributed the code back.

2.3.7 Is the Vitess used at Google the same as the Open Source version?

Mostly. The core functionality remains unchanged, but we add some code to take advantage of Googles infrastructure (so we can use [Bur] Chubby, the Protocol Buffers based RPC system, etc.). Our philosophy is Open Source first when we develop a new feature, we first make it work in the Open Source tree, and only then write a plugin that makes use of Google specific technologies. We believe this is very important to keep us honest, and to ensure that the Open Source version of Vitess is as high quality as the internal one. The vast majority of our development takes place in the open, on GitHub. This also means that Vitess is built with extensibility in mind it should be pretty straightforward to adjust it to the needs of your infrastructure.

2.4 Contributing to Vitess

If you'd like to make simple contributions to Vitess, we recommend that you fork the repository and submit pull requests. If you'd like to make larger or ongoing changes, you'll need to follow a similar set of processes and rules that the Vitess team follows.

Prerequisites

- Install vitess
- The vitess team uses appspot for code reviews. You'll need to create an account at http://codereview.appspot.com.
- Fork the vitess repository, say https://github.com/myfork/vitess.
- Download upload.py and put it in your path.

2.4.1 Small changes

For small, well contained changes, just send us a pull request.

2.4.2 Bigger changes

If you are planning to make bigger changes or add serious features to Vitess, we ask you to follow our code review process.

Recommended Git flow: single contributor

Use your fork as a push-only staging ground for submitting pull requests. The assumption is that you'll never have to fetch from the fork. If this is the case, all you have to do is configure your local repository to pull from youtube, and push to myfork. This can be achieved as follows:

```
   ~/...vitess> git remote -v origin git@github.com:youtube/vitess.git (fetch) origin git@github.com:youtu
(push) ~/...vitess> git remote set-url --push origin git@github.com:myfork/vitess ~/...vitess>
git remote -v origin git@github.com:youtube/vitess.git (fetch) origin git@github.com:myfork/vitess
(push)
```

The limitation of this configuration is that you can only pull from the youtube repository. The `git pull` command will `fetch` from youtube/vitess and `merge` into your master branch.

On the other hand, `git push` will push into your myfork/vitess remote.

The advantage of this workflow is that you don't have to worry about specifying where you're pulling from or pushing to because the default settings *do the right thing*.

Recommended Git flow: multiple contributors

If more than one of you plan on contributing through a single fork, then you'll need to follow a more elaborate scheme of setting up multiple remotes and manually managing merges:

```
   ~/...vitess> git remote -v origin git@github.com:youtube/vitess.git (fetch) origin git@github.com:youtu
(push) ~/...vitess> git remote add myfork git@github.com:myfork/vitess.git ~/...vitess> git
remote -v myfork git@github.com:myfork/vitess.git (fetch) myfork git@github.com:myfork/vitess.git
(push) origin git@github.com:youtube/vitess.git (fetch) origin git@github.com:youtube/vitess.git
(push)
```

With this setup, commands like `git pull` and `git push` with default settings are not recommended. You will be better off using `git fetch` and `git merge`, which let you micromanage your remote interactions. For example, you'll need to `git push myfork` to explicitly push your changes to myfork.

Changes and code reviews

We recommend that you make your changes in a separate branch. Make sure you're on the master branch when you create it.

```
~/...vitess> git status # On branch master # Your branch is up-to-date with 'origin/master'.
# nothing to commit, working directory clean ~/...vitess> git checkout -b newfeature Switched
to a new branch 'newfeature'
```

Once your changes are ready for review and committed into your branch, you can run the createcl tool, for example: `createcl -r alainjobart` This command will automatically run a diff of the current branch `newfeature` against `master` and create an appspot code review with `alainjobart` as reviewer. vitess-issues will be cc'd. If necessary, createcl allows you to specify the exact versions to diff. (but we recommend that you don't use those).

After getting feedback about your code, you can update the code by calling

```
createcl -i 12345
```

with the actual id of your change instead of 12345.

During your feature development, you can fetch and merge new changes from the main youtube repository. If you choose to do so, make sure you merge the changes to both the `master` and `newfeature` branches. In the sole contributor case, your commands will look like this: `git checkout master git pull git checkout newfeature git merge master` Once your change is approved, push and submit it as a pull request: `git checkout master git merge newfeature git push` The above commands will merge `newfeature` into `master` and push the changes to the myfork remote. You can then go to https://github.com/myfork/vitess to submit the branch as your pull request. If done correctly, only your changes will show up in the pull request. github will cancel out changes you merged from youtube master, unless you resolved merge conflicts.

If necessary, you can work on multiple branches at the same time. When the time comes to submit, you just have to merge the branch onto `master` and push.

2.4.3 Other Git setups

As you can see above, the only requirement from the Vitess team is that you send your code reviews through appspot, and then submit the same changes as a pull request.

Our workflow recommendation is mainly to simplify your life. If you prefer to use a different workflow, you can choose to do so as long as you can figure out a way to meet the necessary requirements.

Chapter 3

Using Vitess

3.1 Getting Started

If you run into issues or have questions, you can use our mailing list: vitess@googlegroups.com.

3.1.1 Dependencies

- We currently develop on Ubuntu 12.04 and 14.04.
- You'll need some kind of Java Runtime (for ZooKeeper). We use OpenJDK (*sudo apt-get install openjdk-7-jre*).
- Go 1.2+: Needed for building Vitess.
- MariaDB: We currently develop with version 10.0.13. Other 10.0.x versions may also work.
- ZooKeeper: By default, Vitess uses Zookeeper as the lock service. It is possible to plug in something else as long as the new service supports the necessary API functions.
- Memcached: Used for the rowcache.
- Python: For the client and testing.

3.1.2 Building

Install Go.

Install MariaDB. You can use any installation method (src/bin/rpm/deb), but be sure to include the client development headers (**libmariadbclient-dev**).

