Atomic Distributed Transactions In Vitess

Harshit Gangal
Software Engineer

Manan Gupta
Software Engineer

Achieving *Atomic Commits* across shards

Background

- Multi Commit
- Strict Single Mode
- Consistent Lookup Vindex
- Need for Atomic Cross Shard Commit

- Best Effort Commit
- Commit in a sequential shard order
- Failures reported with shard information
- Application-Level Rollback/Rollforward
- Single shard transactions are ACID

Background

- Multi Commit
- Strict Single Mode
- Consistent Lookup Vindex
- Need for Atomic Cross Shard Commit

- Restrict Transactions to Single Shard
- Rollbacks on Cross Shard

Background

- Multi Commit
- Strict Single Mode
- Consistent Lookup Vindex
- Need for Atomic Cross Shard Commit

- Global Secondary Index
- Open Cross-shard Transaction
- Uses Locking and Transaction sequence

Background

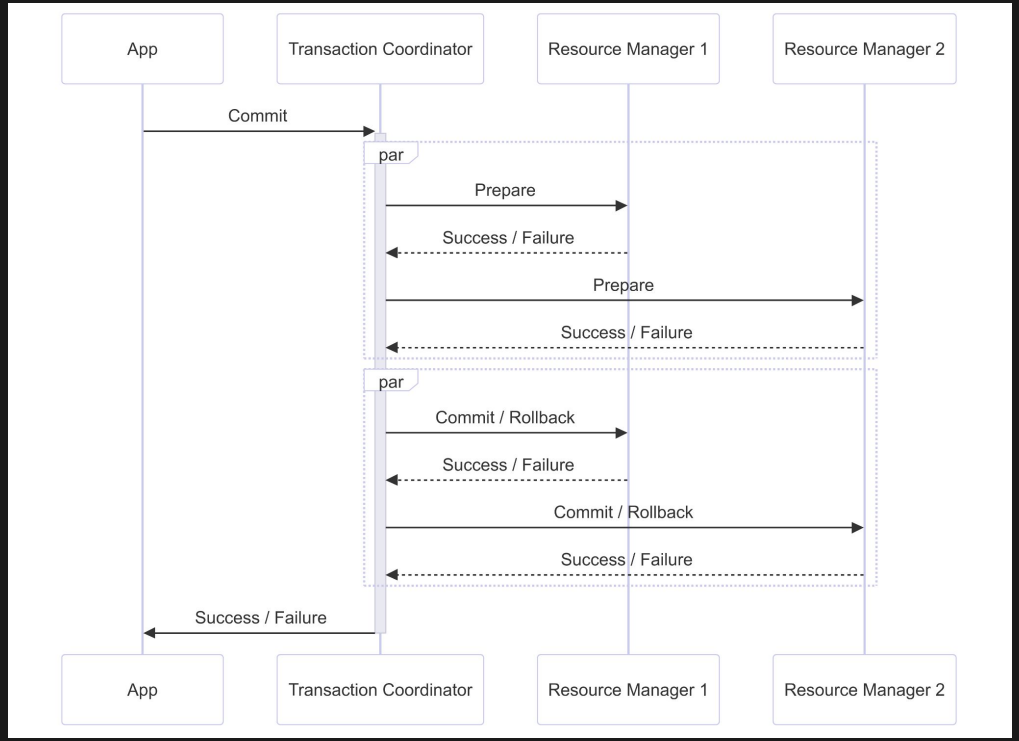
- Multi Commit
- Strict Single Mode
- Consistent Lookup Vindex
- Need for Atomic Cross Shard Commit

- Reduce Application complexity
- Handle failure modes

Two-Phase Commit

Overview

- Transaction Coordinator
- Resource Manager



Two-Phase Commit

Guarantees

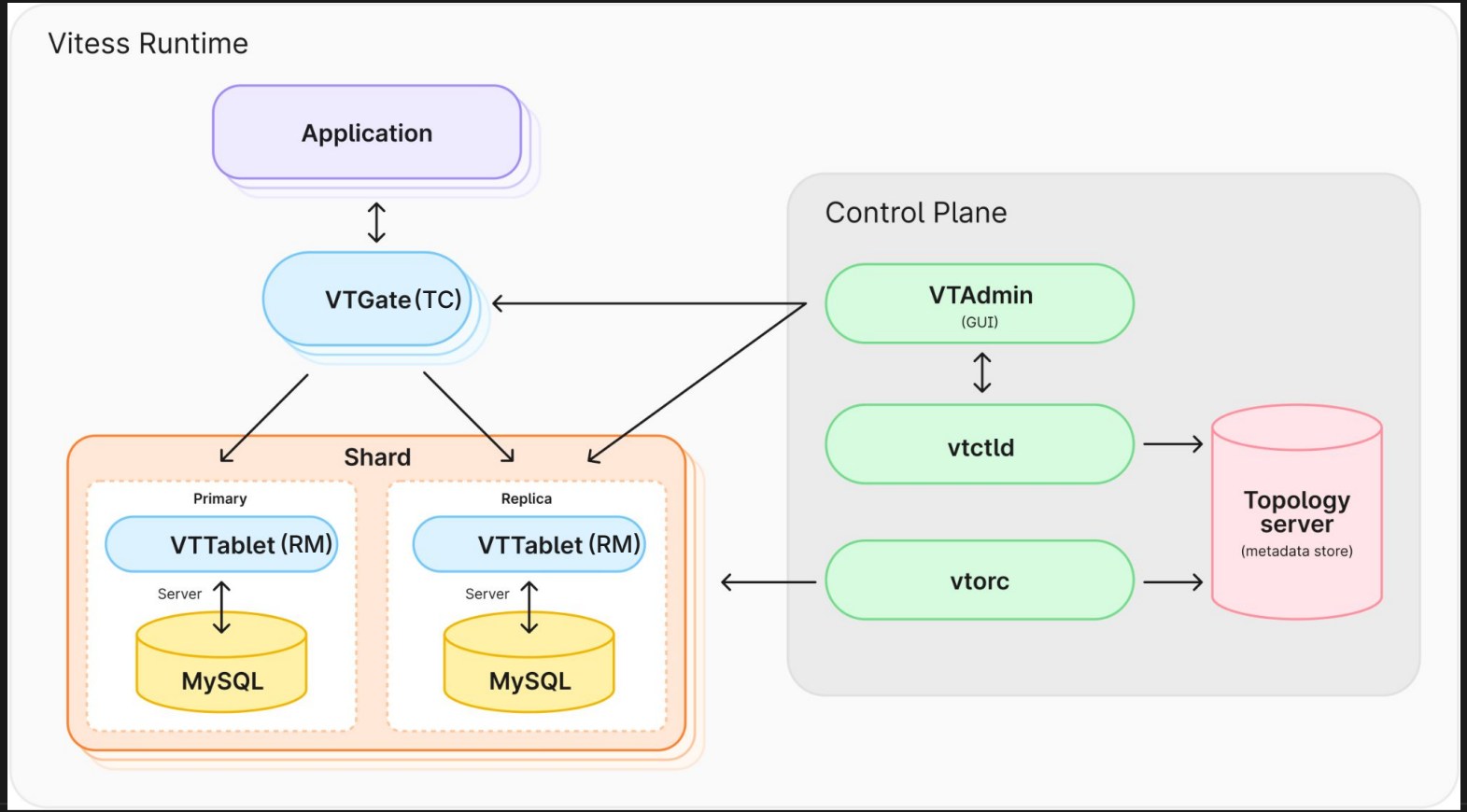
- Prepare Protocol
  - Never abort a transaction, unless requested
  - Never refuse a commit
  - After a crash, reinstate transaction to its prepared state on recovery

