Distributed Transactions in MySQL

Percona Live MySQL Conference & Expo
April 2013

Marcos Albe
Fernando Ipar
Ryan Lowe
Randy Wigginton

What are transactions?

A Simple Sample Transaction

- (A simple bank transfer transaction in SQL)

What are distributed transactions?

A standard interface for coordinating distributed transactions, allowing multiple databases to participate in a transaction while maintaining ACID compliance. When multiple databases participate in the transaction, either all databases commit the changes, or all databases roll back the changes.

A Sample Distributed Transaction

- (A bank transfer transaction in SQL, across 2 hosts [sharded data])

Why do we care?

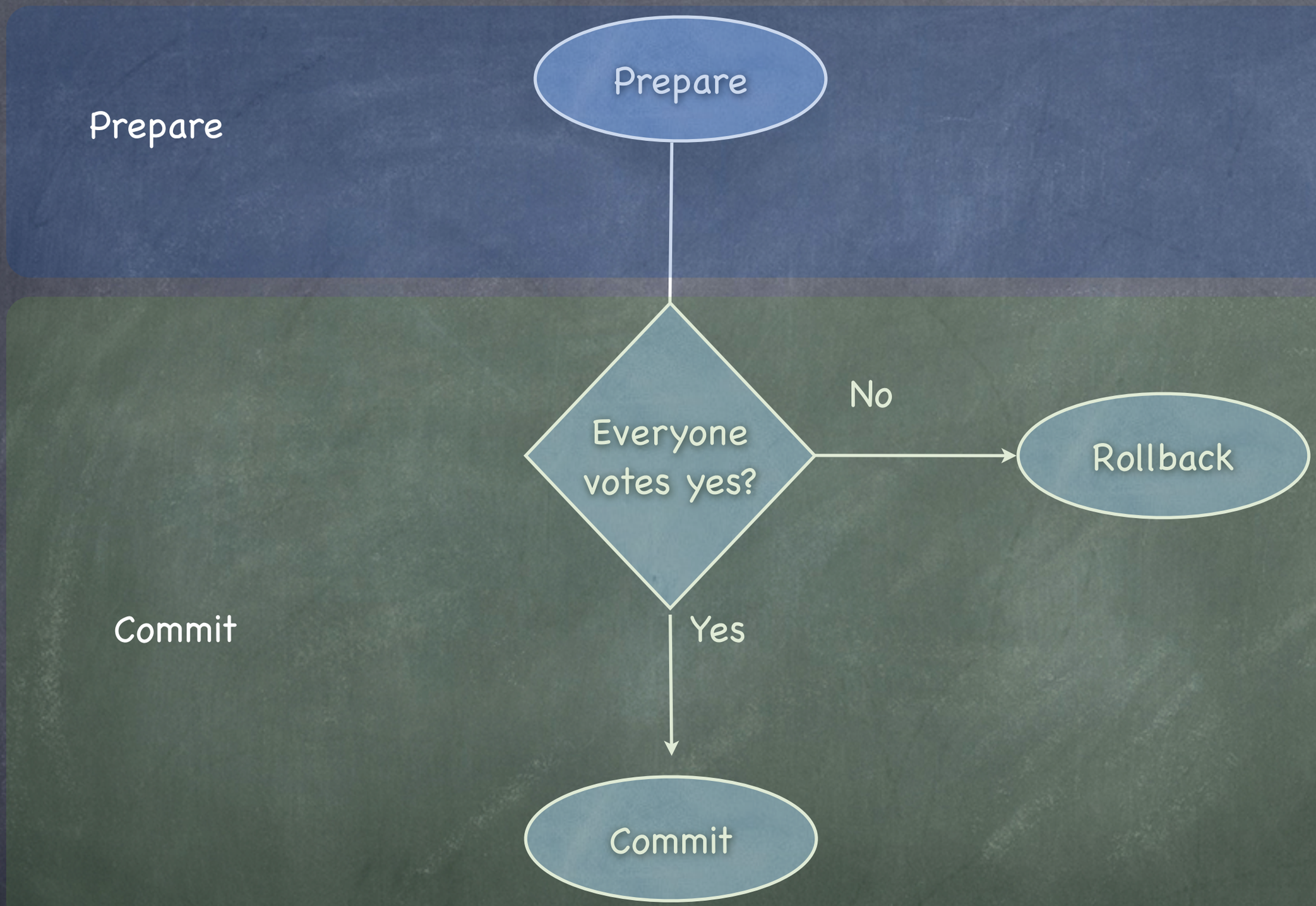
The most compelling reason to understand and use distributed transactions are fault tolerance and high availability. Solutions like Percona XtraDB Cluster and NDB aim to do this within the database management system, but often times application requirements can be more complex. We would want to look at this in order, for example, to protect against datacenter failure.