Then download and build Vitess. Note that the value of MYSQL_FLAVOR is case-sensitive. If the mysql_config command from libmariadbclient-dev is not on the PATH, you'll need to *export VT_MYSQL_ROOT=/path/to/mariadb* before running bootstrap.sh, where mysql_config is found at /path/to/mariadb/**bin**/mysql_config.

```
sh cd $WORKSPACE sudo apt-get install make automake libtool memcached python-dev python-mysqldb
libssl-dev g++ mercurial git pkg-config bison curl git clone https://github.com/youtube/vitess.git
src/github.com/youtube/vitess cd src/github.com/youtube/vitess export MYSQL_FLAVOR=MariaDB ./bootstrap.sh
.  ./dev.env make build
```

3.1.3 Testing

The full set of tests included in the default *make* and *make test* targets is intended for use by Vitess developers to verify code changes. These tests simulate a small cluster by launching many servers on the local machine, so they require a lot of resources (minimum 8GB RAM and SSD recommended).

If you are only interested in checking that Vitess is working in your environment, you can run a set of lighter tests:

```
sh make site_test
```

Common Test Issues

Many common failures come from running the full developer test suite (*make* or *make test*) on an underpowered machine. If you still get these errors with the lighter set of site tests (*make site_test*), please let us know on the mailing list.

Node already exists, port in use, etc. Sometimes a failed test may leave behind orphaned processes. If you use the default settings, you can find these by looking for *vtdataroot* in the command line, since every process is told to put its files there with a command line flag. For example:

```
sh pgrep -f -l '(vtdataroot|VTDATAROOT)' # list Vitess processes pkill -f '(vtdataroot|VTDATAROOT)'
# kill Vitess processes
```

Too many connections to MySQL, or other timeouts This often means your disk is too slow. If you don't have access to an SSD, you can try testing against a ramdisk.

Connection refused to tablet, MySQL socket not found, etc. This could mean you ran out of RAM and a server crashed when it tried to allocate more. Some of the heavier tests currently require up to 8GB RAM.

Connection refused in zkctl test This could indicate that no Java Runtime is installed.

Running out of disk space Some of the larger tests use up to 4GB of temporary space on disk.

3.1.4 Setting up a cluster

3.1.5 TODO

TODO: Expand on all sections

- Setup zookeeper
- Start a MySql instance
- Start vttablet
- Start vtgate
- Write a client
- Test

3.2 Tools and servers

The vitess tools and servers are designed to help you even if you start small, and scale all the way to a complete fleet of databases.

In the early stages, connection pooling, rowcache and other efficiency features of vttablet help you get more from your existing hardware. As things scale out, the automation tools start to become handy.

vtctl

vtctl is the main tool for performing administrative operations. It can be used to track shards, replication graphs and db categories. It's also used to initiate failovers, reshard, etc.

As vtctl performs operations, it updates the necessary changes to the lockserver (zookeeper). The rest of the vitess servers observe those changes and react accordingly. For example, if a master database if failed over to a new one, the vitess servers will see the change and redirect future writes to the new master.

vttablet

One of vttablet's main function is to be a proxy to MySQL. It performs tasks that attempt to maximize throughput as well as to protect MySQL from harmful queries. There is one vttablet per MySQL instance.

vttablet is also capable of executing necessary management tasks initiated from vtctl. It also provides streaming services that are used for filtered replication and data export.

vtocc

Vtocc is the previous version of vttablet. It handles query management (same as vttablet) but is not part of a larger system, it's a standalone program that doesn't require a Topology Server. It is useful for unit tests and when the only required feature is the query service (with connection pooling, query de-dup, ...).

Note we may eventually produce a version of vttablet that runs without a Topology Server, and use it instead of vtocc.

vtgate

vtgate's goal is to provide a unified view of the entire fleet. It will be the server that applications will connect to for queries. It will analyze, rewrite and route queries to various vttablets, and return the consolidated results back to the client.

vtctld

vtctld is an HTTP server that lets you browse the information stored in the lockserver. This is useful for trouble-shooting, or to get a good high level picture of all the servers and their current state.

vtworker

vtworker is meant to host long-running processes. It supports a plugin infrastructure, and offers libraries to easily pick tablets to use. We have developped: - resharding differ jobs: meant to check data integrity during shard splits and joins. - vertical split differ jobs: meant to check data integrity during vertical splits and joins.

It is very easy to add other checker processes for in-tablet integrity checks (verifying foreign key-like relationships), and cross shard data integrity (for instance, if a keyspace contains an index table referencing data in another keyspace).

vtprimecache

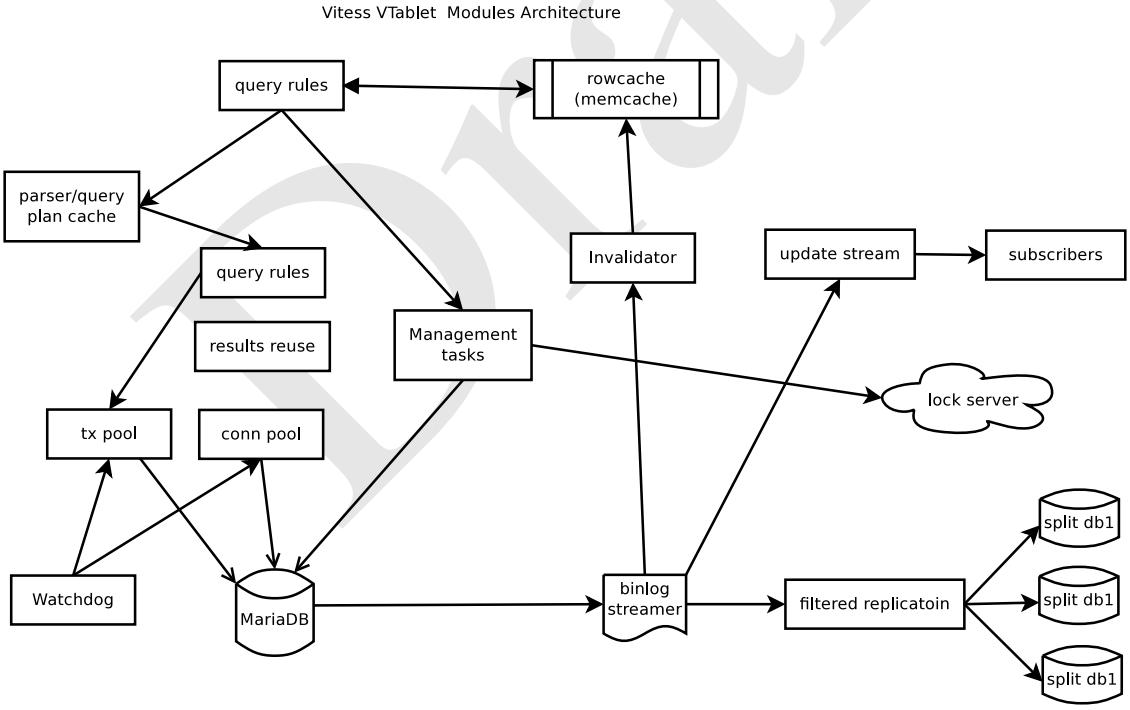
vtprimecache is a mysql cache primer for faster replication. If the single MySQL replication thread is falling behind, vtprimecache activates and starts reading the available relay logs. It then uses a few threads / connections to MySQL to execute modified statements and prime the MySQL buffer cache. The idea is for instance if an 'update table X where id=2' statement is going to be executed by the replication SQL thread 2 or 3 seconds from now, might as well execute a concurrent 'select from table X where id=2' now and prime the MySQL buffer cache. In practice, this shows a speed improvement in replication speed by 30 to 40 percents.