Two-Phase Commit

Guarantees

- Prepare Protocol
  - Never abort a transaction, unless requested
  - Never refuse a commit
  - After a crash, reinstate transaction to its prepared state on recovery
- MySQL
  - Transaction does not abort
    - High connection wait timeout
    - No TCP connection
  - Commit almost never fails
    - No group replication
  - Need for Transaction Logs for crash recovery

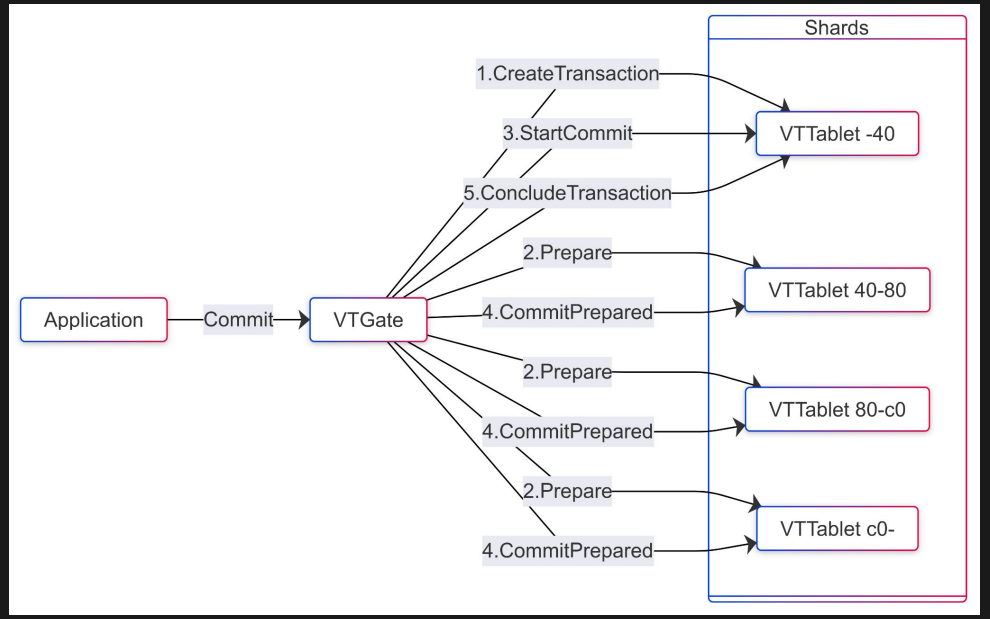
Vitess Architecture



Two Phase Commit Flow

Create Transaction Record

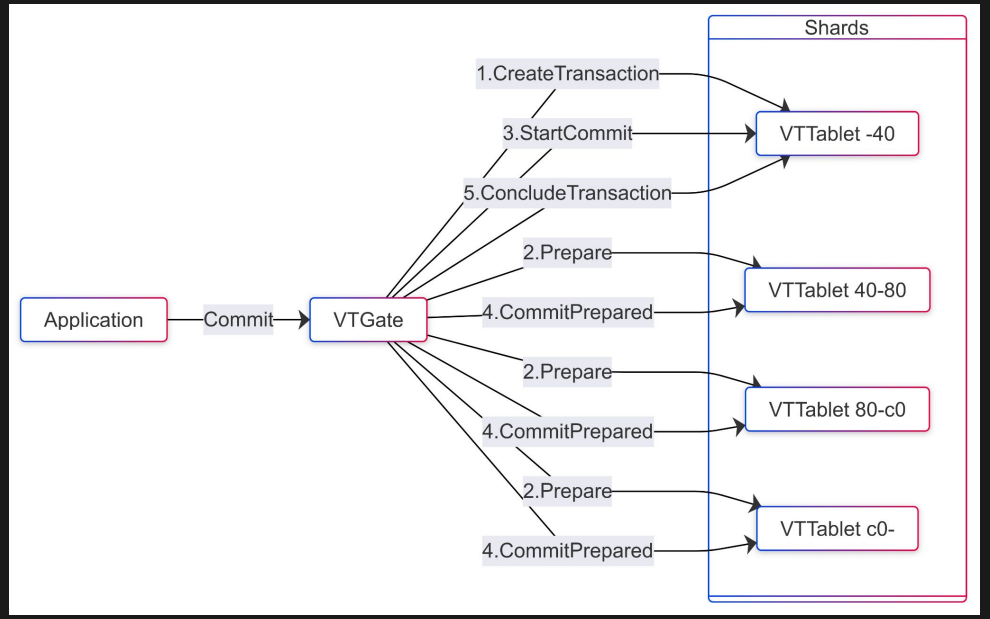
- Generate Unique Distributed Transaction ID (DTID)
- One of Participating VTTablet takes role of Metadata Manager
- Persist the transaction metadata in a separate autocommit transaction



