NewSQL: TDB简介

马如悦@百度 2017.05

Bai du 大数据

分享内容

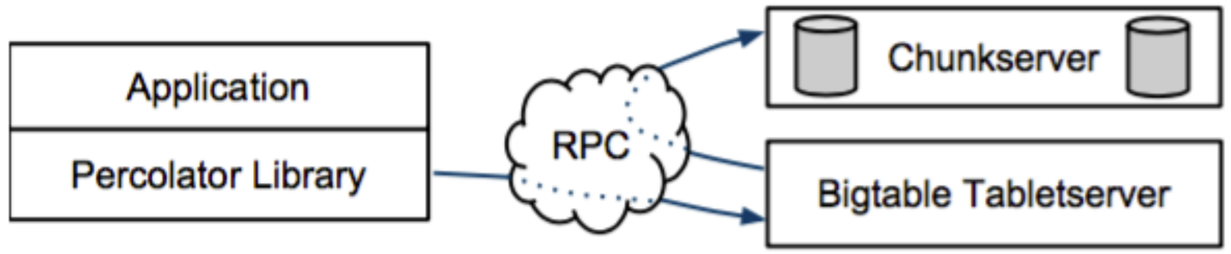
- OLTP
- 相关数据库技术
- TDB 关键技术点

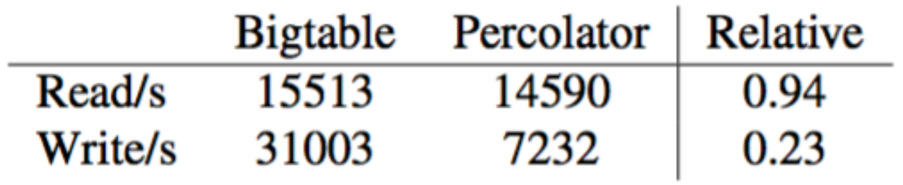
OLTP

- 应用场景
  - 电商订单、CRM系统、金融交易
  - CRUD，ACID
- 方案
  - OldSQL : Oracle、SQL Server、MySQL、PostgreSQL
  - OldSQL+ ： 百度DDBS、阿里DRDS、腾讯TDSQL
  - NoSQL : Mola、Bigtable、Mongodb
  - NewSQL : OceanBase、Spanner、TiDB、NesoiDB、CockroachDB、**TDB**

Percolator（TiDB）

- 索引增量更新系统
  - 批处理太慢，大量重复计算，浪费资源
  - DBMS扩展性太差
- 思路
  - 基于bigtable，提供跨行跨表事务支持
  - 两阶段提交，事务协调信息也存储在bigtable中，和数据混合存储
  - 单机的时间戳服务器Time Oracle， 200w/s
  - 读性能差别不大，写性能下降巨大





	Bigtable	Percolator	Relative
Read/s	15513	14590	0.94
Write/s	31003	7232	0.23

Figure 8: The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

key	bal:data	bal:lock	bal:write
Bob	6: 5: $10	6: 5:	6: data @ 5 5:
Joe	6: 5: $2	6: 5:	6: data @ 5 5:

1. Initial state: Joe's account contains $2 dollars, Bob's $10.

Bob	7:**$3** 6: 5: $10	7: **I am primary** 6: 5:	7: 6: data @ 5 5:
Joe	6: 5: $2	6: 5:	6: data @ 5 5:

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

Bob	7: $3 6: 5: $10	7: I am primary 6: 5:	7: 6: data @ 5 5:
Joe	**7: $9** 6: 5: $2	**7: primary @ Bob.bal** 6: 5:	7: 6: data @ 5 5:

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

Column	*Use*
c:lock	An uncommitted transaction is writing this cell; contains the location of primary lock
c:write	Committed data present; stores the Bigtable timestamp of the data
c:data	Stores the data itself

Bob	8: 7: $3 6: 5: $10	8: **7:** 6: 5:	**8: data @ 7** 7: 6: data @ 5 5:
Joe	7: $9 6: 5: $2	7: primary @ Bob.bal 6: 5:	7: 6:data @ 5 5:

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value $3.

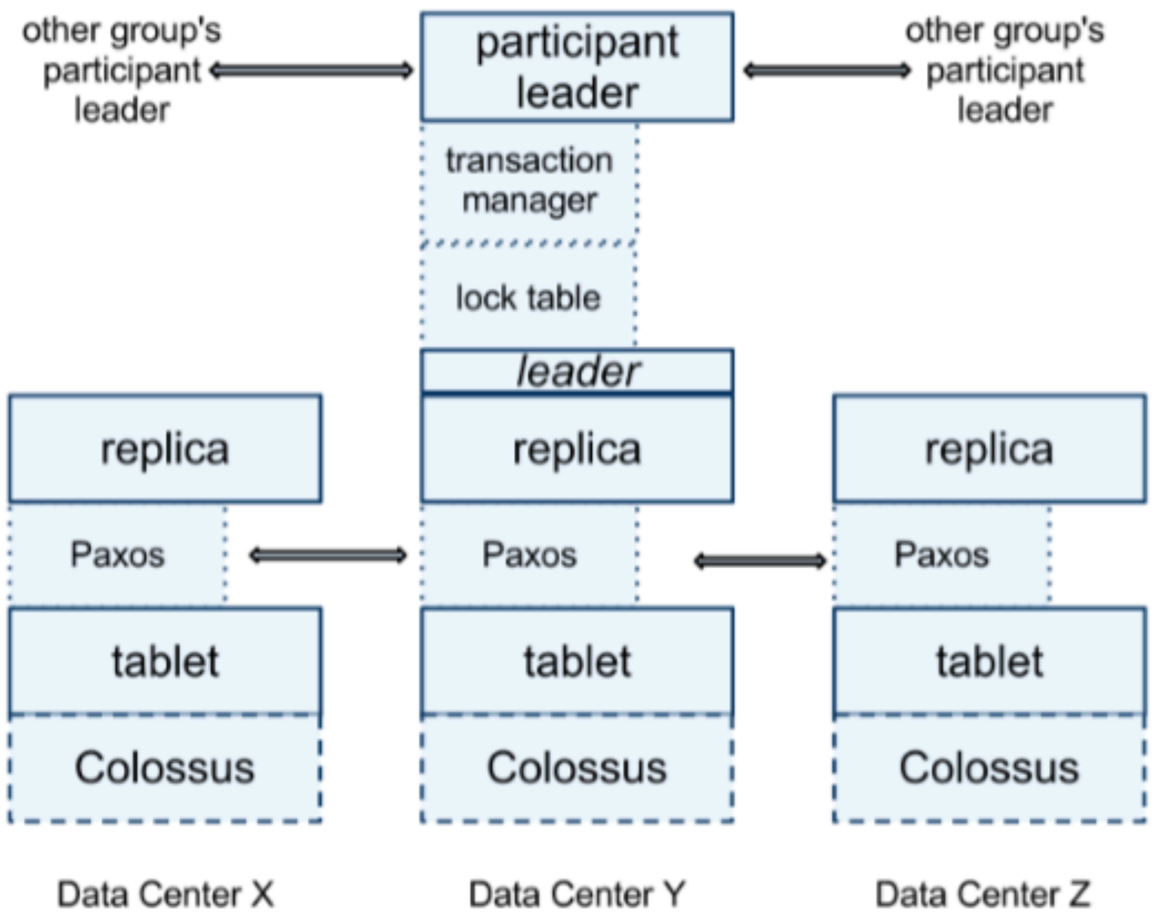
Bob	8: 7: $3 6: 5: $10	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Joe	8: 7: $9 6: 5:$2	8: **7:** 6: 5:	**8: data @ 7** 7: 6: data @ 5 5:

