UPSQL:中国银联MySQL实践之路

操作系统与数据库团队 周家晶

2017年12月9日

UPSQL产品介绍

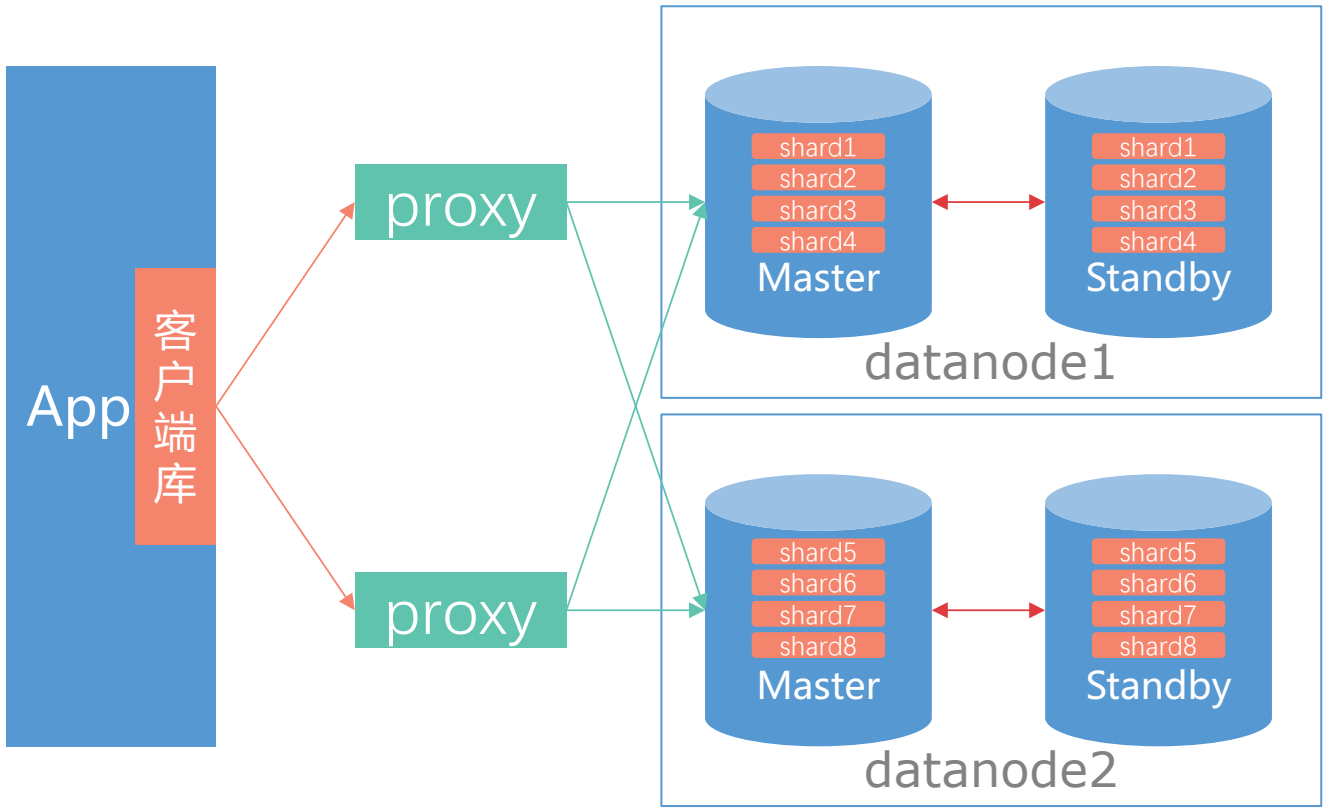
一. 产品组成

二. 发展点滴

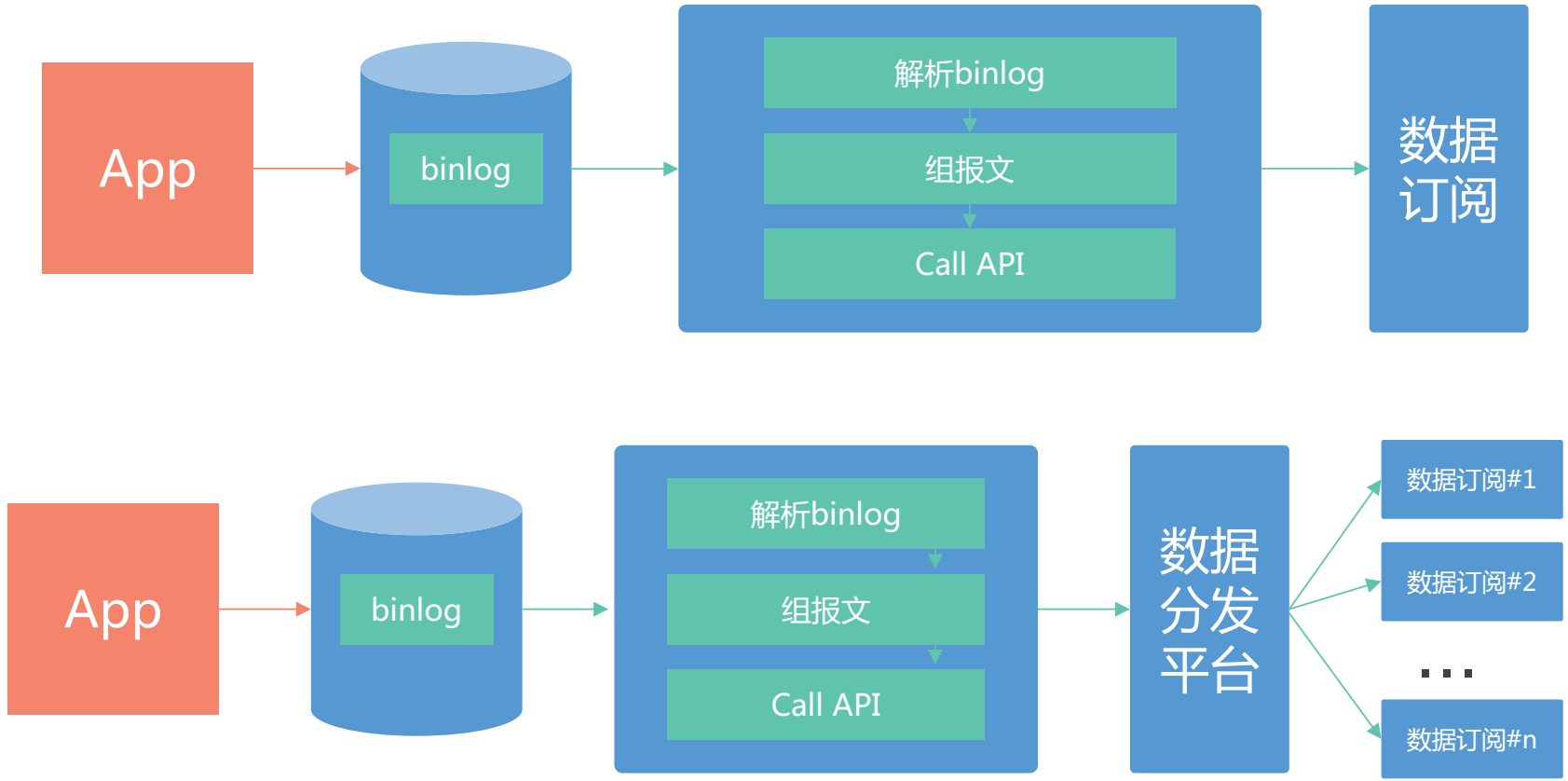
产品组成

- **UPSQL Server: MySQL功能定制版， 当前版本对应MySQL 5.7**
- **UPSQL Proxy: 高可用与数据拆分中间件**
- **UPSQL Mover: 基于binlog的数据分发工具**
- **DBaaS: 数据即服务平台**