Other support tools

- *mysqlctl*: manage MySQL instances.
- *zkctl*: manage ZooKeeper instances.
- *zk*: command line ZooKeeper client and explorer.

3.2.1 Vitess components block diagram

Figure 3.1: components block diagram



3.3 vttablet

Smart middleware sitting in front of MySQL and serving clients requests.

- Connection pooling.
- SQL parser: Although very close, the vtocc SQL parser is not SQL-92 compliant. It has left out constructs that are deemed uncommon or OLTP-unfriendly. It should, however, allow most queries used by a well-behaved web application.
- Query rewrite and sanitation (adding limits, avoiding non-deterministic updates).
- Query consolidation: reuse the results of an in-flight query to any subsequent requests that were received while the query was still executing.
- Rowcache: the mysql buffer cache is optimized for range scans over indices and tables. Unfortunately, its not good for random access by primary key. The rowcache will instead maintain a row based cache (using memcached as its backend) and keep it consistent by fielding all DMLs that could potentially affect them.
- Update stream: A server that streams the list of rows that are changing in the database, which can be used as a mechanism to continuously export the data to another data store.
- Integrated query killer for queries that take too long to return data.
- Discard idle backend connections to avoid offline db errors.
- Transaction management: Ability to limit the number of concurrent transactions and manage deadlines.

3.4 Reparenting

This document describes the reparenting features of Vitess. Reparenting is used when the master for a Shard is changing from one host to another. It can be triggered (for maintenance for instance) or happen automatically (based on the current master dying for instance).

Two main types of reparenting supported by Vitess are Active Reparents (the Vitess toolchain is handing it all) and External Reparents (another tool is responsible for reparenting, and the Vitess toolchain just update its internal state.

3.4.1 Active Reparents

They are triggered by using the 'vtctl ReparentShard' command. See the help for that command. It currently doesn't use transaction GroupId.

3.4.2 External Reparents

In this part, we assume another tool has been reparenting our servers. We then trigger the 'vtctl ShardExternallyReparented' command.

The flow for that command is as follows: - the shard is locked in the global topology server. - we read the Shard object from the global topology server. - we read all the tablets in the replication graph for the shard. We also check the new master is in the map. Note we allow partial reads here, so if a data center is down, as long as the data center containing the new master is up, we keep going. - we call the 'SlaveWasPromoted' remote action on the new master. This remote action makes sure the new master is not a MySQL slave of another server (the 'show slave status' command should not return anything, meaning 'reset slave' should ave been called). - for every host in the replication graph, we call the 'SlaveWasRestarted' action. It takes as paremeter the address of the new master. On each slave, it executes a 'show slave status'. If the master matches the new master, we update the topology server record for that tablet with the new master, and the replication graph for that tablet as well. If it doesn't match, we keep the old record in the replication graph (pointing at whatever master was there before). We optionally Scrap tablets that bad (disabled by default). - if a smaller percentage than a configurable value of the slaves works (80% be default), we stop here. - we then update the Shard object with the new master. - we rebuild the serving graph for that shard. This will update the 'master' record for sure, and also keep all the tablets that have successfully reparented.

Optional Flags: - -accept-success-percents=80: will declare success if more than that many slaves can be reparented - -continue_on_unexpected_master=false: if a slave has the wrong master, we'll just log the error and keep going - -scrap-stragglers=false: will scrap bad hosts

Failure cases: - The global topology server has to be available for locking and modification during this operation. If not, the operation will just fail. - If a single topology server is down in one data center 9and it's nto the master data center), the tablets in that data center will be ignored by the reparent. Provided it doesn't trigger the 80% threshold, this is not a big deal. When the topology server comes back up, just re-run 'vtctl InitTablet' on the tablets, and that will fix their master record. - If scrap-straggler is false (the default), a tablet that has the wrong master will be kept in the replication graph with its original master. When we rebuild the serving graph, that tablet won't be added, as it doesn't have the right master. - if more than 20% of the tablets fails, we don't update the Shard object, and don't rebuild. We assume something is seriously wrong, and it might be our process, not the servers. Figuring out the cause and re-running 'vtctl ShardExternallyReparented' should work. - if for some reasons none of the slaves report the right master (replication is going through a proxy for instance, and the master address is not what the clients are showing in 'show slave status'), the result is pretty bad. All slaves are kept in the replication graph, but with their old (incorrect) master. Next time a Shard rebuild happens, all the servers will disappear. At that point, fixing the issue and then re-parenting will work.

3.4.3 Reparenting And Serving Graph

When reparenting, we shuffle servers around. A server may get demoted, another promoted, and some servers may end up with the wrong master in the replication graph, or scrapped.

It is important to understand that when we build the serving graph, we go through all the servers in the replication graph, and check their masters. If their master is the one we expect (because it is in the Shard record), we keep going and add them to the serving graph. If not, they are skipped, and a warning is displayed.