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

```
class Transaction {
 struct Write { Row row; Column col; string value; };
 vector<Write> writes_;
 int start_ts_;

 Transaction() : start_ts_(oracle.GetTimestamp()) {}
 void Set(Write w) { writes_.push_back(w); }
 bool Get(Row row, Column c, string* value) {
  while (true) {
   bigtable::Txn T = bigtable::StartRowTransaction(row);
   // Check for locks that signal concurrent writes.
   if (T.Read(row, c+"lock", [0, start_ts_])) {
    // There is a pending lock; try to clean it and wait
    BackoffAndMaybeCleanupLock(row, c);
    continue;
   }

   // Find the latest write below our start_timestamp.
   latest_write = T.Read(row, c+"write", [0, start_ts_]);
   if (!latest_write.found()) return false; // no data
   int data_ts = latest_write.start_timestamp();
   *value = T.Read(row, c+"data", [data_ts, data_ts]);
   return true;
  }
 }
// Prewrite tries to lock cell w, returning false in case of conflict.
 bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp ...
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // ... or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
    {primary.row, primary.col});      // The primary's location.
  return T.Commit();
 }
 bool Commit() {
  Write primary = writes_[0];
  vector<Write> secondaries(writes_.begin()+1, writes_.end());
  if (!Prewrite(primary, primary)) return false;
  for (Write w : secondaries)
   if (!Prewrite(w, primary)) return false;

  int commit_ts = oracle_.GetTimestamp();

  // Commit primary first.
  Write p = primary;
  bigtable::Txn T = bigtable::StartRowTransaction(p.row);
  if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
   return false;     // aborted while working
  T.Write(p.row, p.col+"write", commit_ts,
      start_ts_); // Pointer to data written at start_ts_.
  T.Erase(p.row, p.col+"lock", commit_ts);
  if (!T.Commit()) return false;      // commit point

  // Second phase: write out write records for secondary cells.
  for (Write w : secondaries) {
   bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
   bigtable::Erase(w.row, w.col+"lock", commit_ts);
  }
  return true;
 }
} // class Transaction
```

Spanner

- paxos保证副本之间的强一致性
- 事务使用两阶段提交
- 跨地域
- TrueTime
  - 通过原子钟和GPS来同步时钟
  - 任何一个spanner server和标准时钟相差不超过e
  - 任何两个spanner server之间的时钟相差不超过2e
  - 时间校准周期短的话，e减少，在30s校准周期下，e=7ms