产品组成 / UPSQL Proxy



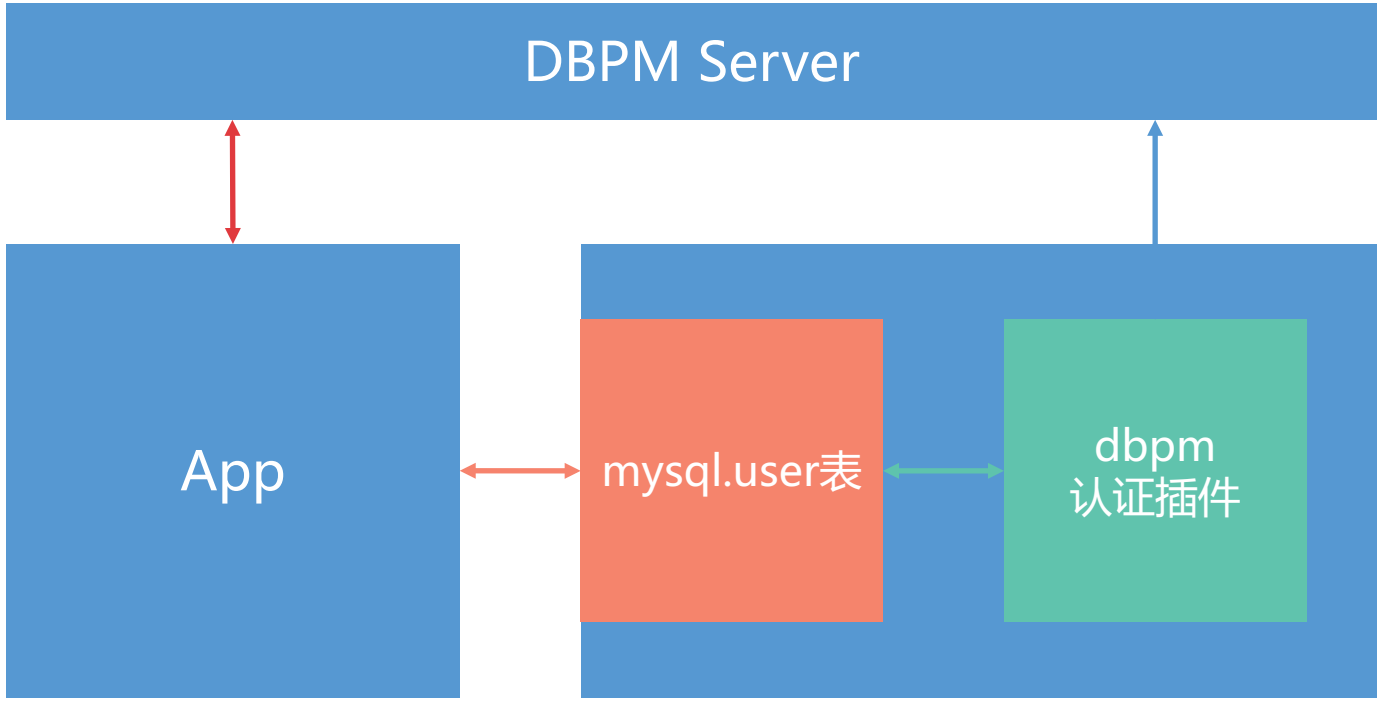
产品组成 / UPSQL Mover



产品组成 / DBaaS

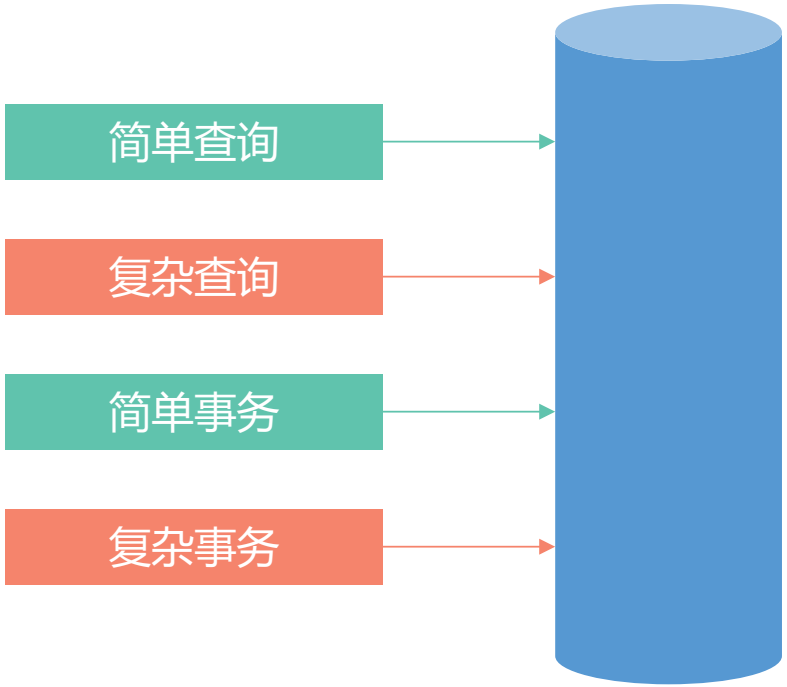