When such a slave with the wrong master is present, re-running 'vtclt InitTablet' with the right parameters will fix the server. So the order of operations should be to fix mysql replication, make sure it is caught up, run 'vtctl InitTablet', and maybe restart vttablet if needed.

Alternatively, if another reparent happens, and the bad slave recovers and now replicates from the new master, it will be re-added, and resume proper operation.

The old master for reparenting is a specific case. If it doesn't have the right master during the reparent, it will be scrapped (because it's not in the replication graph at all, so it would get lost anyway).

3.5 Resharding

In Vitess, resharding describes the process of re-organizing data dynamically, with very minimal downtime (we manage to completely perform most data transitions with less than 5 seconds of read-only downtime - new data cannot be written, existing data can still be read).

3.5.1 Process

The process to achieve this goal is composed of the following steps:

- pick the original shard(s)
- pick the destination shard(s) coverage
- create the destination shard(s) tablets (in a mode where they are not used to serve traffic yet)
- bring up the destination shard(s) tablets, with read-only masters.
- backup and split the data from the original shard(s)
- merge and import the data on the destination shard(s)
- start and run filtered replication from original to destination shard(s), catch up
- move the read-only traffic to the destination shard(s), stop serving read-only traffic from original shard(s). This transition can take a few hours. We might want to move rdonly separately from replica traffic.
- in quick succession:
  - make original master(s) read-only
  - flush filtered replication on all filtered replication source servers (after making sure they were caught up with their masters)
  - wait until replication is caught up on all destination shard(s) masters
  - move the write traffic to the destination shard(s)
  - make destination master(s) read-write
- scrap the original shard(s)

3.5.2 Applications

The main application we currently support:

- in a sharded keyspace, split or merge shards (horizontal sharding)
- in a non-sharded keyspace, break out some tables into a different keyspace (vertical sharding)

With these supported features, it is very easy to start with a single keyspace containing all the data (multiple tables), and then as the data grows, move tables to different keyspaces, start sharding some keyspaces, ... without any real downtime for the application.

3.5.3 Scaling Up and Down

Here is a quick table of what to do with Vitess when a change is required:

- uniformly increase read capacity: add replicas, or split shards
- uniformly increase write capacity: split shards
- reclaim free space: merge shards / keyspaces
- increase geo-diversity: add new cells and new replicas
- cool a hot tablet: if read access, add replicas or split shards, if write access, split shards.

3.5.4 Filtered Replication

The cornerstone of Resharding is being able to replicate the right data. Mysql doesn't support any filtering, so the Vitess project implements it entirely:

- the tablet server tags transactions with comments that describe what the scope of the statements are (which keyspace_id,
- which table, ...). That way the MySQL binlogs contain all filtering data.
- a server process can filter and stream the MySQL binlogs (using the comments).
- a client process can apply the filtered logs locally (they are just regular SQL statements at this point).

3.6 Production setup

Setting up vitess in production will depend on many factors. Here are some initial considerations:

- *Global Transaction IDs*: Vitess requires a version of MySQL that supports GTIDs, such as Google MySQL 5.1+, MariaDB 10.0+, or MySQL 5.6+. We currently support Google MySQL and MariaDB, with plans to add MySQL 5.6.
- *Firewalls*: Vitess tools and servers assume that they can open direct TCP connection to each other. If you have firewalls between your servers, you may have to add exceptions to allow these communications.
- *Authentication*: If you need authentication, you need to setup SASL, which is supported by Vitess.
- *Encryption:* Vitess RPC servers support SSL. TODO: Document how to setup SSL.
- *MySQL permissions*: Vitess currently assumes that all application clients have uniform permissions. The connection pooler opens a number of connections under the same user (vt_app), and rotates them for all requests. Vitess management tasks use a different user name (vt_dba), which is assumed to have all administrative privileges.
- *Client Language*: We currently support Python and Go. It's not too hard to add support for more languages, and we are open to contributions in this area.

3.6.1 Setting up Zookeeper

Global zk setup TODO: Explain ### Local zk setup TODO: Explain

3.6.2 Launch vttablets

vttablet is designed to run on the same machine as mysql. You'll need to launch one instance of vttablet for every MySQL instance you want to track.

TODO: Specify order and command-line arguments

3.6.3 Launch vtgate(s)

TODO: Explain

3.7 Schema Management

The schema is the list of tables and how to create them. It is managed by vtctl.

3.7.1 Looking at the Schema

The following vtctl commands exist to look at the schema, and validate it's the same on all databases.

`GetSchema <zk tablet path>` displays the full schema for a tablet

`ValidateSchemaShard <zk shard path>` validate the master schema matches all the slaves.

`ValidateSchemaKeyspace <zk keyspace path>` validate the master schema from shard 0 matches all the other tablets in the keyspace.

Example:

```
$ vtctl -wait-time=30s ValidateSchemaKeyspace /zk/global/vt/keyspaces/user
```

3.7.2 Changing the Schema

Goals: - simplify schema updates on the fleet - minimize human actions / errors - guarantee no or very little downtime for most schema updates - do not store any permanent schema data in Topology Server, just use it for actions. - only look at tables for now (not stored procedures or grants for instance, although they both could be added fairly easily in the same manner)

Were trying to get reasonable confidence that a schema update is going to work before applying it. Since we cannot really apply a change to live tables without potentially causing trouble, we have implemented a Preflight operation: it copies the current schema into a temporary database, applies the change there to validate it, and gathers the resulting schema. After this Preflight, we have a good idea of what to expect, and we can apply the change to any database and make sure it worked.