Two Phase Commit Flow

Prepare

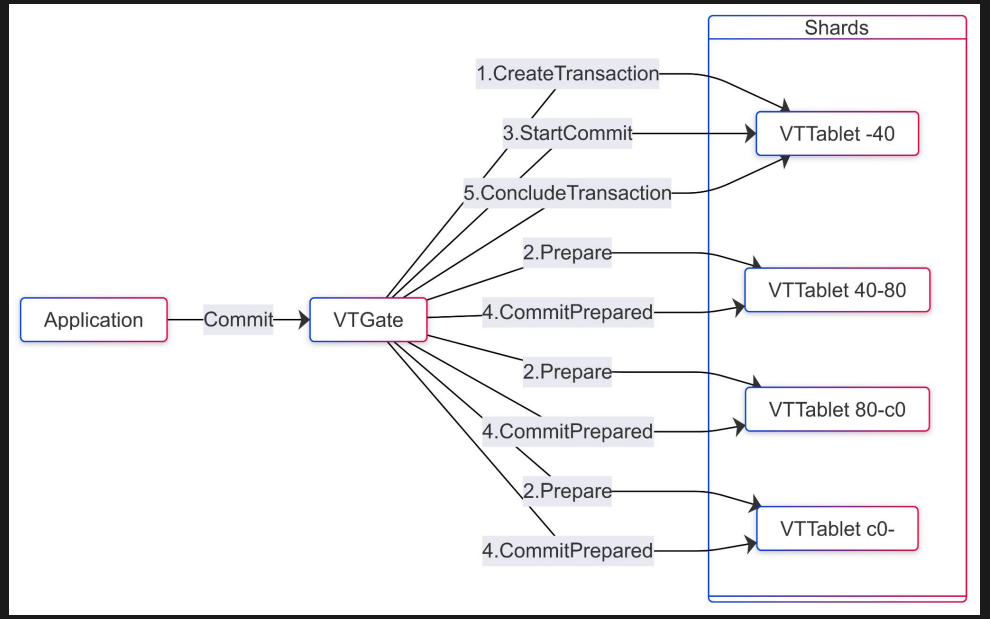
- VTTablet persists the transaction logs in separate autocommit transaction
- Moves open transaction out of transaction timeout scope



Two Phase Commit Flow

Start Commit

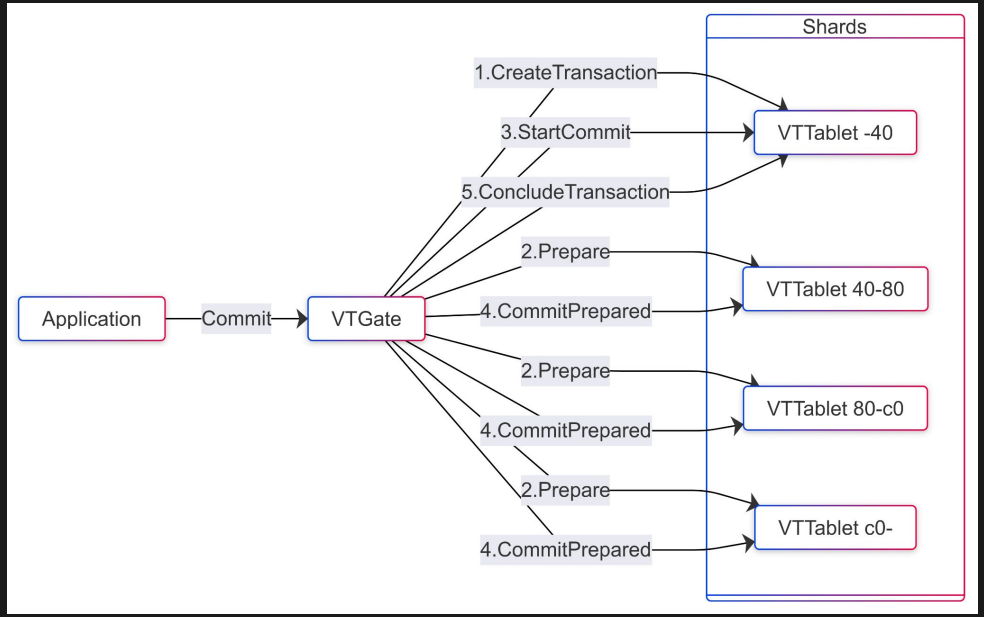
- Metadata Manager marks the transaction status to Commit on same transaction
- Commits the open transaction