发展点滴 / DBMP

■首个定制功能



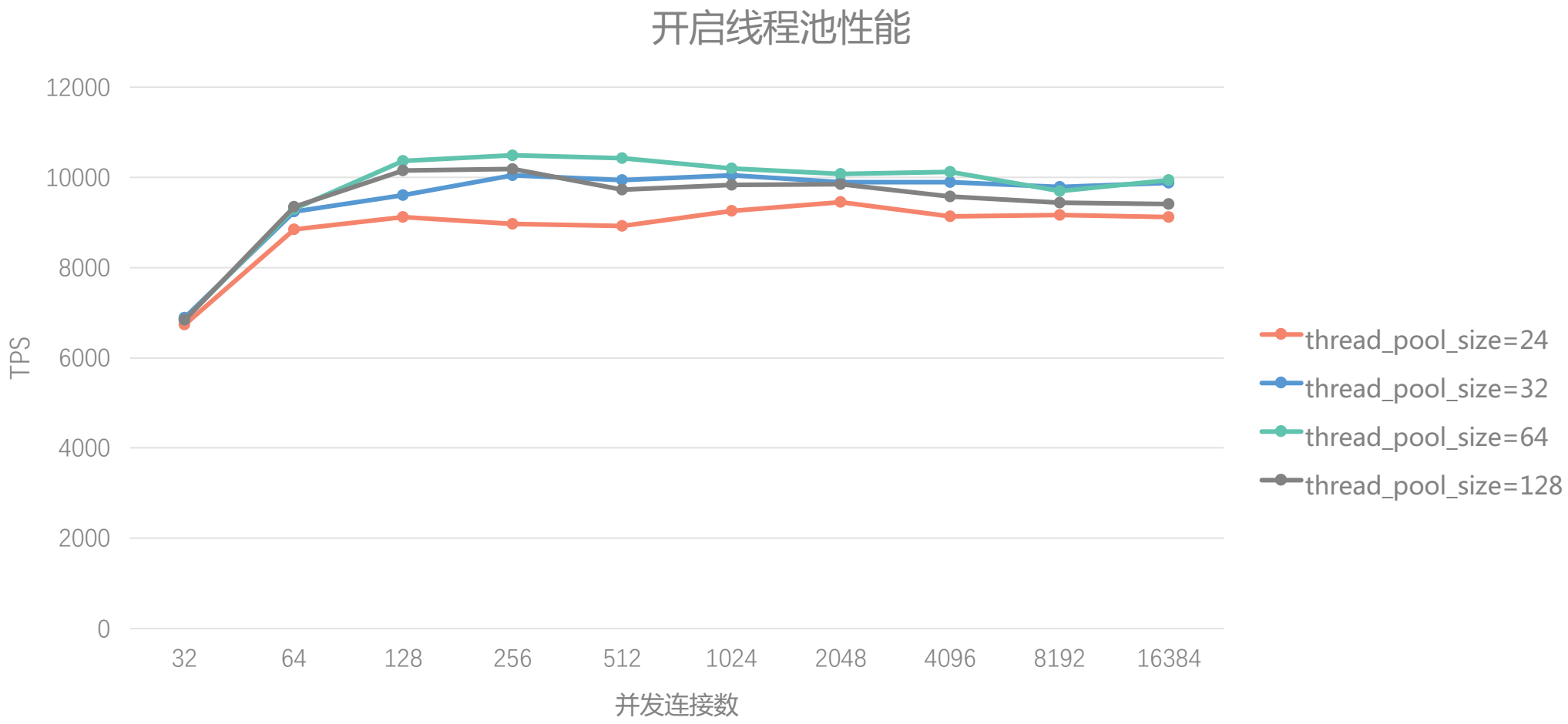
- 应用程序和数据库均不保存密码
- 密码统一存放在DBPM服务器，方便定期修改
- 应用程序和数据库均从DBPM服务器获取密码，由认证插件完成认证