The Preflight operation takes a sql string, and returns a SchemaChangeResult: `go type SchemaChangeResult struct { Error string BeforeSchema *SchemaDefinition AfterSchema *SchemaDefinition }`

The ApplySchema action applies a schema change. It is described by the following structure (also returns a SchemaChangeResult): `go type SchemaChange struct { Sql string Force bool AllowReplication bool BeforeSchema *SchemaDefinition AfterSchema *SchemaDefinition }`

And the associated ApplySchema remote action for a tablet. Then the performed steps are: - The database to use is either derived from the tablet dbName if UseVt is false, or is the _vt database. A use dbname is prepended to the Sql. - (if BeforeSchema is not nil) read the schema, make sure it is equal to BeforeSchema. If not equal: if Force is not set, we will abort, if Force is set, well issue a warning and keep going. - if AllowReplication is false, well disable replication (adding SET sql_log_bin=0 before the Sql). - We will then apply the Sql command. - (if AfterSchema is not nil) read the schema again, make sure it is equal to AfterSchema. If not equal: if Force is not set, we will issue an error, if Force is set, well issue a warning. We will return the following information: - whether it worked or not (doh!) - BeforeSchema - AfterSchema

Use case 1: Single tablet update:

- we first do a Preflight (to know what BeforeSchema and AfterSchema will be). This can be disabled, but is not recommended.
- we then do the schema upgrade. We will check BeforeSchema before the upgrade, and AfterSchema after the upgrade.

Use case 2: Single Shard update:

- need to figure out (or be told) if its a simple or complex schema update (does it require the shell game?). For now we'll use a command line flag.
- in any case, do a Preflight on the master, to get the BeforeSchema and AfterSchema values.

- in any case, gather the schema on all databases, to see which ones have been upgraded already or not. This guarantees we can interrupt and restart a schema change. Also, this makes sure no action is currently running on the databases we're about to change.
- if simple:
- nobody has it: apply to master, very similar to a single tablet update.
- some tablets have it but not others: error out
- if complex: do the shell game while disabling replication. Skip the tablets that already have it. Have an option to re-parent a the end.
- Note the Backup, and Lag servers won't apply a complex schema change. Only the servers actively in the replication graph will.
- the process can be interrupted at any time, restarting it as a complex schema upgrade should just work.

Use case 3: Keyspace update:

- Similar to Single Shard, but the BeforeSchema and AfterSchema values are taken from the first shard, and used in all shards after that.
- We don't know the new masters to use on each shard, so just skip re-parenting all together.

This translates into the following vtctl commands:

`PreflightSchema {-sql=<sql> || -sql_file=<filename>} <zk tablet path>` apply the schema change to a temporary database to gather before and after schema and validate the change. The sql can be inlined or read from a file. This will create a temporary database, copy the existing keyspace schema into it, apply the schema change, and re-read the resulting schema.

```
$ echo "create table test_table(id int);" > change.sql $ vtctl PreflightSchema -sql_file=change.sql
/zk/nyc/vt/tablets/0002009001
```

`ApplySchema {-sql=<sql> || -sql_file=<filename>} [-skip_preflight] [-stop_replication] <zk tablet path>` apply the schema change to the specific tablet (allowing replication by default). The sql can be inlined or read from a file. a PreflightSchema operation will first be used to make sure the schema is OK (unless skip_preflight is specified).

`ApplySchemaShard {-sql=<sql> || -sql_file=<filename>} [-simple] [-new_parent=<zk tablet path>] <zk shard path>` apply the schema change to the specific shard. If simple is specified, we just apply on the live master. Otherwise we do the shell game and will optionally re-parent. if new_parent is set, we will also reparent (otherwise the master won't be touched at all). Using the force flag will cause a bunch of checks to be ignored, use with care.

```
$ vtctl ApplySchemaShard --sql-file=change.sql -simple /zk/global/vt/keyspaces/vtx/shards/0
$ vtctl ApplySchemaShard --sql-file=change.sql -new_parent=/zk/nyc/vt/tablets/0002009002 /zk/global/vt/keys
```

`ApplySchemaKeyspace {-sql=<sql> || -sql_file=<filename>} [-simple] <zk keyspace path>` apply the schema change to the specified shard. If simple is specified, we just apply on the live master. Otherwise we will need to do the shell game. So we will apply the schema change to every single slave.

Chapter 4

Reference

4.1 Concepts

We need to introduce some common terminologies that are used in Vitess: ### Keyspace A keyspace is a logical database. In its simplest form, it directly maps to a MySQL database name. When you read data from a keyspace, it is as if you read from a MySQL database. Vitess could fetch that data from a master or a replica depending on the consistency requirements of the read.

When a database gets sharded, a keyspace maps to multiple MySQL databases, and the necessary data is fetched from one of the shards. Reading from a keyspace gives you the impression that the data is read from a single MySQL database.

Shard

A division within a Keyspace. All the instances inside a Shard have the same data (or should have the same data, modulo some replication lag).

A Keyspace usually has one shard when not using any sharding (we name it '0' by convention). When sharded, a Keyspace will have N shards (usually, N is a power of 2) with non-overlapping data.

We support dynamic resharding, when one shard is split into 2 shards for instance. In this case, the data in the source shard is duplicated into the 2 destination shards, but only during the transition. Afterwards, the source shard is deleted.

A shard usually contains one MySQL master, and many MySQL slaves. The slaves are used to serve read-only traffic (with eventual consistency guarantees), run data analysis tools that take a long time to run, or perform administrative tasks (backups, restore, diffs, . . .)

Tablet

A tablet is a single server that runs: - a MySQL instance - a vttablet instance - a local row cache instance - an other per-db process that is necessary for operational purposes

It can be idle (not assigned to any keyspace), or assigned to a keyspace/shard. If it becomes unhealthy, it is usually changed to scrap.

It has a type. The commonly used types are: - master: for the mysql master, RW database. - replica: for a mysql slave that serves read-only traffic, with guaranteed low replication latency. - rdonly: for a mysql slave that serves read-only traffic for backend processing jobs (like map-reduce type jobs). It has no real guaranteed replication latency. - spare: for a mysql slave not use at the moment (hot spare). - experimental, schema, lag, backup, restore, checker, . . . : various types for specific purposes.

Only master, replica and rdonly are advertised in the Serving Graph.

Keyspace id

A keyspace id (keyspace_id) is a column that is used to identify a primary entity of a keyspace, like user, video, order, etc. In order to shard a database, all tables in a keyspace need to contain a keyspace id column. Vitess sharding ensures that all rows that have a common keyspace id are always together.