Two Phase Commit Flow

Commit Prepared

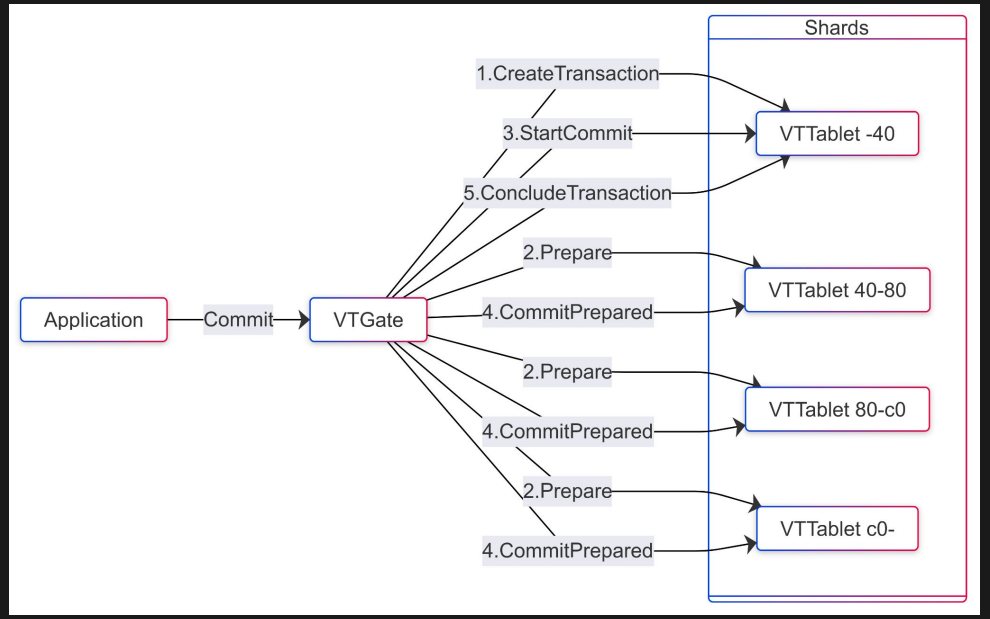
- Commits all the prepared transactions



Two Phase Commit Flow

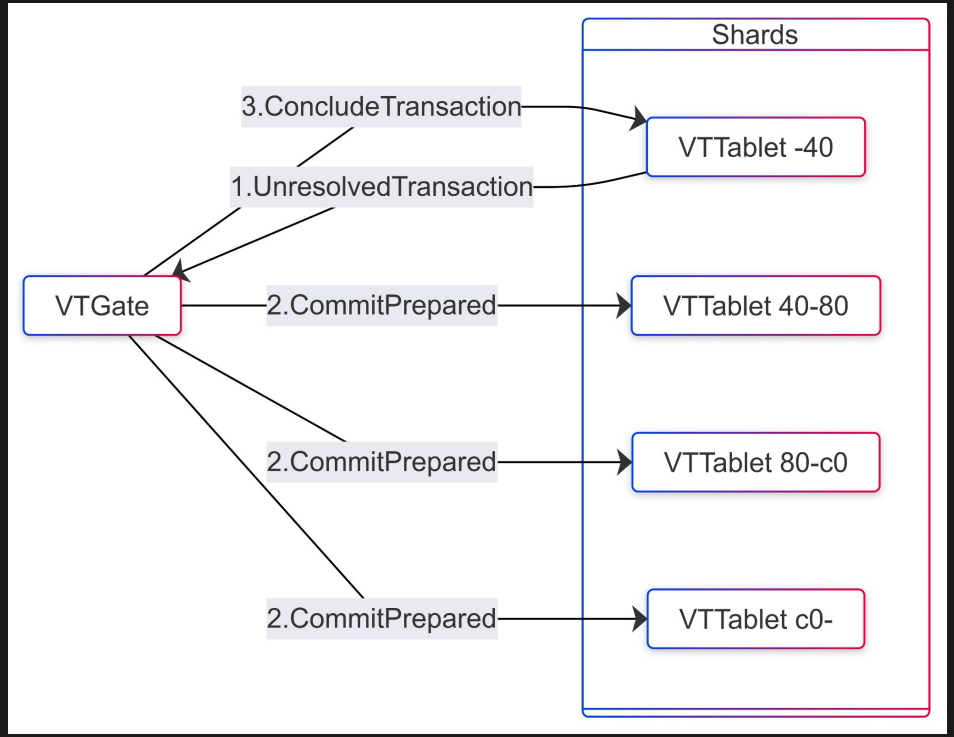
Conclude Transaction

- Removes the Transaction Record



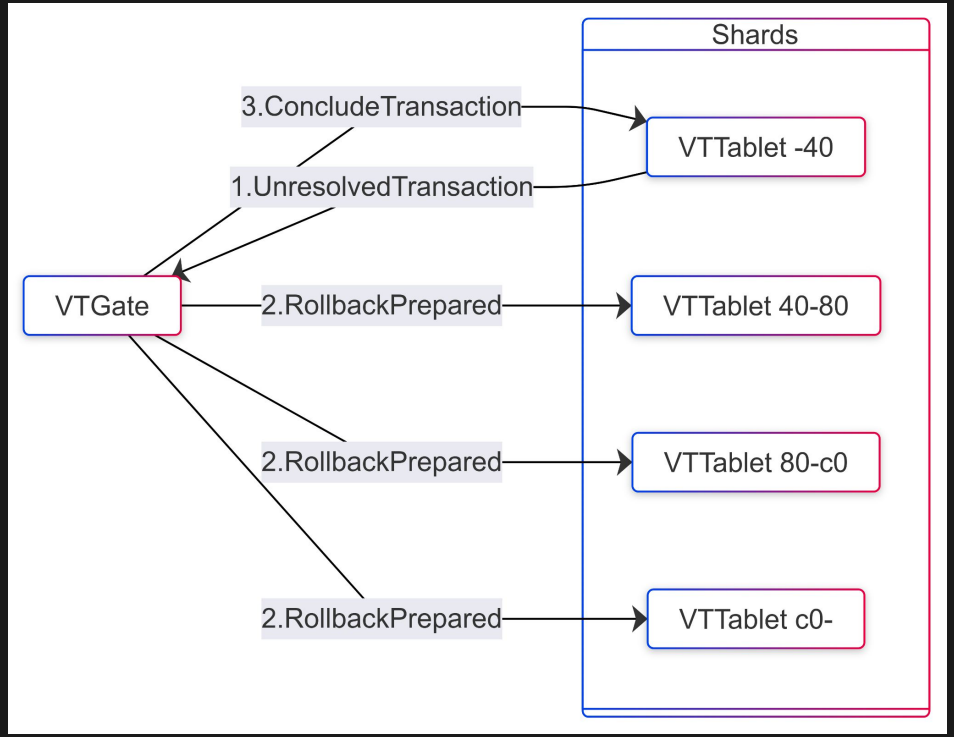
Transaction Resolution Watcher

- Handling unresolved transactions
- Transaction State
  - Commit
  - Rollback
  - Prepare