发展点滴 / 多队列并发控制

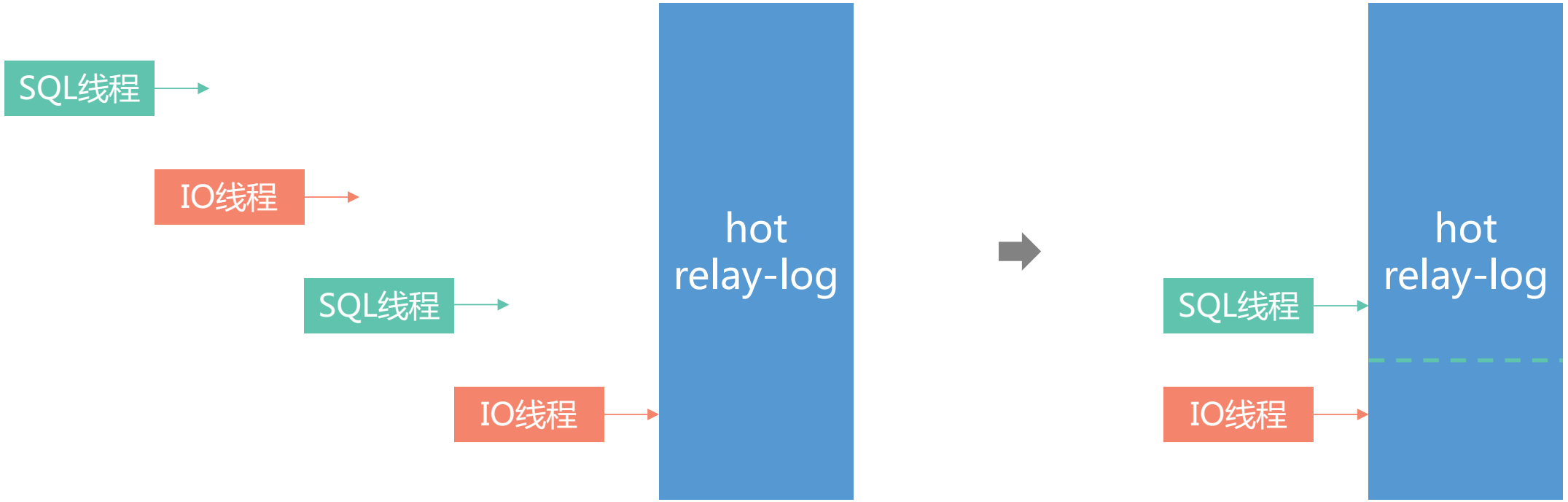


- 对查询请求和事务请求进行分类
- 对不同请求类型设置最大并发数
- 并发数超过阈值进入等待队列
- 可设置等待超时时间
- 支持不受限用户白名单

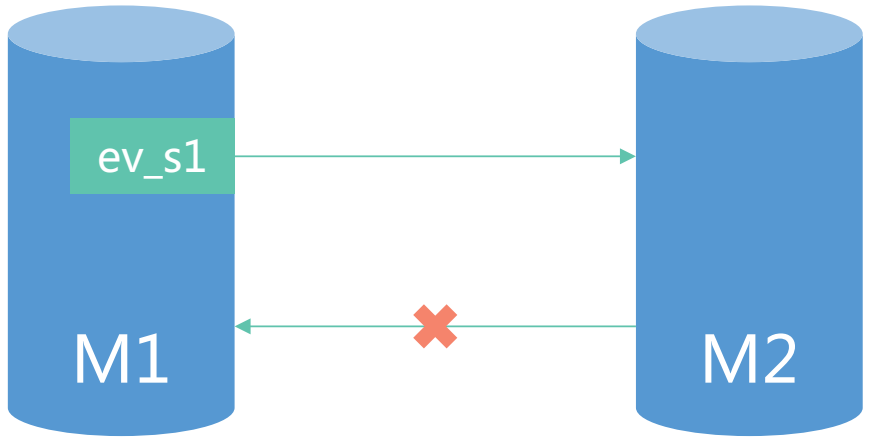
发展点滴 / 线程池



发展点滴 / relay-log锁优化



发展点滴 / 主主日志回传优化



- 优化前，ev_s1需回传至M1，然后被IO线程丢弃
- 优化后，在M2上即完成日志过滤，避免不必要的回传，节约带宽
- 在主主半同步复制场景减少一次ACK等待
- M2日志位点更新通过心跳事件通知M1