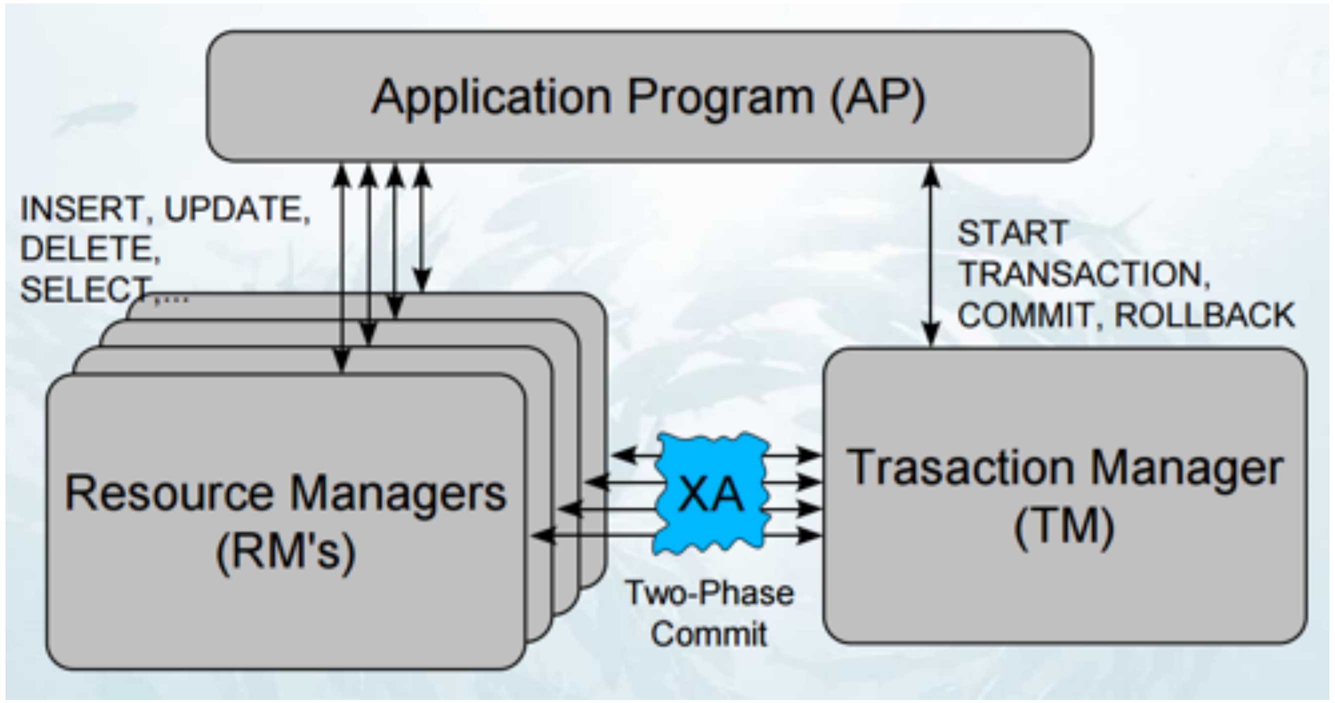
MySQL XA - NesoiDB

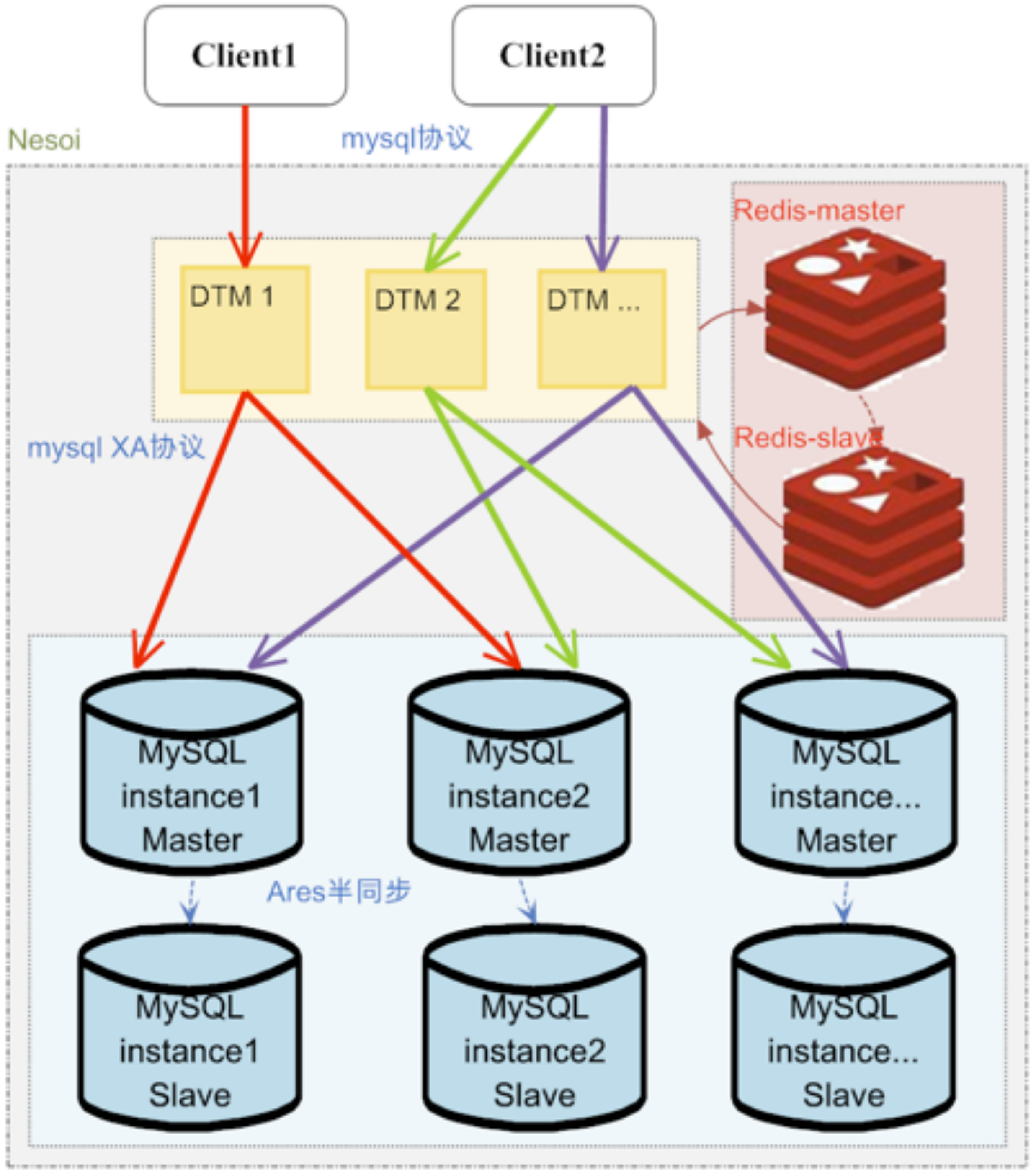
- XA(eXtended Architecture) 标准是由 X/Open group 制定的关于分布式事务 处理的一份声明, 该声明描述了 RM 需要实现的接口, 以便 TM 使用这些接口与RM 进行交互. 这些接口的目的在于让访问多个 RM 的操作处于同一个分布式事务内, 以便保证这些跨 RM 操作的 ACID 属性.

- NesoiDB是百度基于MySQL XA实现的分布式事务数据库，包含对MySQL XA在异常情况下的 bug 修复和DTM的实现.

- NesoiDB的主要问题
  - 性能不优，过多的RPC交互以及单机innoDB事务开销较大
  - 没有死锁避免，只能通过超时来处理死锁
  - MySQL主从半同步复制并不能保证强一致
  - 依赖Redis保存事务的状态，redis并不能保证高可用、强一致