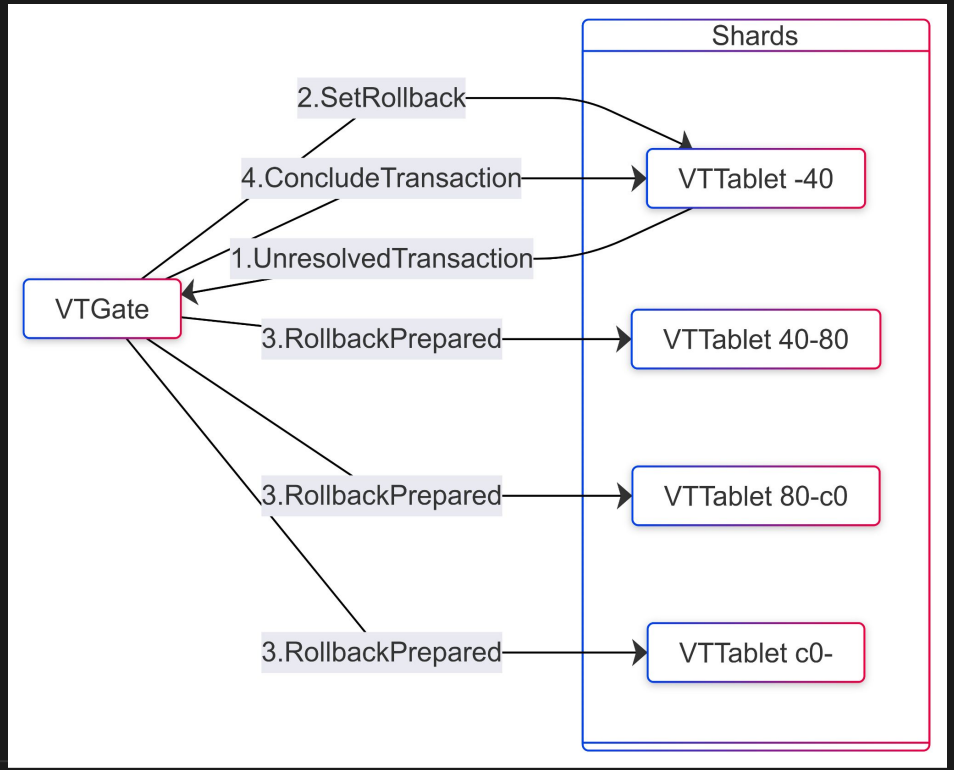
Transaction Resolution Watcher

- Handling unresolved transactions
- Transaction State
  - Commit
  - Rollback
  - Prepare



Transaction Resolution Watcher

- Handling unresolved transactions
- Transaction State
  - Commit
  - Rollback
  - Prepare



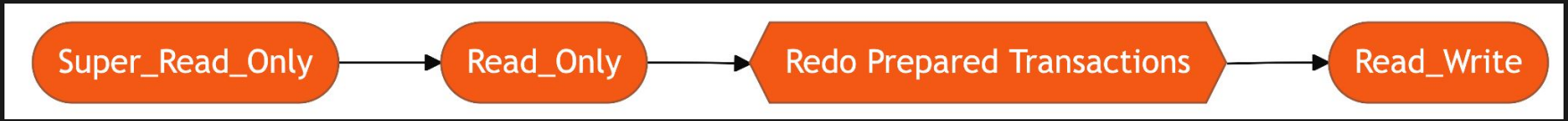
Design Benefits

- Zero impact single shard transactions
- Stateless coordinator model
- Avoiding prepare phase of Metadata Manager
- Relaxed Isolation for scalability

Resilience During Disruptions

MySQL Restarts

- All Ongoing Connections Dropped!
  - This includes all Prepared Transactions
- MySQL starts in Super_Read_Only state



VTTablet Restarts

- Similar to MySQL restart.
- All connections dropped

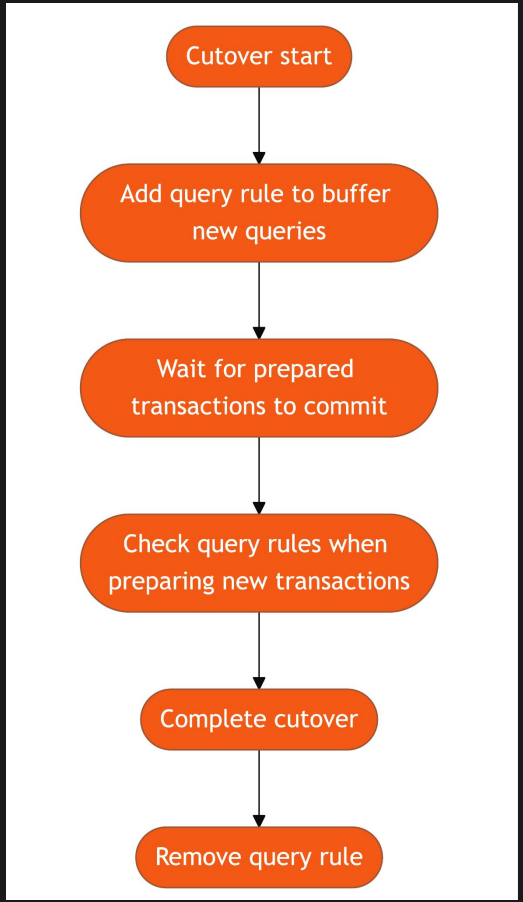
- VTTablet starts in non-serving state

VTGate Restarts

- Nothing to do!
- Transaction resolution watcher takes care of it!

OnlineDDL

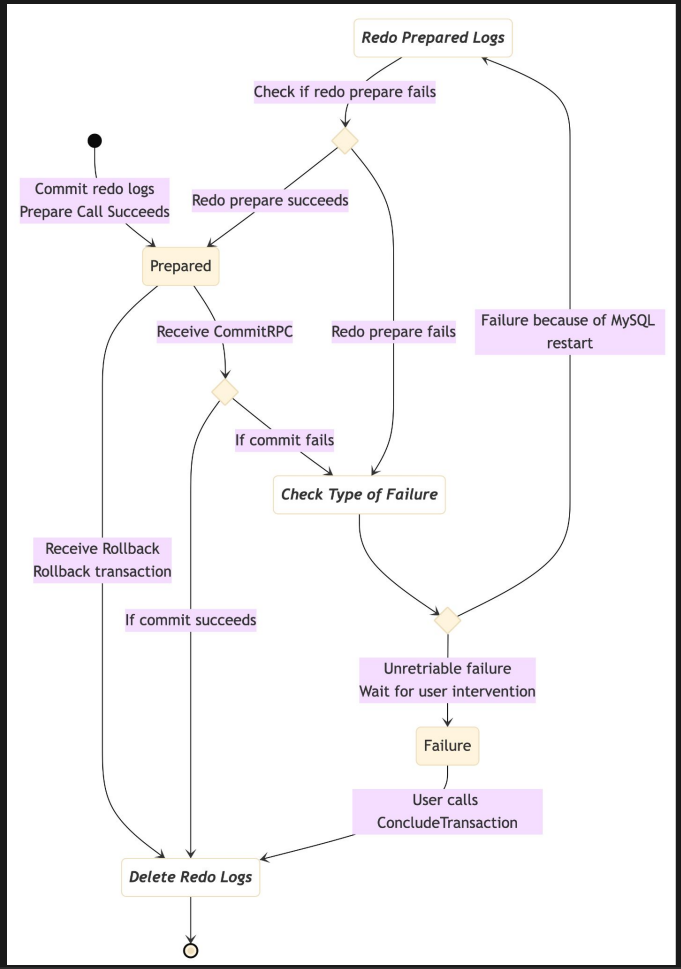
- Cannot have any prepared transactions dependent on the old schema.
- Use query rules to tie everything together.