X/Open

They are an open group who published the standard called “Distributed Transaction Processing: The XA Specification”. There are other proposals for distributed transaction processing, but this is considered to be the gold standard and it is the specification against which the MySQL XA implementation was based.

A DT Walkthrough

Brief discussion of the actors involved. Transaction manager is either a JEE container, or something like what we used (bitronix). Resource manager is anything that supports transactional semantics. Usually DBs, also caches, messanging middleware, etc.



Discuss the two phases using the visual aid. Emphasis on ‘rollback’ and ‘commit’ here being global (i.e. affecting all participants).

Actors

- Transaction manager (coordinator)
- Resource manager

Brief discussion of the actors involved. Transaction manager is either a JEE container, or something like what we used (bitronix). Resource manager is anything that supports transactional semantics. Usually DBs, also caches, messanging middleware, etc.

Preparation

- Initiated by coordinator
- Participants
  - attempt local commit
  - vote

Preparation phase is started by the coordinator/manager. Participants just have to support a special type of commit (commit, but you may have to roll that back later), and vote.

Commit

- <=> everybody votes yes

If all participants vote yes, the coordinator sends a confirmation message, and everyone makes their commit permanent. Mention that the system only works if no node have permanent failure. This is fine, same happens with non-distributed transactions (i.e. they're only guaranteed to work if there is not a permanent failure on the system. If the machine goes down with a corrupt hard drive, you're screwed either way).

XA Transactions

- Open Group
- XA == eXtended Architecture
- Uses two phase commit

Brief introductions to the XA standard, as a formalization of the two phase commit protocol.

XA in MySQL

- Only a resource manager
- innodb_support_xa
- Internal use too

MySQL can only participate as a resource manager. Briefly explain the innodb_support_xa flag, and mention the fact that XA transactions are used internally to coordinate Innodb transactions with binlog writes.

¿Applicability?

Is this really useful at all in the real world?

Examples

- Java
- http://docs.codehaus.org/display/BTM/Home
- https://github.com/fipar/plmce13_xa_examples
- https://github.com/rlowe/distributed_transactions

The examples use JRuby, and the bitronix manager. In our delivery, Marcos and I discussed a couple of slides with source code (up next), and then interactivelly ran examples. Heavy QA during this phase for us.

XA Restrictions

- InnoDB Only
- XA + Replication == Additional Edge Cases
- http://dev.mysql.com/doc/refman/5.6/en/xa-restrictions.html

The examples use JRuby, and the bitronix manager. In our delivery, Marcos and I discussed a couple of slides with source code (up next), and then interactivelly ran examples. Heavy QA during this phase for us.

```
require 'java'
require 'config'

BTM = Java::BitronixTm::bitronixTransactionManager
TxnSvc = Java::BitronixTm::TransactionManagerServices
PDS = Java::BitronixTmResourceJdbc::PoolingDataSource

ds1 = PDS.new
ds1.set_class_name Configuration::DB_CONFIG_1["class_name"]
ds1.set_unique_name 'mysql1'
ds1.set_max_pool_size 3
ds1.get_driver_properties.set_property 'url', Configuration::DB_CONFIG_1["url"]
ds1.init

ds2 = PDS.new
ds2.set_class_name Configuration::DB_CONFIG_2["class_name"]
ds2.set_unique_name 'mysql2'
ds2.set_max_pool_size 3
ds2.get_driver_properties.set_property 'url', Configuration::DB_CONFIG_2["url"]
ds2.init
```


Here we just create two data sources, and a transaction manager.

```
c1 = ds1.get_connection
c2 = ds2.get_connection

btm = TxnSvc.get_transaction_manager

btm.begin

begin
  puts "Checking source account balance"
  rs = c1.create_statement.execute_query "SELECT * FROM xa_examples.accounts WHERE id = 1 AND balance > 100 "
  btm.rollback unless rs.next
  puts "recording withdrawal"
  c1.create_statement.execute_update "INSERT INTO xa_examples.transactions (account_id,amount) VALUES (1,-100)"
  puts "widthdrawing funds"
  c1.create_statement.execute_update "UPDATE xa_examples.accounts SET balance = balance - 100 WHERE id = 1"
  puts "recording credit"
  c2.create_statement.execute_update "INSERT INTO xa_examples.transactions (account_id,amount) VALUES (2,100)"
  puts "crediting funds"
  c2.create_statement.execute_update "UPDATE xa_examples.accounts SET balance = balance + 100 WHERE id = 2"
  btm.commit
  puts "Successfully finished money transfer"
rescue
  puts "Something bad happened while attempting the transfer: " + $!
  btm.rollback
end

btm.shutdown
```


This should be an ok/successful transaction.Note that we invoke begin/rollback/commit on the manager, not on individual connections.

Questions?

For the PLMCE talk, I think we'll need more examples, and basic benchmarks, as I expect people there will inquire about the performance penalty of this.