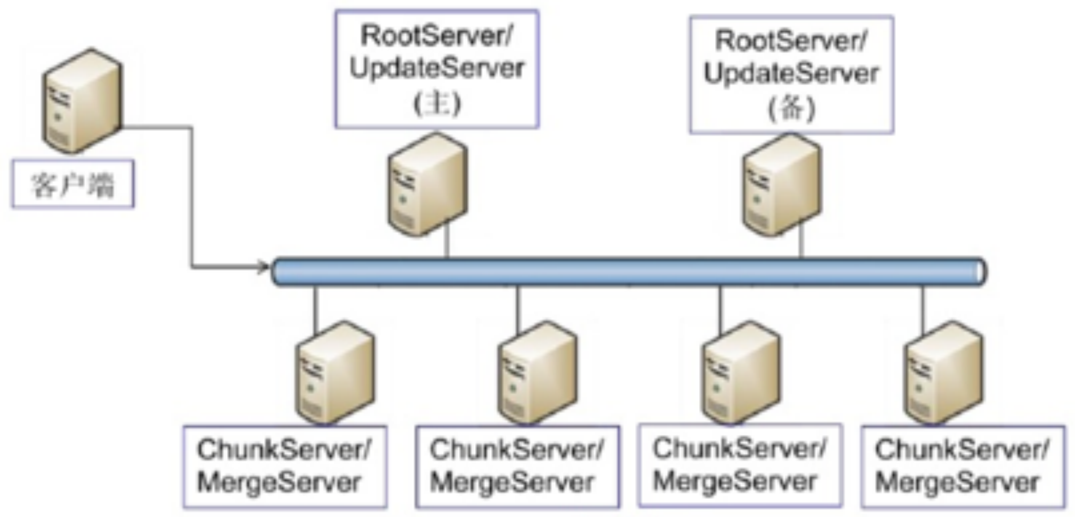
MySQL XA - NesoiDB





OceanBase

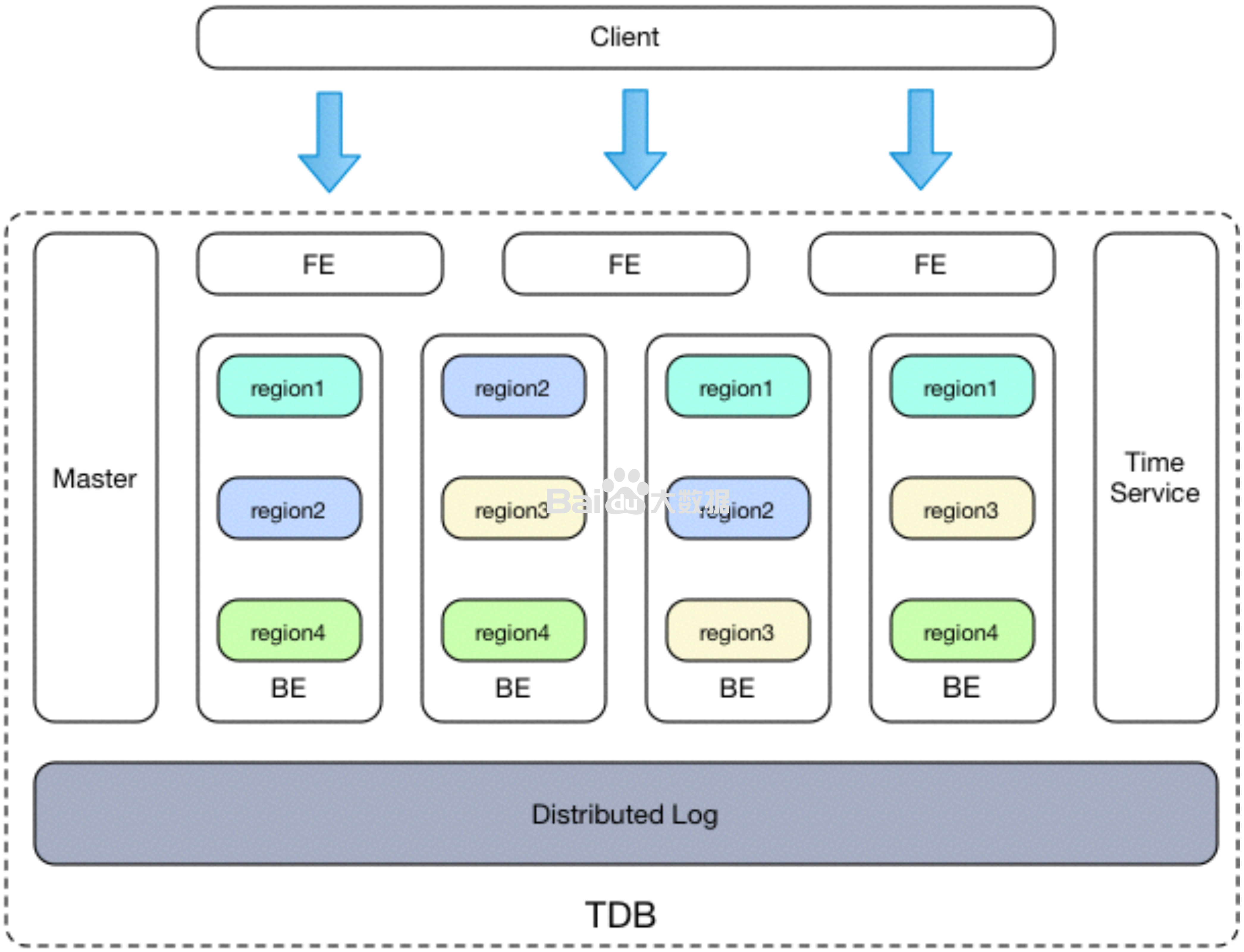
- OB 0.5
  - 分布式事务转成updateserver单机事务



- OB 1.0
  - 基于paxos的副本强一致复制
  - 两阶段提交

TDB

- 较高的可扩展性
- ACID分布式事务
- 强一致
- 二级索引
- Online Schema Change
- SQL接口 & 分布式计算层
- 非事务操作优化
- 异地多活


Client
FE
FE
FE
Master
region1
region2
region4
BE
region2
region3
region4
BE
region1
region2
region3
BE
region1
region3
region4
BE
Time Service
Distributed Log
TDB

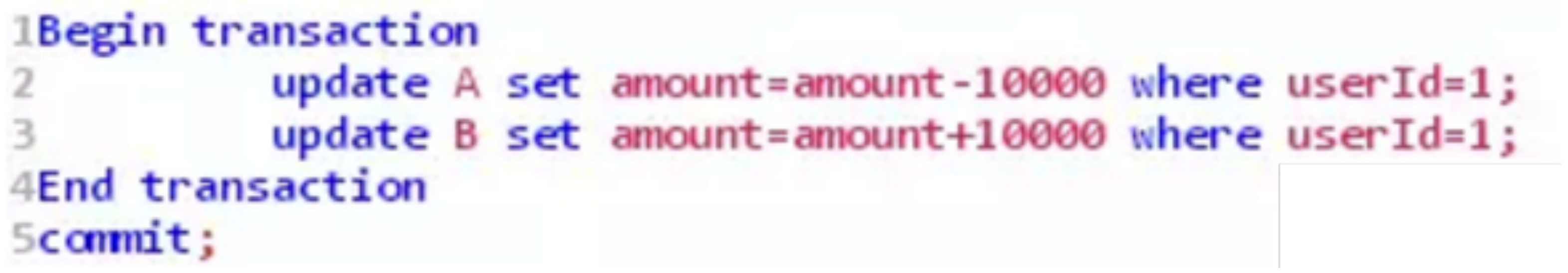
- Master
  - 管理元数据：database、table、shard、replica分布，schema等
- Time Service
  - 混合逻辑时钟，用来确定事务的时序
- Backend(BE)
  - 数据存储节点
- Frontend(FE)
  - 缓存客户端的读写请求
  - 两阶段提交事务协调者
- DistributedLog(DL)
  - 副本一致

可扩展性

- 单机群扩展到100台
- Master、Time Service单点，支撑20w+ TPS
- Master性能通过更好的硬件scale up
    - nvram，提供几十万的iops
    - open power架构，cpu性能比高端x86性能提升一倍
- FE和BE线性扩展
- 多业务，多集群

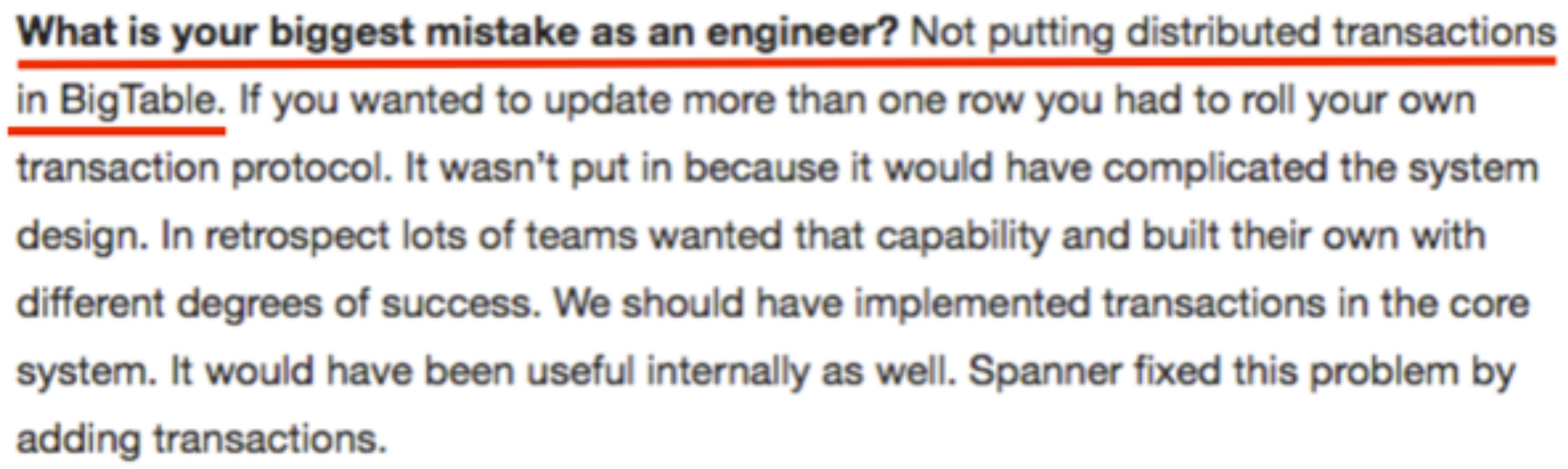
分布式事务

- 支持分布式事务能大大简化应用



```
Begin transaction
        update A set amount=amount-10000 where userId=1;
        update B set amount=amount+10000 where userId=1;
End transaction
commit;
```

- 通过消息队列规避分布式事务，业务需要处理大量复杂场景
    - A表减余额成功和消息发送本身就得有事务保证
    - B表的对应账户如果不存在，怎么回滚A表
    - B表加余额成功和消息删除也得有事务保证
- Google Jeff Dean经验



What is your biggest mistake as an engineer? Not putting distributed transactions in BigTable. If you wanted to update more than one row you had to roll your own transaction protocol. It wasn't put in because it would have complicated the system design. In retrospect lots of teams wanted that capability and built their own with different degrees of success. We should have implemented transactions in the core system. It would have been useful internally as well. Spanner fixed this problem by adding transactions.

