Horizontal Sharding with Vitess

Andres Taylor, Rohit Nayak

PlanetScale
- Why Shard?
- What is Vitess?
- Specifying sharding strategies
- Data sharding
- Query Planning
Why Shard?

- Physical limitations:
  - Large database size
  - Large number of rows in a table
  - High QPS / CPU /IO usage requiring high-end hardware
- Massively scalable
- On-demand scaling (up or down)
- More resilient
- Enables the use of commodity hardware
- Isolation of tenants
- Differential SLA for some tenants
Horizontal Sharding

- aka Data sharding
- Common schema on all shards
- Tables spread across databases
- Related rows on the same shard
- Challenges
  - Cross-shard queries
  - Foreign Keys
  - Unique Keys
  - Autoincrement
- Why Shard?
- **What is Vitess?**
- Specifying sharding strategies
- Data sharding
- Query Planning
Vitess is a scalable, distributed database system built around MySQL
### What is Vitess?

- **Cloud Native Database**
- **Highly Available**
- **Massively Scalable**
- **MySQL Compatible**

### Logical Database

- **Many Physical Databases**
- **Query Routing**
- **gRPC Clients**
- **MySQL protocol**
- **Single Connection**

### Works With

- **Database Frameworks**
- **Legacy Code**
- **ORMs**
- **Third-Party Applications**

- **slack**
- **New Relic.**
- **Square**
- **Flipkart**
- **HubSpot**
- **peak**
- **Pinterest**
- **shopify**
- **nozzle**
- **weave**
- **GitHub**
- **JD.com**
- **Quiz of Kings**
- **stitchlabs**
- **PlanetScale**

@vitessio
- Why Shard?
- What is Vitess?
- **Specifying sharding strategies**
- Data sharding
- Query Planning
Sharding In Vitess

- **Vertical Sharding:**
  - Multiple unsharded keyspaces, related tables split across keyspaces
  - Use **MoveTables** VReplication workflows
  - Intermediate step before data sharding

- **Horizontal Sharding:**
  - Sharded Keyspace: defined by a VSchema
  - Sharding Key: per table, one or more columns,
  - Primary Vindex: maps sharding key to shard
  - Secondary Vindexes: for common predicate columns
  - Use **Reshard** VReplication workflows
  - Use Sequences for Autoincrements, backed by unsharded keyspace
  - Reference and **Materialize**’d tables for data locality
Sharding Strategies

- Range-based Sharding
  - \{\}, \{-80,80-\}, \{-80, 80-c0, c0-dc00, dc00-dc80, dc80-\}
  - row => 64 bit keyspace_id, using one or more column values
  - Mapping done by a Vindex function
  - One shard per key range of contiguous keyspace_ids

- Sharding Key: per-row tuple of one or more column values

- Primary Vindex: projects the sharding key to a keyspace id (and hence shard)

- Vindexes defined in a VSchema

- Vindex types: binary, xxhash, custom json map, unicode_loose_xxhash, multicol

- Generic: strategy is not hard coded, nor is the app sharding aware

- Sharding key can be changed using MoveTables workflows
- Why Shard?
- What is Vitess?
- Specifying sharding strategies
- Data sharding
- Query Planning
Performing Reshards

- **Sharding**
  - `Reshard -w wf1 --target-keyspace customer Create --source-shards '0' --target-shards '-80,80-'`

- **Resharding**
  - `Reshard -w wf2 --target-keyspace customer Create --source-shards '-80' --target-shards '80-c0,c0-'`

- **Control plane cli:** `vtctl client`
  - `Create ➔ SwitchTraffic [ ➔ ReverseTraffic ] ➔ Complete`
  - `Show / Progress to debug/monitor`
VReplication Workflows

- Target streams from source vttablets (replica/primary)
- Starts with a Copy phase
  - One table at a time, in batches
  - On Source: Take consistent snapshot, streaming select
  - On Target: Bulk insert into target
  - State maintained in a sidecar database.
  - Between tables/batches, stream binary logs, with dmls for copied ranges
- Move to Running (binlog streaming) phase until cutover
VReplication Workflows