发展点滴 / 其他功能开发

01 复制容错处理

02 多源复制支持半同步插件

03 热点数据更新自动提交

04 无效链接自动清理

05 表碎片统计和自动整理

06 动态增加字段

07 Binlog闪回工具

08 国密SM3

09 ……

技术实现

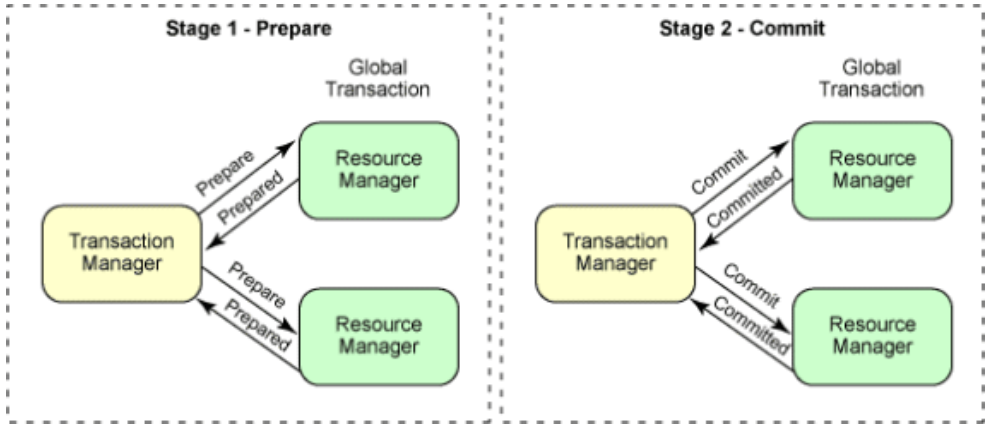
一. 分布式事务

二. SQL解析优化

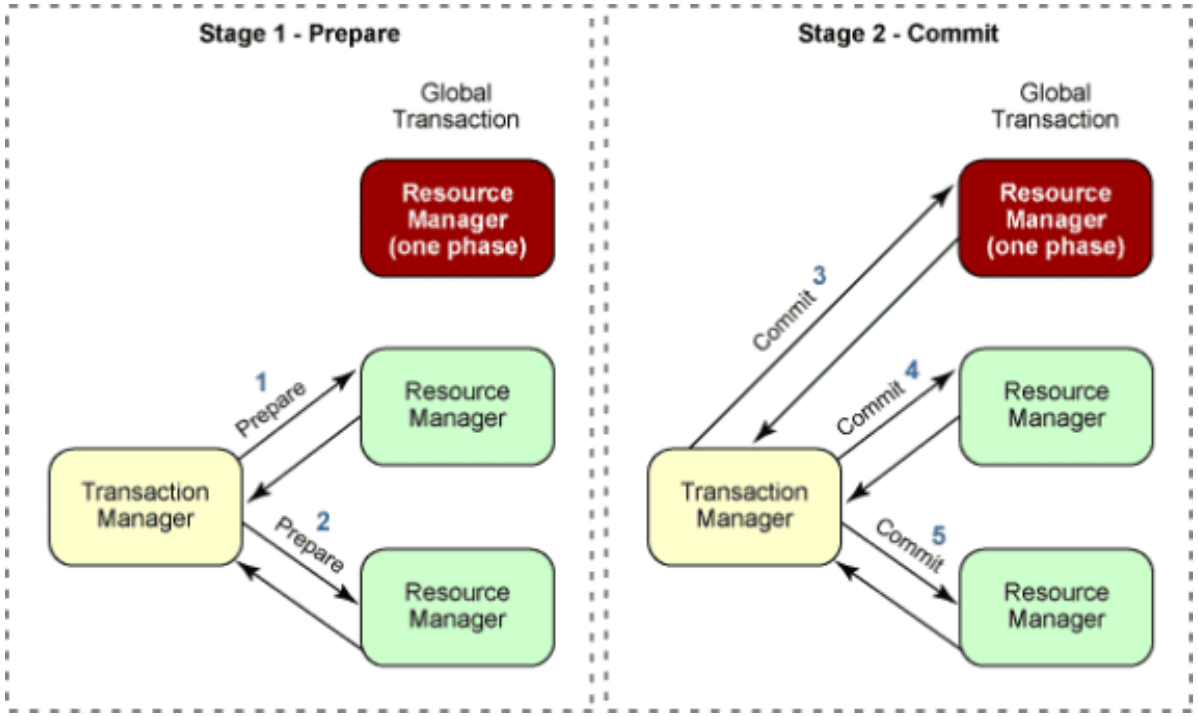
分布式事务

■核心思路

- □使用后端数据库存储XA日志
- □两阶段的实践优化：
  - 最后参与者策略
  - 一阶段提交优化



两阶段



最后参与者策略

分布式事务 / 实现对比

		友商方案			银联方案	
Client		Set_1	Set_2		Datanode_1	Datanode_2
begin;						
insert into t1 values(1);		xa start 'xa-gtid-1';			xa start 'xa-gtid-1';	
		insert into t1 values(1);			insert into t1 values(1);	
insert into t1 values(2);			xa start 'xa-gtid-1';			xa start 'xa-gtid-1';
			insert into t1 values(1);			insert into t1 values(1);
commit;		xa end 'xa-gtid-1';	xa end 'xa-gtid-1';		insert into xa.commit_log...;	xa end 'xa-gtid-1';
		xa prepare 'xa-gtid-1';	xa prepare 'xa-gtid-1';		xa end 'xa-gtid-1';	xa prepare 'xa-gtid-1';
		insert into xa.commit_log...;			xa commit 'xa-gtid-1' one phase;	
		xa commit 'xa-gtid-1';	xa commit 'xa-gtid-1';			xa commit 'xa-gtid-1';

■ 主要区别

- 最后参与者退化为一阶段提交，并负责XA日志记录。

■ 主要优势

- 不需要使用单独会话进行xa日志记录 （友商方案中向xa日志表的写入是自动提交的另一个会话）
- 减少一个两阶段事务

■ 主要缺点

- 涉及分布式事务的后端用户，都需要配置xa日志表权限，因而存在运维风险

分布式事务 / 死锁

■**分布式死锁：**

□场景（互斥、请求与保持、不剥夺、循环等待）：

- a & b 开启分布式事务
- time 1: a　write db1.resource1
- time 2: b　write db2.resource2
- time 3: a　write db2.resource2 -> 锁：a 等待b的 db2.resource2
- time 4: b　write db1.resource1 -> 锁：b 等待a的 db1.resource1

□ db1和db2上的本地写锁，在分布式事务场景下，出现了循环等待，成为了分布式死锁

■**死锁的两个解决路径：**

□死锁检测：剥夺

□死锁预防：避免循环等待

分布式事务 / 死锁检测

- **死锁检测的核心是获取锁信息，但MySQL没有提供XAID与锁的关联信息。**
- **解决方案：**
  - 扩展innodb_trx表，增加XAID信息，与innodb_locks关联即可进行分布式死锁检测

```
mysql> desc innodb_trx;