ACID

- **Atomicity** is the “all or nothing” guarantee for transactions — either all of a transaction’s actions commit or none do.
- **Consistency** is an application-specific guarantee; SQL integrity constraints are typically used to capture these guarantees in a DBMS. Given a definition of consistency provided by a set of constraints, a transaction can only commit if it leaves the database in a consistent state.
- **Isolation** is a guarantee to application writers that two concurrent transactions will not see each other’s in-flight (not-yet-committed) updates.
- **Durability** is a guarantee that the updates of a committed transaction will be visible in the database to subsequent transactions independent of subsequent hardware or software errors, until such time as they are overwritten by another committed transaction.

事务并发控制模型

- **Strict two-phase locking (2PL)**: Transactions acquire a shared lock on every data record before reading it, and an exclusive lock on every data item before writing it. All locks are held until the end of the transaction, at which time they are all released atomically. A transaction blocks on a wait-queue while waiting to acquire a lock.

- **Multi-Version Concurrency Control (MVCC)**: Transactions do not hold locks, but instead are guaranteed a consistent view of the database state at some time in the past, even if rows have changed since that fixed point in time.

- **Optimistic Concurrency Control (OCC)**: Multiple transactions are allowed to read and update an item without blocking. Instead, transactions maintain histories of their reads and writes, and before committing a transaction checks history for isolation conflicts they may have occurred; if any are found, one of the conflicting transactions is rolled back.

ANSI SQL事务隔离级别

- **READ UNCOMMITTED**: A transaction may read any version of data, committed or not. This is achieved in a locking implementation by read requests proceeding without acquiring any locks.

- **READ COMMITTED**: A transaction may read any committed version of data. Repeated reads of an object may result in different (committed) versions. This is achieved by read requests acquiring a read lock before accessing an object, and unlocking it immediately after access.

- **REPEATABLE READ**: A transaction will read only one version of committed data; once the transaction reads an object, it will always read the same version of that object. This is achieved by read requests acquiring a read lock before accessing an object, and holding the lock until end-of-transaction.

- **SERIALIZABLE**: Full serializable access is guaranteed.

TDB分布式事务

- 基于MVCC的Snapshot Isolation隔离
- 优化的两阶段提交，减少RPC调用次数，降低单事务Latency
- 死锁避免

Snapshot Isolation

- **Snapshot Isolation** : When the transaction starts, it gets a unique start-timestamp from a monotonically increasing counter; when it commits it gets a unique end-timestamp from the counter. The transaction commits only if no other transaction with an overlapping start/end-transaction pair wrote data that this transaction also wrote. This isolation mode depends upon a multi-version concurrency implementation, rather than locking.

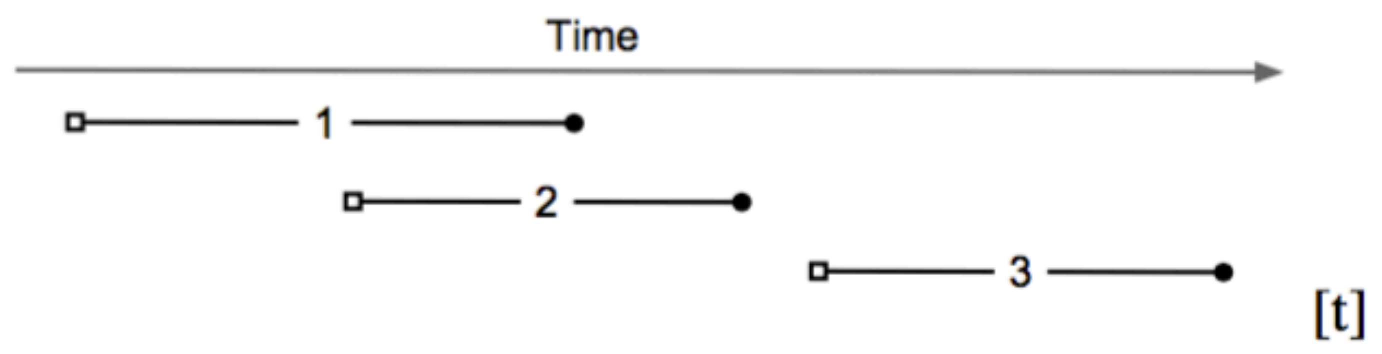


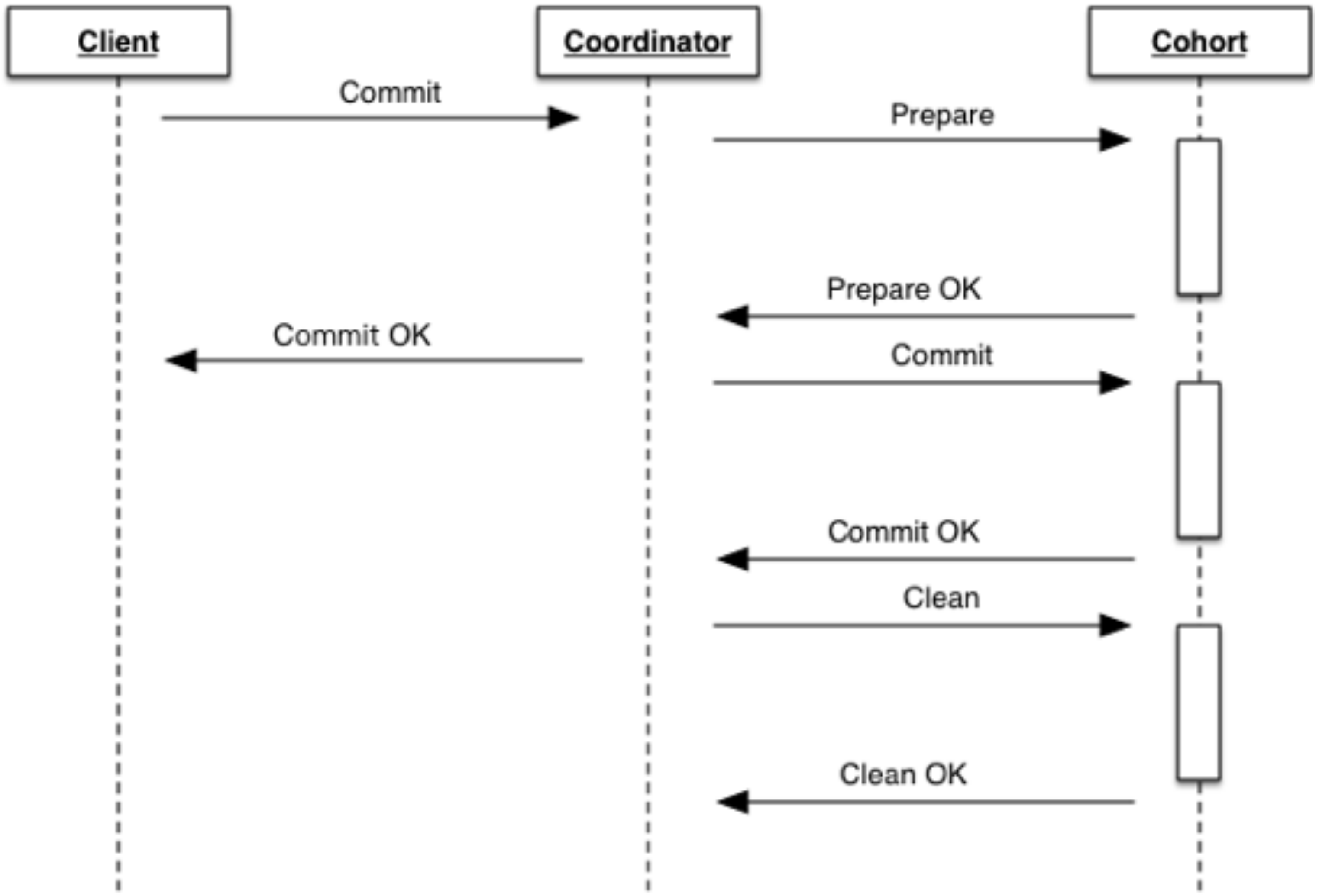
Figure 3: Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2's start timestamp is before transaction 1's commit timestamp. Transaction 3, however, will see writes from both 1 and 2. Transaction 1 and 2 are running concurrently: if they both write the same cell, at least one will abort.