It's recommended, but not necessary, that the keyspace id be the leading primary key column of all tables in a keyspace.

If you do not intend to shard a database, you do not have to designate a keyspace_id. However, you'll be required to designate a keyspace_id if you decide to shard a currently unsharded database.

A keyspace_id can be an unsigned number or a binary character column (unsigned bigint or varbinary in mysql tables). Other data types are not allowed because of ambiguous equality or inequality rules.

TODO: The keyspace id rules need to be solidified once VTGate features are finalized.

Shard graph

The shard graph defines how a keyspace has been sharded. It's basically a per-keyspace list of non-intersecting ranges that cover all possible values a keyspace id can cover. In other words, any given keypsace id is guaranteed to map to

one and only one shard of the shard graph.

We are going with range based sharding. The main advantage of this scheme is that the shard map is a simple in-memory lookup. The downside of this scheme is that it creates hot-spots for sequentially increasing keys. In such cases, we recommend that the application hash the keys so they distribute more randomly.

For instance, an application may use an incrementing UserId as a primary key for user records, and a hashed version of that UserId as a keyspace_id. All data related to one user will be on the same shard, as all rows will share that keyspace_id.

Replication graph

The Replication Graph represents the relationships between the master databases and their respective replicas. This data is particularly useful during a master failover. Once a new master has been designated, all existing replicas have to repointed to the new master so that replication can resume.

Serving graph

The Serving Graph is derived from the shard and replication graph. It represens the list of active servers that are available to serve queries. VTGate (or smart clients) query the serving graph to find out which servers they are allowed to send queries to.

Topology Server

The Topology Server is the backend service used to store the Topology data, and provide a locking service. The implementation we use in the tree is based on Zookeeper. Each Zookeeper process is run on a single server, but may share that server with other processes.

There is a global instance of that service. It contains data that doesn't change often, and references other local instances. It may be replicated locally in each Data Center as read-only copies. (a Zookeeper instance with two master instances per cell and one or two replicas per cell is a good configuration).

There is one local instance of that service per Cell (Data Center). The goal is to transparently support a Cell going down. When that happens, we assume the client traffic is drained out of that Cell, and the system can survive using the remaining Cells. (a Zookeeper instance running on 3 or 5 hosts locally is a good configuration).

The data is partitioned as follows: - Keyspaces: global instance - Shards: global instance - Tablets: local instances - Serving Graph: local instances - Replication Graph: the master alias is in the global instance, the master-slave map is in the local cells.

Clients usually just read the local Serving Graph, therefore they only need the local instance to be up. Also, we provide a caching layer for Zookeeper, to survive local Zookeeper failures and scale read-only access dramatically.

Cell (Data Center)

A Cell is a group of servers and network infrastructure collocated in an area. It is usually a full Data Center, or a subset of a full Data Center.

A Cell has an associated Topology Server, hosted in that Cell. Most information about the tablets in a cell is hosted in that cell's Topology Server. That way a Cell can be taken down and rebuilt as a unit, for instance.

We try to limit cross-cell traffic (both for data and metadata), and gracefully handle cell-level failures (like a Cell being cut off the network). Having the ability to route client traffic to Cells individually is a great feature to have (but not provided by the Vitess software).

4.2 Zookeeper Data

This document describes the information we keep in zookeeper, how it is generated, and how the python client uses it.

4.2.1 Keyspace / Shard / Tablet Data

Keyspace

Each keyspace is now a global zookeeper path, with sub-directories for its shards and action / actionlog. The Keyspace object there contains very basic information.

“‘go // see go/vt/topo/keyspace.go for latest version type Keyspace struct { // name of the column used for sharding // empty if the keyspace is not sharded ShardingColumnName string

```
// type of the column used for sharding
// KIT_UNSET if the keyspace is not sharded
ShardingColumnType key.KeyspaceIdType

// ServedFrom will redirect the appropriate traffic to
// another keyspace
ServedFrom map[TabletType]string
```

} “‘

```
$ zk ls /zk/global/vt/keyspaces/ruser action actionlog shards
```

The path and sub-paths are created by ’vtctl CreateKeyspace’.

We use the action and actionlog paths for locking only, no process is actively watching these paths.

Shard

A shard is a global zookeeper path, with sub-directories for its action / actionlog, and a node for some more data and replication graph.

“‘go // see go/vt/topo/shard.go for latest version // A pure data struct for information stored in topology server. This // node is used to present a controlled view of the shard, unaware of // every management action. It also contains configuration data for a // shard. type Shard struct { // There can be only at most one master, but there may be none. (0) MasterAlias TabletAlias

```
// This must match the shard name based on our other conventions, but
// helpful to have it decomposed here.
KeyRange key.KeyRange

// ServedTypes is a list of all the tablet types this shard will
// serve. This is usually used with overlapping shards during
// data shuffles like shard splitting.
ServedTypes []TabletType

// SourceShards is the list of shards we're replicating from,
// using filtered replication.
SourceShards []SourceShard

// Cells is the list of cells that have tablets for this shard.
// It is populated at InitTablet time when a tabelt is added
// in a cell that is not in the list yet.
Cells []string
```

}

// SourceShard represents a data source for filtered replication // accross shards. When this is used in a destination shard, the master // of that shard will run filtered replication. type SourceShard struct { // Uid is the unique ID for this SourceShard object. // It is for instance used as a unique index in blp_checkpoint // when storing the position. It should be unique whithin a // destination Shard, but not globally unique. Uid uint32

```
// the source keyspace
Keyspace string

// the source shard
Shard string

// The source shard keyrange
// If partial, len(Tables) has to be zero
KeyRange key.KeyRange

// The source table list to replicate
// If non-empty, KeyRange must not be partial (must be KeyRange{})
Tables []string
```

}

“‘

```
$ zk ls /zk/global/vt/keyspaces/ruser/shards/10-20 action actionlog nyc-0000200278
```

We use the action and actionlog paths for locking only, no process is actively watching these paths.

```
$ zk cat /zk/global/vt/keyspaces/ruser/shards/10-20 { "MasterAlias": { "Cell": "nyc", "Uid":
200278 }, "KeyRange": { "Start": "10", "End": "20" }, "Cells": [ "oe", "yh" ] }
```

The shard path and sub-directories are created when the first tablet in that shard is created.