+---------------------------+---------------------+------+-----+---------------------+-------+
| Field                     | Type                | Null | Key | Default             | Extra |
+---------------------------+---------------------+------+-----+---------------------+———+
… …
| trx_xa_format_id          | int(10)             | NO   |     | 0                   |       |
| trx_xa_gtrid_length       | int(2)              | NO   |     | 0                   |       |
| trx_xa_bqual_length       | int(2)              | NO   |     | 0                   |       |
| trx_xa_data               | tinyblob            | NO   |     | NULL                |       |
+---------------------------+---------------------+------+-----+---------------------+-------+
28 rows in set (0.01 sec)
```

分布式事务 / 死锁预防

- **分布式死锁预防策略：保证单一时序的锁等待(只接受新等旧，或旧等新)**
- **实现方案如下：**
  - 带有时间戳的XAID生成规则：
    - 事务开始时间 + TM(proxy)事务序号 + TM编号 + datanode + ...
  - 修改MySQL死锁检测策略，增加分布式死锁预防策略
    - 在DeadlockChecker::search() 中获取xaid，根据死锁预防策略进行处理
- **待改进：分布式死锁预防复用了死锁检测报错，上层无法区分**
  - ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

分布式事务 / 进一步优化

	当前方案		改进方案	
Client	Datanode_1	Datanode_2	Datanode_1	Datanode_2
begin;				
insert into t1 values(1);	xa start 'xa-gtid-1';		xa start 'xa-gtid-1';	
	insert into t1 values(1);		insert into t1 values(1);	
insert into t1 values(2);		xa start 'xa-gtid-1';		xa start 'xa-gtid-1';
		insert into t1 values(1);		insert into t1 values(1);
commit;	insert into xa.commit_log...;	xa end 'xa-gtid-1';	insert into xa.commit_log...;	
	xa end 'xa-gtid-1';	xa prepare 'xa-gtid-1';		xa prepare 'xa-gtid-1';
	xa commit 'xa-gtid-1' one phase;		xa commit 'xa-gtid-1' one phase;	
		xa commit 'xa-gtid-1';		xa commit 'xa-gtid-1';