Snapshot Isolation

- 隔离级别仅次于Serializable
- 每个事物都是自己的快照，写不会阻塞读

两阶段提交

- 两阶段提交分布式事务
- 协调者不记录状态，一次RPC



强一致

- 问题 - MySQL半同步复制
  - master先写本地binlog，复制binlog失败，切主后，旧master重启提交pending的binlog
  - 网络分割场景下，当master进行切换时，旧master仍有部分client进行读写
- TDB
  - distributed log
  - lease机制
  - 基于checkpoint的快速恢复

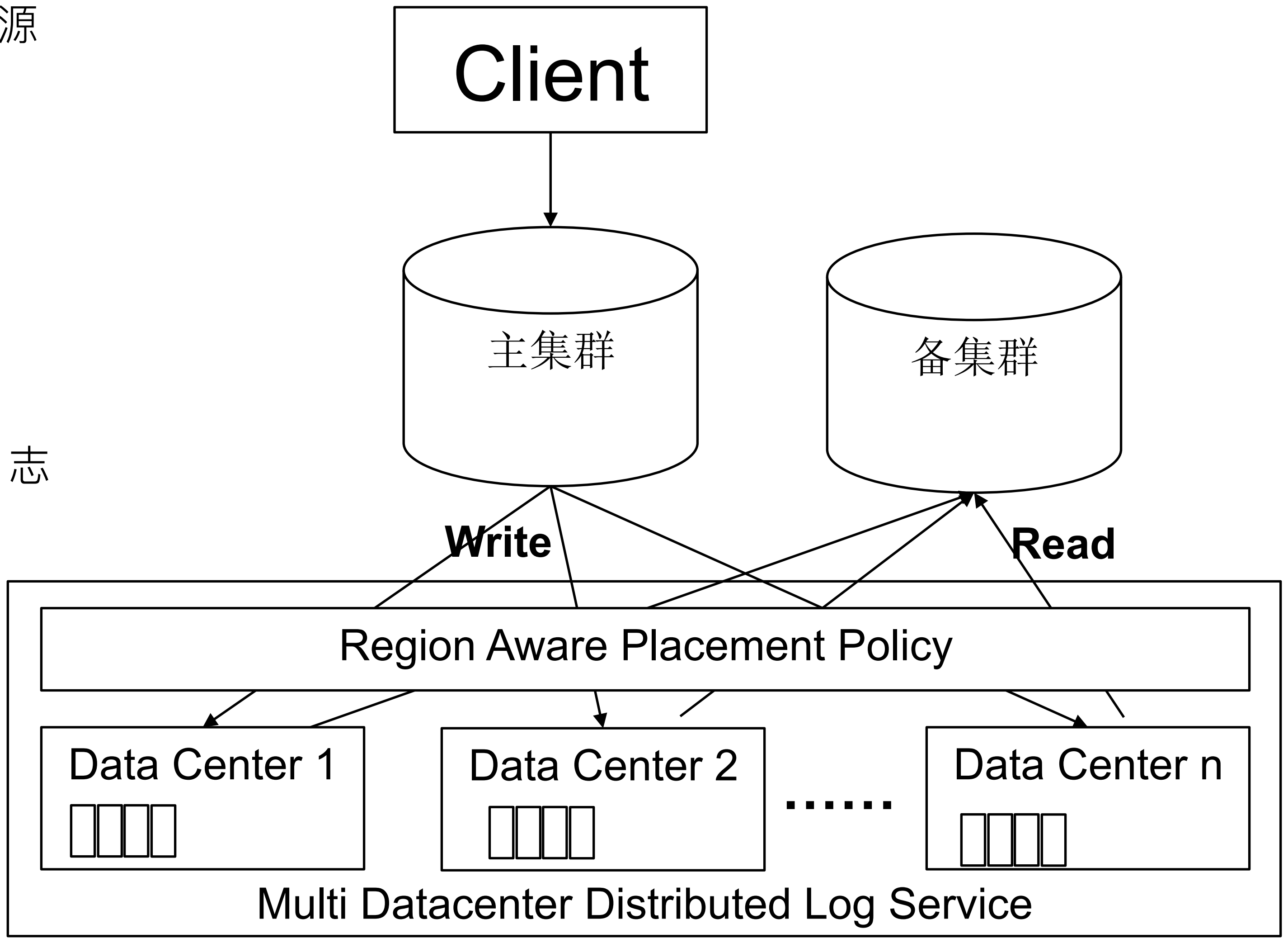
二级索引

- local index
- global index

- sorted index
- full text index (Lucece 索引)

异地多活

- 数据同步需要的log与数据状态分离
  - 分布式的log服务跨机房部署，满足两地三中心的需求
  - 业务数据状态只需要部署两个机房，节省资源
  - 统一机房内 与 跨机房数据同步的方式
- 数据同步
  - 主集群数据通过quorum的方式写入DL
  - 备集群和主集群中的从副本都通过DL回放日志
- 元数据同步
  - 目前两个集群的元数据互相独立
  - 未来把元数据也通过DL管理
- 主备切换：人工完成