MoveTables and Reshard

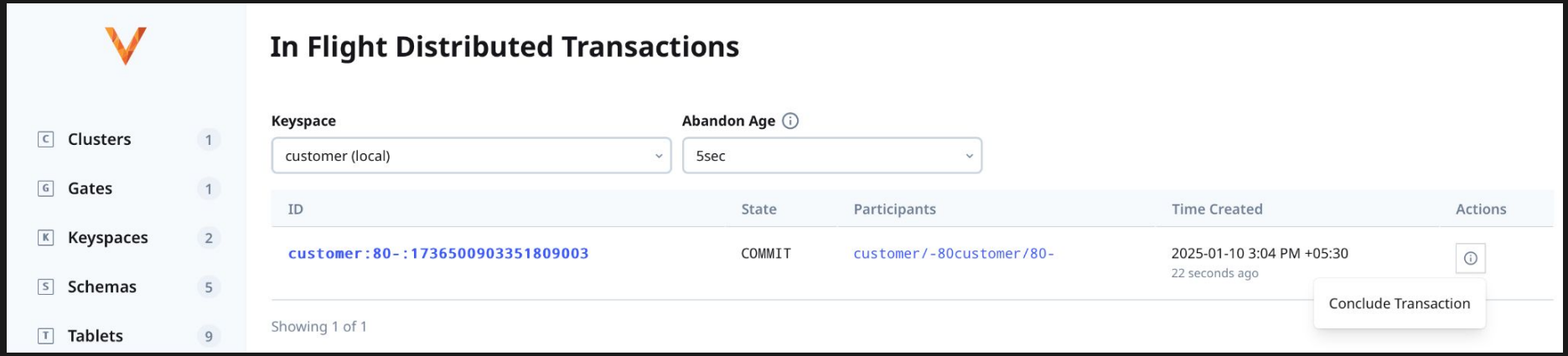
- Very similar solution to OnlineDDL
- Add query rules to fail new queries and prepared transactions
- Wait for open prepared transactions
- Remove the query rule.

VTTablet State Diagram

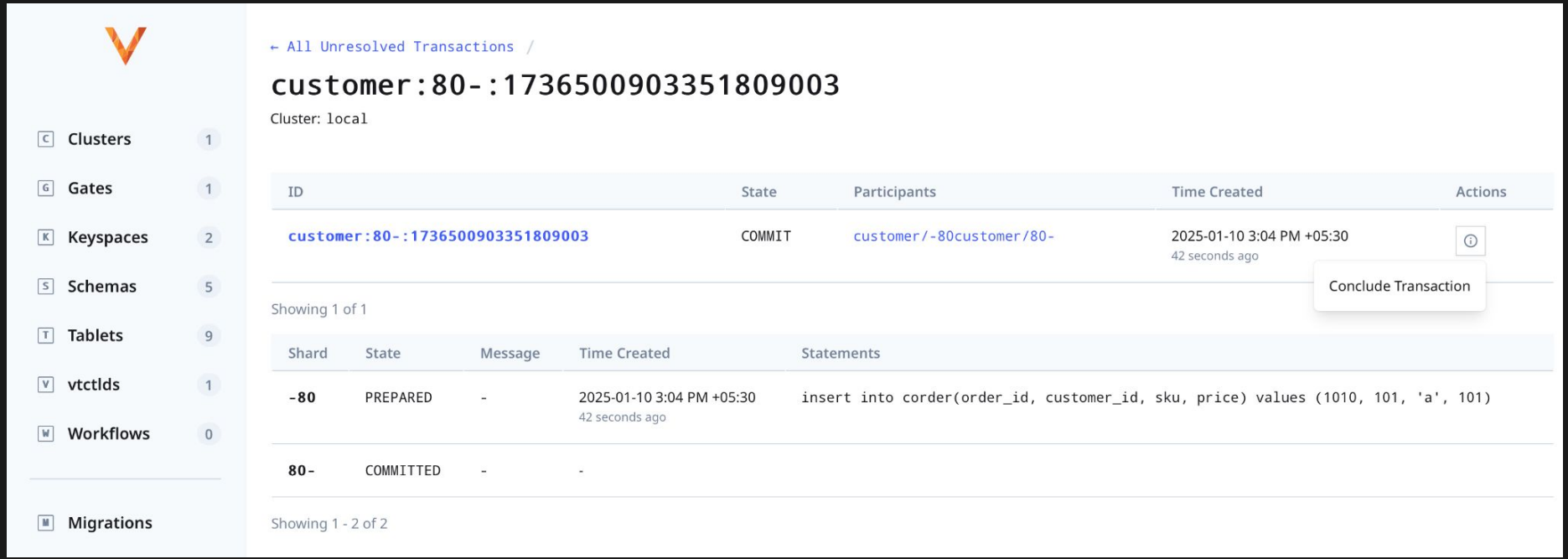


Monitoring

Monitoring - VTAdmin UI



Monitoring - VTAdmin UI



Monitoring - MySQL Query

- In case of a commit failure, `SHOW WARNINGS` can be used to retrieve the dtid.

```
> show transaction status for <dtid>;
+-------------+---------+-------------------------------+-------------------+
| id          | state   | record_time                   | participants      |
+-------------+---------+-------------------------------+-------------------+
| ks:-80:4334 | PREPARE | 2024-07-06 04:05:34 +0000 UTC | ks:80-a0,ks:a0-c0 |
+-------------+---------+-------------------------------+-------------------+
1 row in set (0.00 sec)
```

Monitoring - Metrics & Stats (VTGate)

- CommitUnresolved - Count of transactions that failed during Commit.
- CommitModeTimings - Timings for how long the commit phase takes to complete.