- **最后参与者退化为一阶段提交，但仍然有多一次xa end操作，可将该步骤省略。**
- **实现方法**
  - 修改bool Sql_cmd_xa_commit::trans_xa_commit(THD *thd)
  - 增加预处理操作：当m_xa_opt == XA_ONE_PHASE时，将事务状态XA_ACTIVE直接修改为XA_IDLE
- **同样的，我们也可以将xa prepare前的xa end省略掉。**

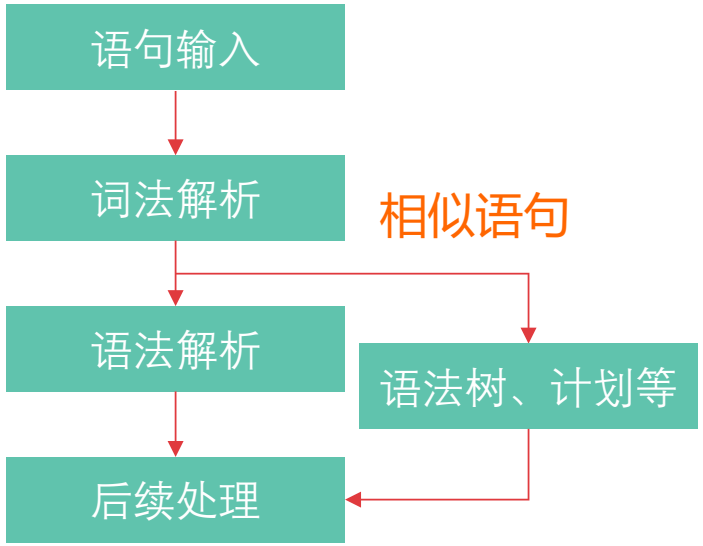
SQL解析优化

■为避免硬解析，业界已提供了两种方法：

- □Prepare：做一次语句解析后，使用语句编号与变量值进行交互
- □软解析：相同的语句复用语法解析树、执行计划等。

SQL解析优化 / 相似性

- 相似性解析优化：在词法解析后进行相似性分析来复用语法解析树、逻辑执行计划等。
- 如何进行相似性分析？
- 如何进行复用？



相似性解析优化

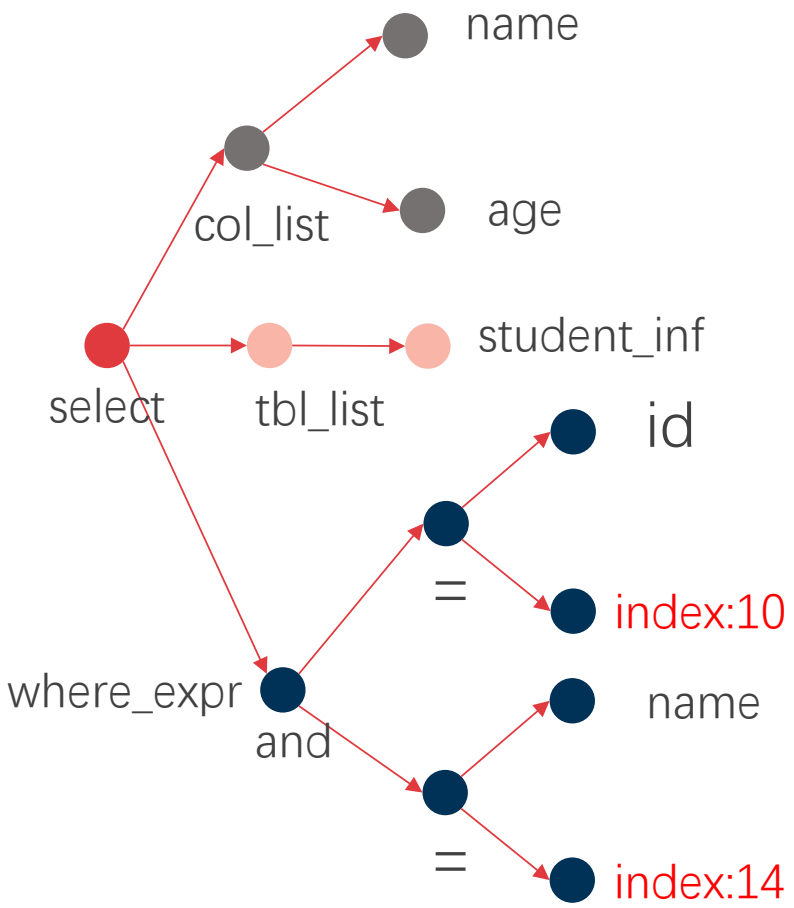
SQL解析优化 / 语句示例

1

Select name, age from student_inf
where id =1 and name = ”dabao”
~

2

Select name, age FROM student_inf
where id =2 and name = “huahua”
~~~


~~~

SQL解析优化 / 相似性分析

■词法分析的结果：

- 一组单词序列
- 每个单词由类型和值组成

■词法解析流程调整：

- MySQL的SQL解析过程，词法和语法解析耦合在一起，即语法解析的移进规约动作触发词法解析，而不是先完成词法解析，然后开始语法解析
- 修改为：先完成词法解析获取单词序列，然后进行语法解析