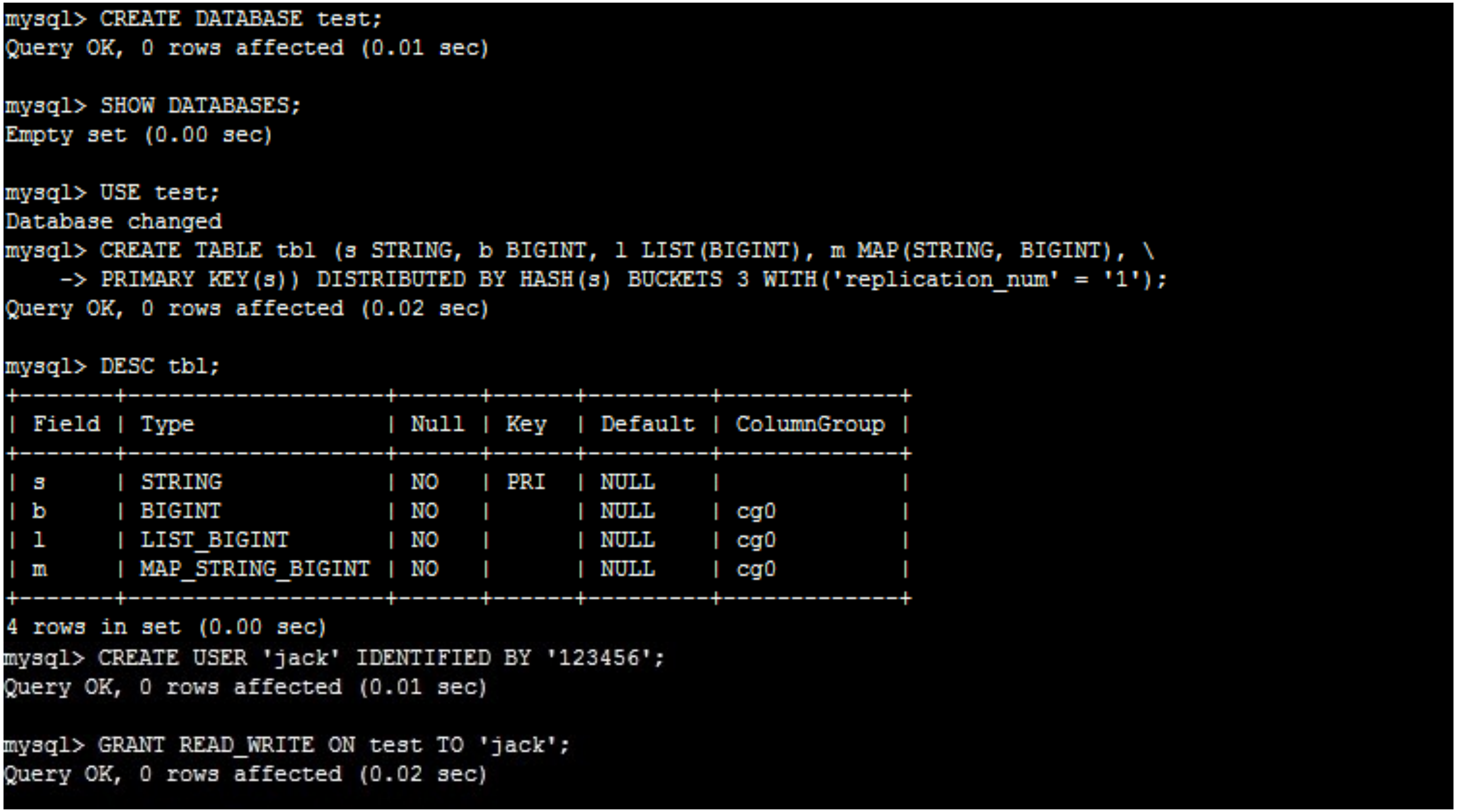

Client
主集群
备集群
Write
Read
Region Aware Placement Policy
Data Center 1
Data Center 2
……
Data Center n
Multi Datacenter Distributed Log Service

Online Schema Change

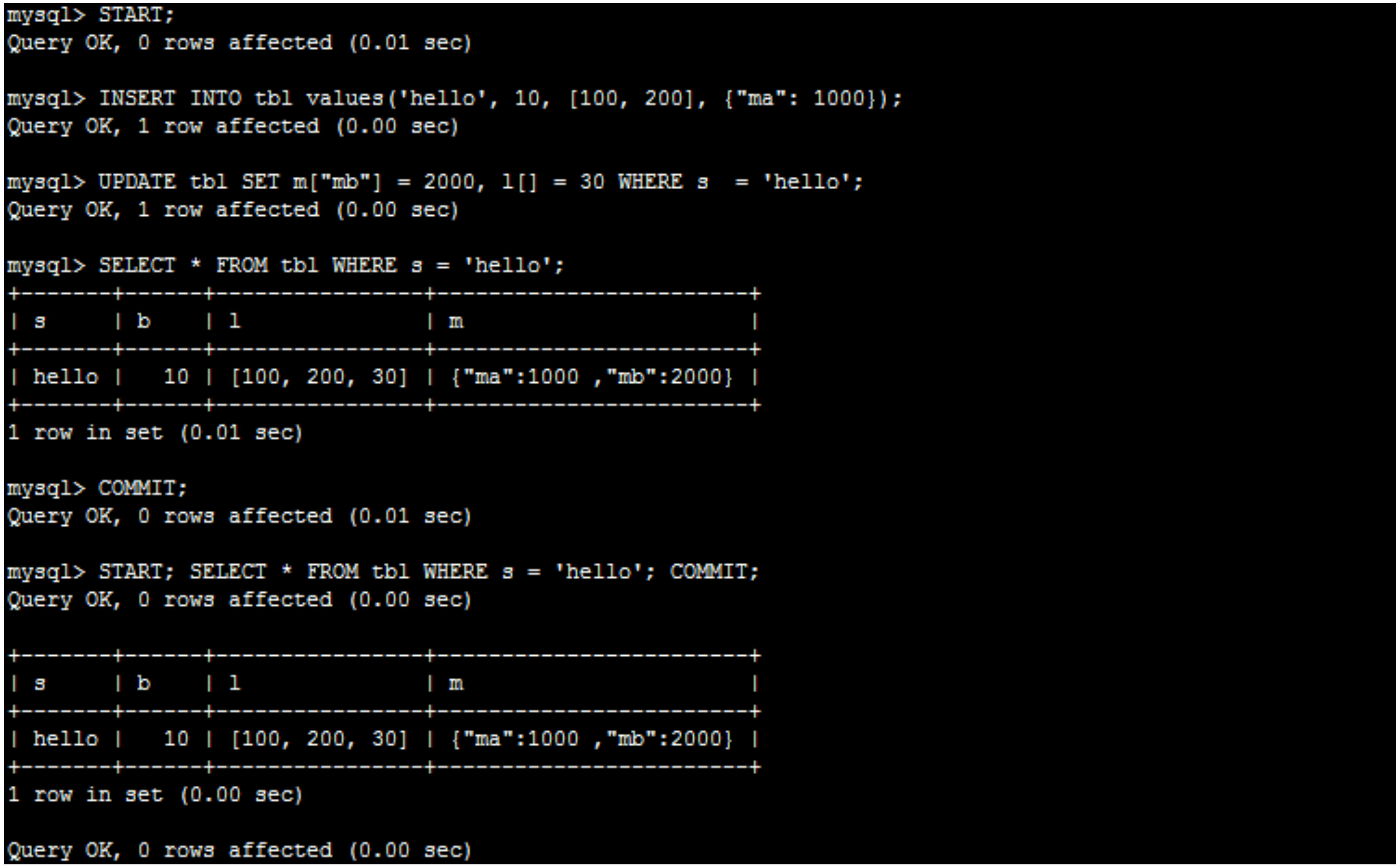
- 问题 - MySQL Schema Change
  - MySQL 5.5之前schema change阻塞服务
  - 在从库上修改表定义，修改之后再提升为新的主库。管理维护复杂，切换中断服务。
  - MySQL 5.6开始提供的InnoDB在线DDL。但是部分操作仍然拷贝表和锁表，从节点会阻塞转换数据造成主从延时较大。
  - Percona公司的pt-online-schema-change和Facebook的OSC。管理维护复杂，切换中断服务。
- TDB
  - 加列、减列
  - 修改列默认值
  - 数据库、表、列增加别名（平滑rename）
  - 创建索引