- Fast, eventually consistent
- Near-zero downtime cutover
- Resumable, resilient to:
  - primary failovers,
  - network outage
- Throttling, based on:
  - replica lag
  - history list length
  - custom mysql query: max #connections, #threads_running,
Indicative Performance

- One Table: 170GB, 3.2B rows, 3 secondary indexes
- Copy: 17K rows/s, 13 hours + 4 hours reindex, (42 hours wo reindex)

- One Table: 4.15TB, 7.8B rows, 3 secondary indexes
- Copy: 62K rows/s, 35 hours total, 1=>4 shards

- Performance factors
  - Environment: CPU/IO/Memory, Network latency/bandwidth, MySQL settings
  - Application:
    - #tables, #rows, row widths, data types/blob, PK types, Indexes
    - write/read QPS, large transactions
  - VReplication Settings: Packet Size, Copy phase duration, Parallel copy, Throttling
Sharding Stories

- Scaling Datastores At Slack With Vitess
  https://slack.engineering/scaling-datastores-at-slack-with-vitess/

- Sharding Cash
  https://developer.squareup.com/blog/sharding-cash/

- Horizontally Scaling The Rails Backend Of Shop App With Vitess

- Scaling Etsy Payments With Vitess

- One Million Queries Per Second With MySQL
  https://planetscale.com/blog/one-million-queries-per-second-with-mysql

- Vinted Vitess Voyage: Chapter 3 - The Great Migration
- Why Shard?
- What is Vitess?
- Specifying sharding strategies
- Data sharding
- Query Planning
Meet the vtgate Query Planner

(all the following images by DALL-E, except one)
Beyond Naive Approach
Evolution of the Vitess Planner
The v3 Planner Breakthrough
Gen4 Planner: A New Era
First Steps in Query Planning

Parsing

string -> AST

Semantic Analysis

AST -> AST++
Simplifying Unsharded Queries
From AST to Operator Tree
The Route Operator in Action
Understanding Vindexes in Sharding
Cost Estimation
Optimizing Joins in Query Planning
Tree Rewriting
Phases of Query Planning

```
SELECT count(*)
FROM user u
JOIN user_extra ue
ON u.id = ue.foo
```
Initial tree

Horizon
  └── QueryGraph (user, user_extra)
PHASE: physical transformation

Horizon
└── ApplyJoin (ue.foo cols: )
    ├── Route (Scatter:user)
    │   └── Table (user_extra AS ue)
    └── Route (Unique user[user_vindex|:ue_foo])
        └── Filter (u.id = :ue_foo)
            └── Table (user AS u)
PHASE: horizon expansion

Aggregator (count(*))
   └── ApplyJoin
       ├── Route
       │   └── Table
       └── Route
           └── Filter
               └── Table
PHASE: split aggregation and push under join

Aggregator (sum_count(c3) AS count(*))
  ├── Projection (c1 * c2 AS c3)
  │    ├── ApplyJoin (cols: c1, c2)
  │    │    ├── Aggregator (count(*) c1 GB ue.foo)
  │    │    │    ├── Route
  │    │    │    │    └── Table
  │    │    └── Aggregator (count(*) as c2)
  │    │        ├── Route
  │    │        │    └── Filter
  │    │        └── Table
>>>>>> push aggregation under route

Aggregator
  └── Projection
    └── ApplyJoin
      └── Route
        └── Aggregator(count(*) as c1 GB ue.foo)
          └── Table
        └── Route
          └── Aggregator (count(*) as c2)
            └── Filter
              └── Table
After offset planning

Aggregator (sum_count(0))
  └── Projection (:0 * :1)
        └── ApplyJoin
              └── Route (Scatter:user)
                      └── select count(*), ue.foo
                          from user_extra as ue
                          group by ue.foo
              └── Route (Unique user[user_vindex|:ue_foo])
                      └── select count(*)
                          from `user` as u
                          where u.id = :ue_foo
Creating the Illusion
The Future: gen5 and Cardinality Model
Questions and Discussions