The Shard object is changed when we add tablets in unknown cells, or when we change the master.

Tablet

A tablet has a path in zookeeper, with its action / actionlog and pid file:

```
$ zk ls /zk/nyc/vt/tablets/0000200308 action actionlog pid
```

We use the action and actionlog paths for remote execution of actions. vttablet will watch that directory and launch a vtaction for every requested action.

A tablet also has a node of type Tablet:

“‘go // see go/vt/topo/tablet.go for latest version type Tablet struct { Parent TabletAlias // the globally unique alias for our replication parent - zero if this is the global master

```
// What is this tablet?
Alias TabletAlias

// Locaiton of the tablet
Hostname string
IPAddr   string

// Named port names. Currently supported ports: vt, vts,
// mysql.
Portmap map[string]int

// Tags contain freeform information about the tablet.
Tags map[string]string
```

```
    // Information about the tablet inside a keyspace/shard
    Keyspace string
    Shard    string
    Type     TabletType

    // Is the tablet read-only?
    State TabletState

    // Normally the database name is implied by "vt_" + keyspace. I
    // really want to remove this but there are some databases that are
    // hard to rename.
    DbNameOverride string
    KeyRange       key.KeyRange

    // BlacklistedTables is a list of tables we're not going to serve
    // data for. This is used in vertical splits.
    BlacklistedTables []string
```

} “‘

```
$ zk cat /zk/nyc/vt/tablets/0000200308 { "Alias": { "Cell": "nyc", "Uid": 200308, }, "Parent":
{ "Cell": "", "Uid": 0 }, "Keyspace": "", "Shard": "", "Type": "idle", "State": "ReadOnly",
"DbNameOverride": "", "KeyRange": { "Start": "", "End": "" } }
```

The Tablet object is created by 'vtctl InitTablet'. Up-to-date information (port numbers, ...) is maintained by the vttablet process. 'vtctl ChangeSlaveType' will also change the Tablet record.

4.2.2 Replication Graph

The data maintained by vt tools is as follows: - it is stored in the global zk cell - the master tablet alias is stored in the Shard object - each cell then has a ShardReplication object that stores to master -> slave pairs.

4.2.3 Serving Graph

The serving graph for a shard is maintained in every cell that contains tablets for that shard. To get all the available keyspaces in a cell, just list the top-level cell serving graph directory:

```
$ zk ls /zk/nyc/vt/ns keyspace1 keyspace2
```

The python client lists that directory at startup to find all the keyspaces.

SrvKeyspace

The keyspace data is stored under /zk/

“‘go // see go/vt/topo/srvshard.go for latest version type SrvShard struct { // Copied from Shard KeyRange key.KeyRange ServedTypes []TabletType

```
    // TabletTypes represents the list of types we have serving tablets
    // for, in this cell only.
    TabletTypes []TabletType

    // For atomic updates
    version int64
```

}

// A distilled serving copy of keyspace detail stored in the local // cell for fast access. Derived from the global keyspace, shards and // local details. // In zk, it is in /zk/local/vt/ns/

```
    // List of available tablet types for this keyspace in this cell.
    // May not have a server for every shard, but we have some.
    TabletTypes []TabletType

    // Copied from Keyspace
    ShardingColumnName string
    ShardingColumnType key.KeyspaceIdType
    ServedFrom         map[TabletType]string

    // For atomic updates
    version int64

}
```

// KeyspacePartition represents a continuous set of shards to // serve an entire data set. type KeyspacePartition struct { // List of non-overlapping continuous shards sorted by range. Shards []SrvShard }

‘‘‘

```
$ zk cat /zk/nyc/vt/ns/rlookup { "Shards": [ { "KeyRange": { "Start": "", "End": "" },
} ], "TabletTypes": [ "master", "rdonly", "replica" ] }
```

The only way to build this data is to run the following vtctl command:

```
$ vtctl RebuildKeyspaceGraph <keyspace>
```

When building a new Cell, this command should be run for every keyspace.

Rebuilding a keyspace graph will: - find all the shard names in the keyspace from looking at the children of /zk/global/vt/keyspaces/

The python client reads the nodes to find the shard map (KeyRanges, TabletTypes, ...)

SrvShard

The shard data is stored under /zk/

```
$ zk cat /zk/nyc/vt/ns/rlookup/0 { "KeyRange": { "Start": "", "End": "" } }
```

EndPoints

We also have per serving type data under /zk/

```
$ zk cat /zk/nyc/vt/ns/rlookup/0/master { "entries": [ { "uid": 200274, "host": "nyc-db274.nyc.youtub
"port": 0, "named_port_map": { "_mysql": 3306, "_vtocc": 8101, "_vts": 8102 } } ] }
```

The shard serving graph can be re-built using the ’vtctl RebuildShardGraph

Note this will rebuild the serving graph for all cells, not just one cell.

Rebuilding a shard serving graph will: - compute the data to write by looking at all the tablets from the replicaton graph - write all the /zk/

The clients read the per-type data nodes to find servers to talk to. When resolving ruser.10-20.master, it will try to read /zk/local/vt/ns/ruser/10-20/master.

4.3 Serving Graph

The Serving Graph is a roll-up view of the state of the system in a consistent state. It is used by clients to know the general topology of the system, and find servers to connect to (EndPoints).

The serving Graph is maintained independently in every cell, and only usually contains information about the servers in that cell.

4.3.1 SrvKeyspace

The toplevel object of the serving graph is the SrvKeyspace object. It exists in each cell that is serving data for that keyspace. It contains: - a shard map, for each serving type. - (optional) information about the sharding key for that keyspace. - (optional) a redirection map, in case some keyspaces are served by another keyspace (this feature is used during vrtical splits).

It is rebuilt by running 'vtctl RebuildKeyspaceGraph'. It is not automatically rebuilt when adding new tablets in a cell. It may also be changed during horizontal and vertical splits.

4.3.2 SrvShard

This object has information about a given shard in a cell. Clients usually skip reading that object, as SrvKeyspace + EndPoints is usually enough.