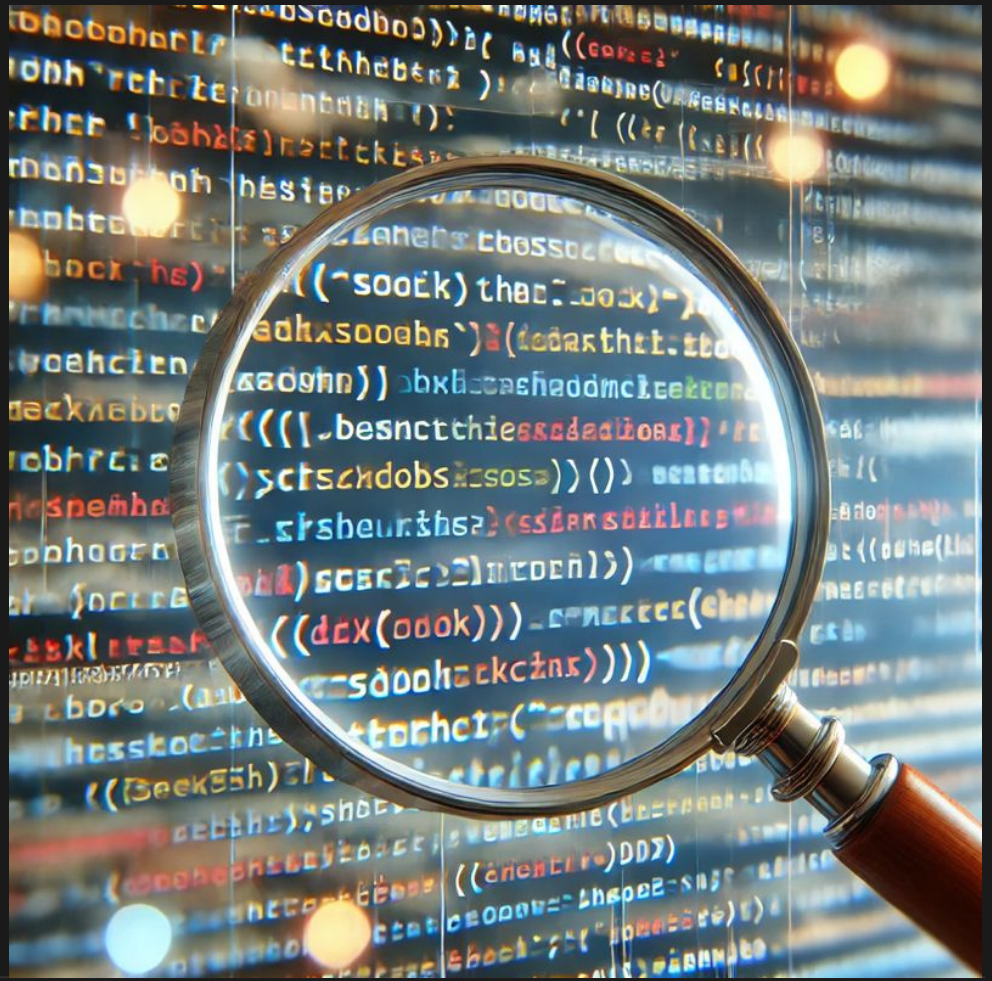

```
"CommitModeTimings": {"TotalCount":3,"TotalTime":32325084,"Histograms":{"Multi":
{"500000":0,"1000000":0,"5000000":0,"10000000":0,"50000000":1,"100000000":0,"500000000":0,"1000000000":0,"5000000000":0,"10000000000":0,"inf":0,"Count":1,"Time":10804250},"TwoPC":
{"500000":0,"1000000":0,"5000000":0,"10000000":1,"50000000":1,"100000000":0,"500000000":0,"1000000000":0,"5000000000":0,"10000000000":0,"inf":0,"Count":2,"Time":21520834}
}},
"CommitUnresolved": 0,
```

Monitoring - Metrics & Stats (VTTablet)

- QueryTimings - Timings of individual phases of 2PC - CreateTransaction, Prepare, StartCommit, SetRollback, CommitPrepared, RollbackPrepared and Resolve.

- Unresolved - Gauge of number of transactions running longer than the abandon age.

Testing



Testing - Challenges

- Ensure everything works as expected.
- Disruption handling code should preserve all the 2PC guarantees
- To test the code, we need a reliable way to inject errors!

Go Build Tags To The Rescue

```go
//go:build debug2PC

package vtgate

import (
    "context"

    "vitess.io/vitess/go/vt/callerid"
    "vitess.io/vitess/go/vt/log"
    querypb "vitess.io/vitess/go/vt/proto/query"
    vtrpcpb "vitess.io/vitess/go/vt/proto/vtrpc"
    "vitess.io/vitess/go/vt/vterrors"
)

const DebugTwoPc = true  no usages  Manan Gupta
```

```go
//go:build !debug2PC

package vtgate

import (
    "context"

    querypb "vitess.io/vitess/go/vt/proto/query"
)

// This file defines debug constants that are always false.
// This file is used for building production code.
// We use go build directives to include a file that defines the constant to true
// when certain tags are provided while building binaries.
// This allows to have debugging code written in normal code flow without affecting
// production performance.

const DebugTwoPc = false  6 usages  Manan Gupta
```

Go Build Tags To The Rescue

```
txPhase = Commit2pcCreateTransaction
if err = txc.tabletGateway.CreateTransaction(ctx, mmShard.Target, dtid, participants); err != nil { return err }

if DebugTwoPc { // Test code to simulate a failure after RM prepare
    if terr := checkTestFailure(ctx, "TRCreated_FailNow", nil); terr != nil {
        return terr
    }
}

txPhase = Commit2pcPrepare
prepareAction := func(ctx context.Context, s *vtgatepb.Session_ShardSession, logging *econtext.ExecuteLogger) error {
    if DebugTwoPc { // Test code to simulate a failure during RM prepare
        if terr := checkTestFailure(ctx, "RMPrepare_-40_FailNow", s.Target); terr != nil { return terr }
    }
    return txc.tabletGateway.Prepare(ctx, s.Target, s.TransactionId, dtid)
}
if err = txc.runSessions(ctx, rmShards, session.GetLogger(), prepareAction); err != nil { return err }

if DebugTwoPc { // Test code to simulate a failure after RM prepare
    if terr := checkTestFailure(ctx, "RMPrepared_FailNow", nil); terr != nil { return terr }
}
```