SQL接口

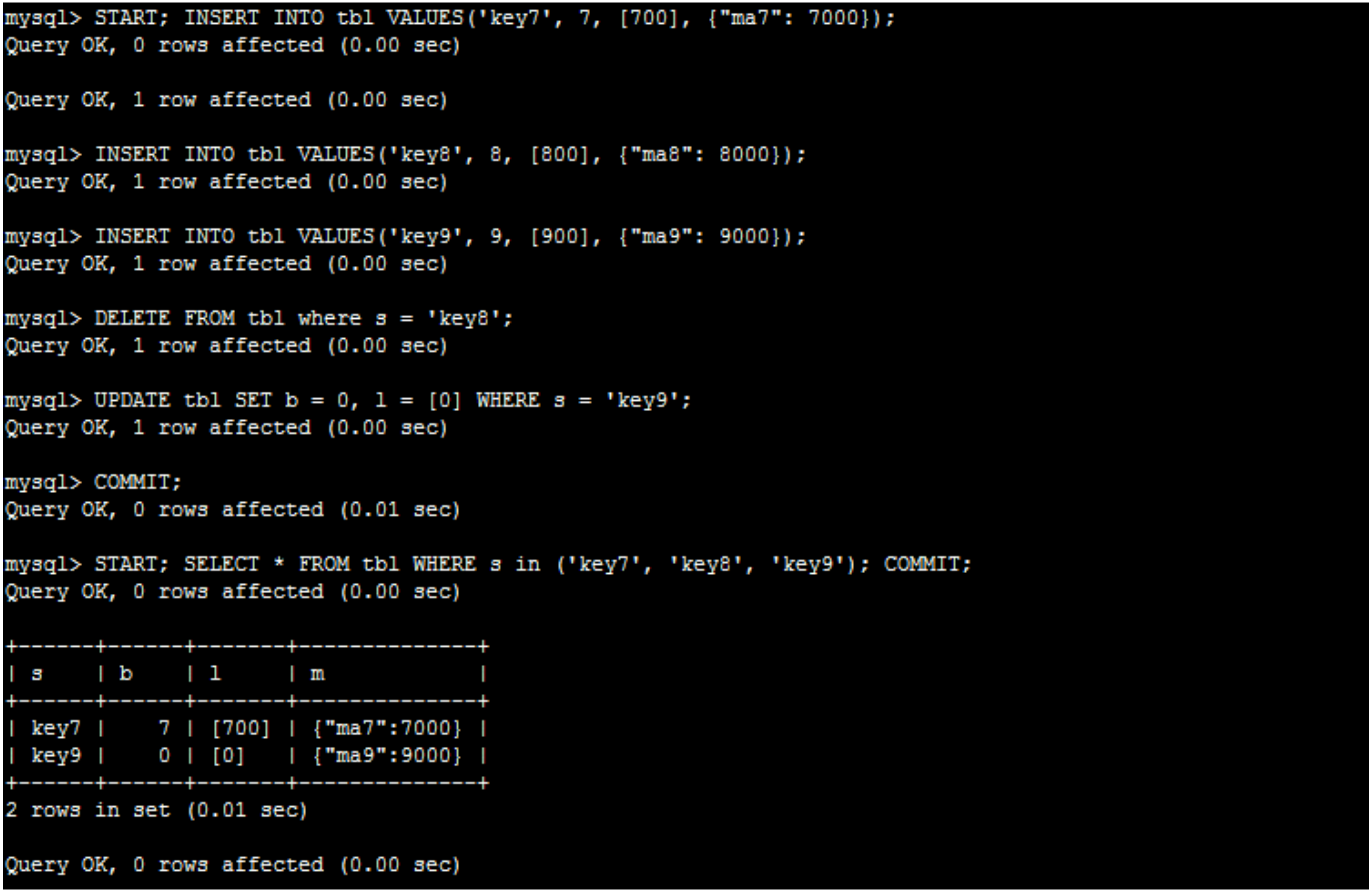
- 基于MySQL协议
- 完善的认证、授权



```
mysql> CREATE DATABASE test;
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW DATABASES;
Empty set (0.00 sec)

mysql> USE test;
Database changed
mysql> CREATE TABLE tbl (s STRING, b BIGINT, l LIST(BIGINT), m MAP(STRING, BIGINT), \
    -> PRIMARY KEY(s)) DISTRIBUTED BY HASH(s) BUCKETS 3 WITH('replication_num' = '1');
Query OK, 0 rows affected (0.02 sec)

mysql> DESC tbl;
+-------+-------------------+------+-----+---------+-------------+
| Field | Type              | Null | Key | Default | ColumnGroup |
+-------+-------------------+------+-----+---------+-------------+
| s     | STRING            | NO   | PRI | NULL    |             |
| b     | BIGINT            | NO   |     | NULL    | cg0         |
| l     | LIST_BIGINT       | NO   |     | NULL    | cg0         |
| m     | MAP_STRING_BIGINT | NO   |     | NULL    | cg0         |
+-------+-------------------+------+-----+---------+-------------+
4 rows in set (0.00 sec)
mysql> CREATE USER 'jack' IDENTIFIED BY '123456';
Query OK, 0 rows affected (0.01 sec)

mysql> GRANT READ_WRITE ON test TO 'jack';
Query OK, 0 rows affected (0.02 sec)
```



```
mysql> START;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO tbl values('hello', 10, [100, 200], {"ma": 1000});
Query OK, 1 row affected (0.00 sec)

mysql> UPDATE tbl SET m["mb"] = 2000, l[] = 30 WHERE s  = 'hello';
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM tbl WHERE s = 'hello';
+-------+------+----------------+-----------------------+
| s     | b    | l              | m                     |
+-------+------+----------------+-----------------------+
| hello |   10 | [100, 200, 30] | {"ma":1000 ,"mb":2000} |
+-------+------+----------------+-----------------------+
1 row in set (0.01 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> START; SELECT * FROM tbl WHERE s = 'hello'; COMMIT;
Query OK, 0 rows affected (0.00 sec)

+-------+------+----------------+-----------------------+
| s     | b    | l              | m                     |
+-------+------+----------------+-----------------------+
| hello |   10 | [100, 200, 30] | {"ma":1000 ,"mb":2000} |
+-------+------+----------------+-----------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```



```
mysql> START; INSERT INTO tbl VALUES('key7', 7, [700], {"ma7": 7000});
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO tbl VALUES('key8', 8, [800], {"ma8": 8000});
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO tbl VALUES('key9', 9, [900], {"ma9": 9000});
Query OK, 1 row affected (0.00 sec)

mysql> DELETE FROM tbl where s = 'key8';
Query OK, 1 row affected (0.00 sec)

mysql> UPDATE tbl SET b = 0, l = [0] WHERE s = 'key9';
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> START; SELECT * FROM tbl WHERE s in ('key7', 'key8', 'key9'); COMMIT;
Query OK, 0 rows affected (0.00 sec)

+------+------+-------+-------------+
| s    | b    | l     | m           |
+------+------+-------+-------------+
| key7 |    7 | [700] | {"ma7":7000} |
| key9 |    0 | [0]   | {"ma9":9000} |
+------+------+-------+-------------+
2 rows in set (0.01 sec)

Query OK, 0 rows affected (0.00 sec)
```

非事务操作

- 事务场景有限，优化非事务操作
- 不需要访问时钟服务器，避免性能单点
- 可以做一些KV / Table的工作

分布式查询层

- 借鉴Palo，实现TDB的分布式查询层
- 可以做一些OLAP分析需求

谢　谢

Baidu大数据