It is rebuilt: - using 'vtctl RebuildShardGraph' - when a SrvKeyspace is rebuilt - automatically after Reparenting and other similar actions that may affect the graph.

4.3.3 EndPoints

This is just a list of possible servers to connect (host, port map) for a given cell, keyspace, shard and server type (master, replica, ...). It is rebuilt at the same time as SrvShard objects.

4.3.4 Rebuilding the Serving Graph

We mentioned the commands to rebuild the serving graph earlier.

For a given tablet, before adding it to the Serving Graph, we will check its master is the Shard's master. So some hosts may be missing when they were expected. The rebuild process will echo a warning in that case.

4.4 Replication Graph

The replication graph contains the mysql replication information for a shard. Currently, we only support one layer of replication (a single master wiht multiple slaves), but the design doesn't preclude us from supporting hierarchical replication later on.

4.4.1 Master

The current master for a shard is represented in the Shard object as MasterAlias, in the global topology server.

When creating a master (using 'vtctl InitTablet ... master'), we make sure MasterAlias is empty in the Shard record, and refuse to proceed if not (unless -force-master is specified). After creation, we update MasterAlias in the Shard.

4.4.2 Slaves

The slaves are added to the ShardReplication object present on each local topology server. So for slaves, the replication graph is colocated in the same cell as the tablets themselves. This makes disaster recovery much easier: when loosing a data center, the replication graph for other data centers is not lost.

When creating a slave (using 'vtctl InitTablet ... replica' for instance), we get the master record (if not specified) from the MasterAlias of the Shard. We then add an entry in the ReplicationLinks list of the ShardReplication object for the tablets cell (we create ShardReplication if it doesnt exist yet).

4.4.3 Discovery

When looking for all the tablets in a Shard, we look for the Shard record, start the list with the MasterAlias, and read the 'Cells' list. Then for each Cell, we get the ShardReplication object, and find all the tablets, and add them to the list.

If a cell is down, the result is partial. Some actions are resilient to partial results, like reparenting.

4.4.4 Reparenting

Reparenting will update the MasterAlias record in the Shard (after having acquired the Shard lock). See the Reparenting doc for more information.

4.5 Testing On A Ramdisk

The `integration_test` testsuite contains tests that may time-out if run against a slow disk. If your workspace lives on hard disk (as opposed to SSD), it is recommended that you run tests using a ramdisk.

4.6 Setup

First, set up a normal vitess development environment by running `bootstrap.sh` and sourcing `dev.env` (see GettingStarted). Then overwrite the testing temporary directories and make a 2GiB ramdisk at the location of your choice (this example uses `/tmp/vt`):

“‘sh export TEST_TMPDIR=/tmp/vt
mkdir ${TEST_TMPDIR} sudo mount -t tmpfs -o size=2g tmpfs ${TEST_TMPDIR}
export VTDATAROOT=${TEST_TMPDIR} export TEST_UNDECLARED_OUTPUTS_DIR=${TEST_TMPDIR}
“‘

You can now run tests (either individually or as part of `make test`) normally.

4.7 Teardown

When you are done testing, you can remove the ramdisk by unmounting it and then removing the directory:

```
sh sudo umount ${TEST_TMPDIR} rmdir ${TEST_TMPDIR}
```

Appendix A

Making of this book

A.1 Where to download the book source

This is book version hobbit manpages.

If you are using a Unix-like system that has a sufficiently recent version of Python (2.3 or newer) available, it is easy to install Mercurial from source.

1. Download a recent source tarball from http://www.selenic.com/mercurial/download.
2. Unpack the tarball:

```
gzip -dc mercurial-version.tar.gz | tar xf -
```

3. Go into the source directory and run the installer script. This will build Mercurial and install it in your home directory.

```
cd mercurial-version
python setup.py install --force --home=$HOME
```

Once the install finishes, Mercurial will be in the `bin` subdirectory of your home directory. Don't forget to make sure that this directory is present in your shell's search path.

You will probably need to set the `PYTHONPATH` environment variable so that the Mercurial executable can find the rest of the Mercurial packages. For example, on my laptop, I have set it to `/home/bos/lib/python`. The exact path that you will need to use depends on how Python was built for your system, but should be easy to figure out. If you're uncertain, look through the output of the installer script above, And see where the contents of the `mercurial` directory were installed to.

A.2 Book Revision History

A.3 Xymon Revision History

Table A.1: Book Revision History.

People contribute to this book		
Name	Date	Contribution
Henrik Storner	Winter 2002	hobbit manpages created
	Winter 2008	update hobbit manpages to become Xymon manpages
T.J. Yang	12/16/2009	convert troff file to LaTeX
	12/16/2009	Use Dia to draw Xymon Architecture diagram.

Table A.2: Xymon Revision History.

People contribute to Xymon		
Name	Date	Contribution
Defenders	LB	Lucus Radebe
	DC	Michael Duberry
	DC	Dominic Matteo
	RB	Didier Domi
Midfielders	MC	David Batty
	MC	Eirik Bakke
	MC	Jody Morris
Forward	FW	Jamie McMaster
Strikers	ST	Alan Smith
	ST	Mark Viduka

Appendix B

Open Publication License

Version 1.0, 8 June 1999

B.1 Requirements on both unmodified and modified versions

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

> Copyright (c) *year* by *author's name or designee*. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v*x.y* or later (the latest version is presently available at http://www.opencontent.org/openpub/).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section B.6).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

B.2 Copyright

The copyright to each Open Publication is owned by its author(s) or designee.

B.3 Scope of license

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

Severability. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

No warranty. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

B.4 Requirements on modified works

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

B.5 Good-practice recommendations

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

B.6 License options

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase "Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder." to the license reference or copy.

B To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase "Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder." to the license reference or copy.

Bibliography

[Bur] Mike Burrows. Chubby—the chubby lock service for loosely-coupled distributed systems. http://research.google.com/archive/chubby.html.

Index

`PYTHONPATH` environment variable, 37
`Voltron` Vitess Concepts keywords, 11

environment variables
    `PYTHONPATH`, 37

Vitess Concepts
    `Voltron`, 11