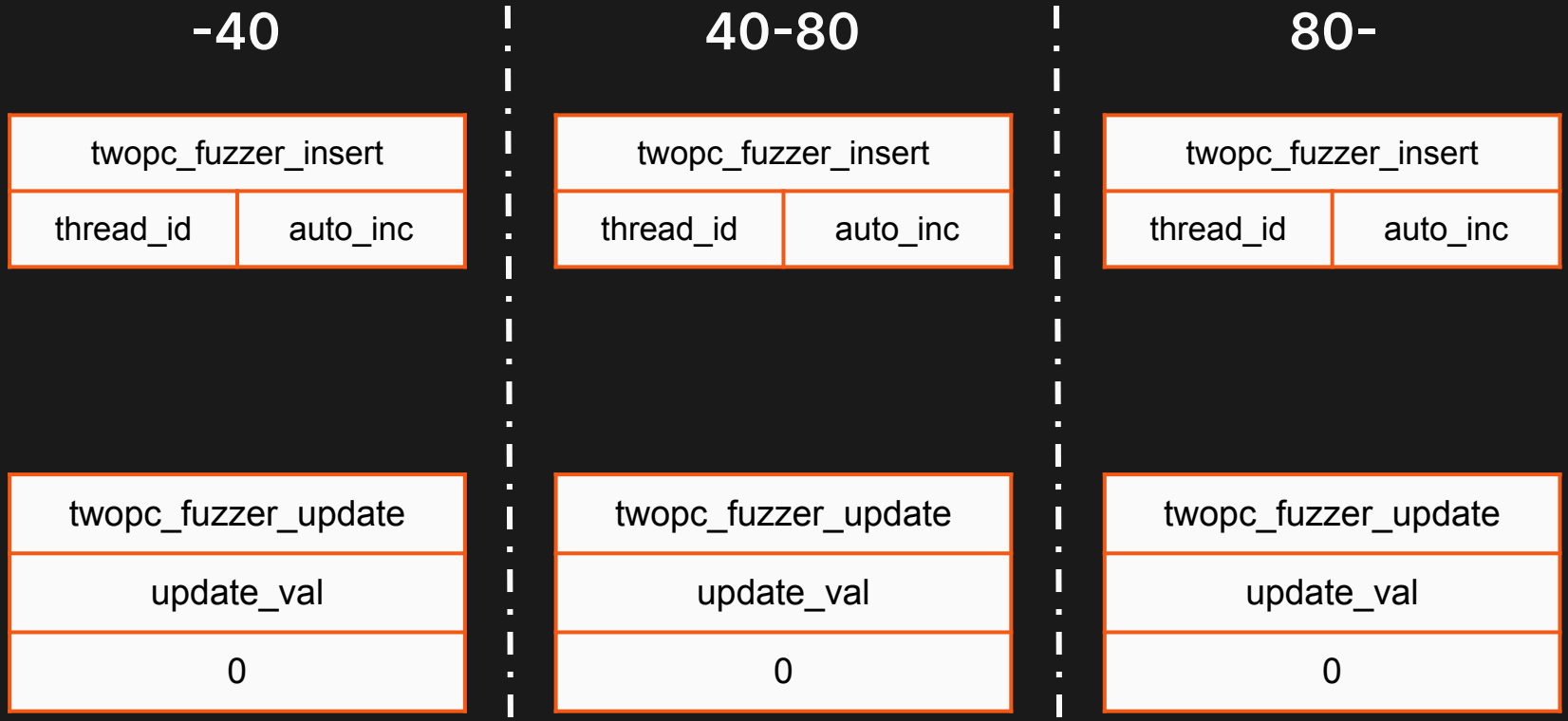
Comprehensive Testing

- Unit Testing
- Disruptions handling tests using gobuild trick to inject delays and ensure guarantees are still met.
- Other miscellaneous end to end tests for metrics, UI, etc.
- Fuzzer testing for extended reliability and to find unknown cases!
    - This worked very well for us in the past with foreign keys work.

Fuzzer Testing Ideas Explained

- Multiple threads running distributed transactions.
- We want to check they're all atomic.
- Another thread running disruptions.
- `twopc_fuzzer_insert` and `twopc_fuzzer_update` tables we use.
- Each transaction inserts a row in the first table, and updates a row in the latter table in all the shards.
- For the insertion, we have an auto-increment column. For the update, we have a column that we increment with a random value, but it is same across all the shards.
- Check atomiticity by ensuring that column in `twopc_fuzzer_update` matches in all shards.
- Check order of commit by ensuring that for each thread, the order of auto increment column matches in `twopc_fuzzer_insert` in all the shards.

Fuzzer Testing - Example Run



Fuzzer Testing - Example Run

-40

twopc_fuzzer_insert	
thread_id	auto_inc
1	1

twopc_fuzzer_update
update_val
23

40-80

twopc_fuzzer_insert	
thread_id	auto_inc
1	3

twopc_fuzzer_update
update_val
23

80-

twopc_fuzzer_insert	
thread_id	auto_inc
1	1

twopc_fuzzer_update
update_val
23

Fuzzer Testing - Example Run

-40

twopc_fuzzer_insert	
thread_id	auto_inc
1	1
2	2

twopc_fuzzer_update
update_val
123

40-80

twopc_fuzzer_insert	
thread_id	auto_inc
1	3
2	4

twopc_fuzzer_update
update_val
123

80-

twopc_fuzzer_insert	
thread_id	auto_inc
1	1
2	3

twopc_fuzzer_update
update_val
123

Fuzzer Testing - Example Run

-40

twopc_fuzzer_insert	
thread_id	auto_inc
1	1
2	2

twopc_fuzzer_update
update_val
123

40-80

twopc_fuzzer_insert	
thread_id	auto_inc
1	3
2	4

twopc_fuzzer_update
update_val
123

80-

twopc_fuzzer_insert	
thread_id	auto_inc
1	1

twopc_fuzzer_update
update_val
23

Fuzzer Testing - Example Run

-40

twopc_fuzzer_insert	
thread_id	auto_inc
1	1
2	2

twopc_fuzzer_update
update_val
123

40-80

twopc_fuzzer_insert	
thread_id	auto_inc
2	3
1	4

twopc_fuzzer_update
update_val
123

80-

twopc_fuzzer_insert	
thread_id	auto_inc
1	1
2	3

twopc_fuzzer_update
update_val
123

Future Enhancements

- Read Isolation Guarantee
- Distributed Deadlock Avoidance


Vitess
PlanetScale