■相似性规则：

- 对比2个SQL的单词序列，单词一一对比满足下列2个条件即说明语句相似：
  - 单词类型相同
  - 如果类型非参数，则要求单词的值相等(忽略大小写)
- 同样的我们根据上述2个约束，设计了一个相似性hash算法，用于提升相似性查找性能

SQL解析优化 / 相似性复用

- **LEX_STRING是词法分析和语法解析的最基础数据， 增加2个字段：**
  - index:其在词法序列中的位置
  - next_lex_string:需要合并的单词
- **新语句，通过相似语句的语法树、逻辑执行计划和当前词法序列，即可以进行相应计算操作，实现复用。**

```
struct st_mysql_lex_string
{
  char *str;
  size_t length;

  int index; /* 在词法分析结果内，单词序列的位置 */
  struct st_mysql_lex_string *next_lex_string; /* 需要合并的单词 */
};
```

SQL解析优化 / 使用

- **应用在UPSQL Proxy，提升了约66%的语句解析性能和44%综合性能，达到Prepare模式72%性能水准**
- **该方案暂未引入UPSQL Server(牵涉复杂)**
- **单就相似性查找而言：还有一种方案是利用词法分析对语句做值替换(为?)，规整化大小写和空白，然后用调整后的语句进行相似性查找（Statement Digests）**

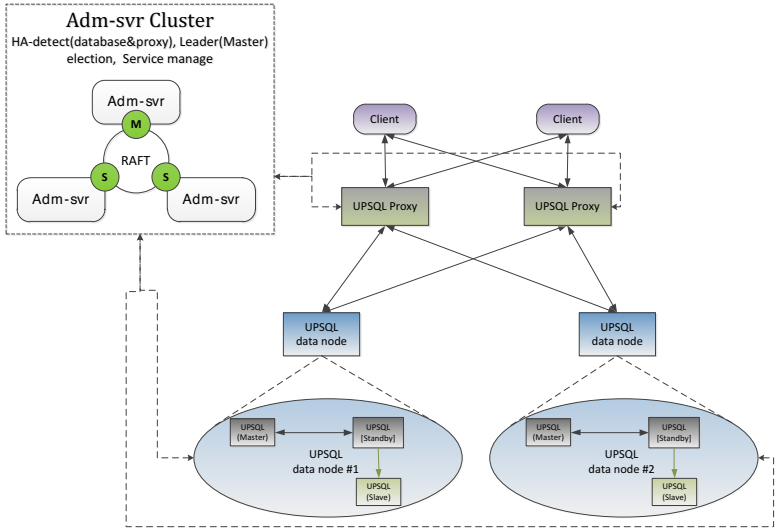
分布式数据库

一. 现状

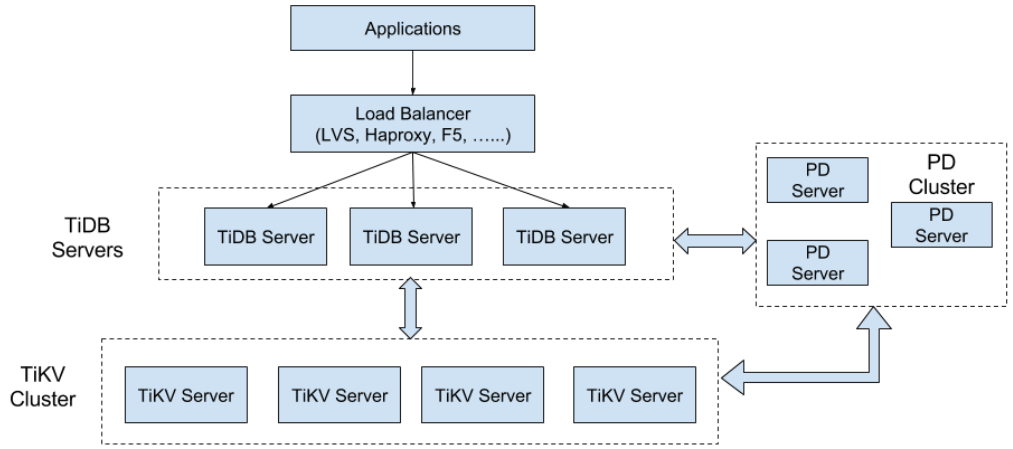
二. 预期

分布式数据库 / 现状

- **MySQL深刻影响着分布式数据库的实现标准**
- **Proxy方案与Spnner方案对比**
  - 调度层Proxy比Spnner轻
  - 存储层Proxy比Spnner重



UPSQL Proxy



TiDB

分布式数据库 / 预期

- **MySQL在分布式数据库领域的潜力未被充分挖掘**
- **MySQL Spider方案依然有较强的生命力，但Spider的底层调用为同步调用，其性能和资源消耗都存在较大劣势，可以与Proxy异步方案相结合**
- **MGR已经实现了Paxos算法，但在跨中心和异地环境下表现不佳，存在优化空间**

谢谢

中国银联的UPSQL实践之路，也是与业界的合作历程，包含不限于：技术咨询、研发培训、DBaaS合作研发、运维保障等。

UPSQL的后续工作、以及分布式数据库选型等方面，也会继续与各方保持合作共赢。