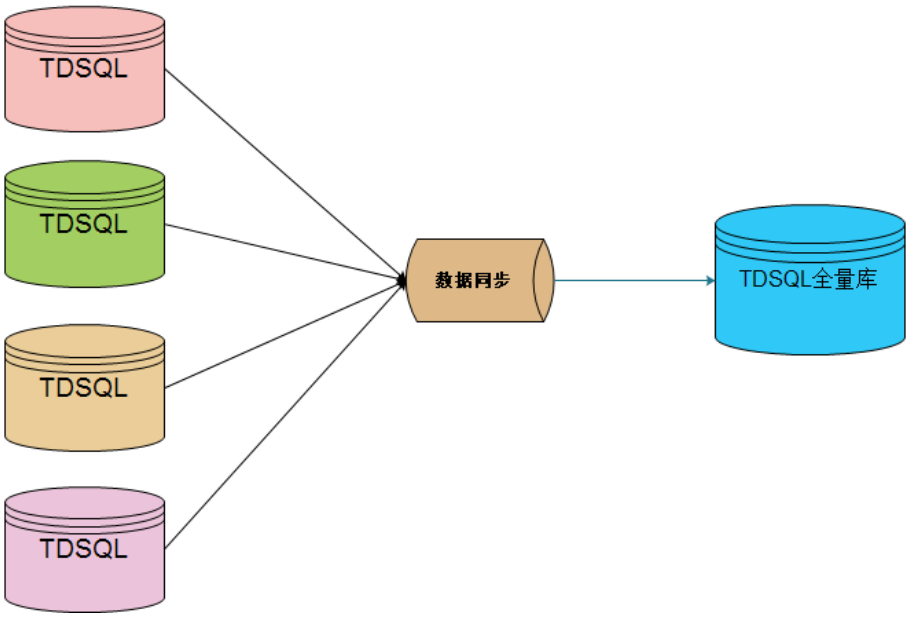
TDSQL多源同步架构与实现详解

一、场景及需求

分布式数据库TDSQL(Tencent distribute database)是腾讯打造的一款分布式数据库产品，具备强一致、高可用、全球部署架构、分布式水平扩展、高性能、企业级安全等特性，同时提供智能DBA、自动化运营、监控告警等配套设施，受到了众多金融企业的青睐。在金融业务场景中，数据的同步，订阅，分发是常见需求，例如保险行业常见的总分系统架构，多个子库需要实时的将业务数据同步至总库汇总查询；银行核心交易系统中，需要将交易数据实时同步至分析子系统进行报表，跑批等业务员操作。因此，作为一个金融级数据库产品来说，数据的分发，解耦能力是必不可少的。



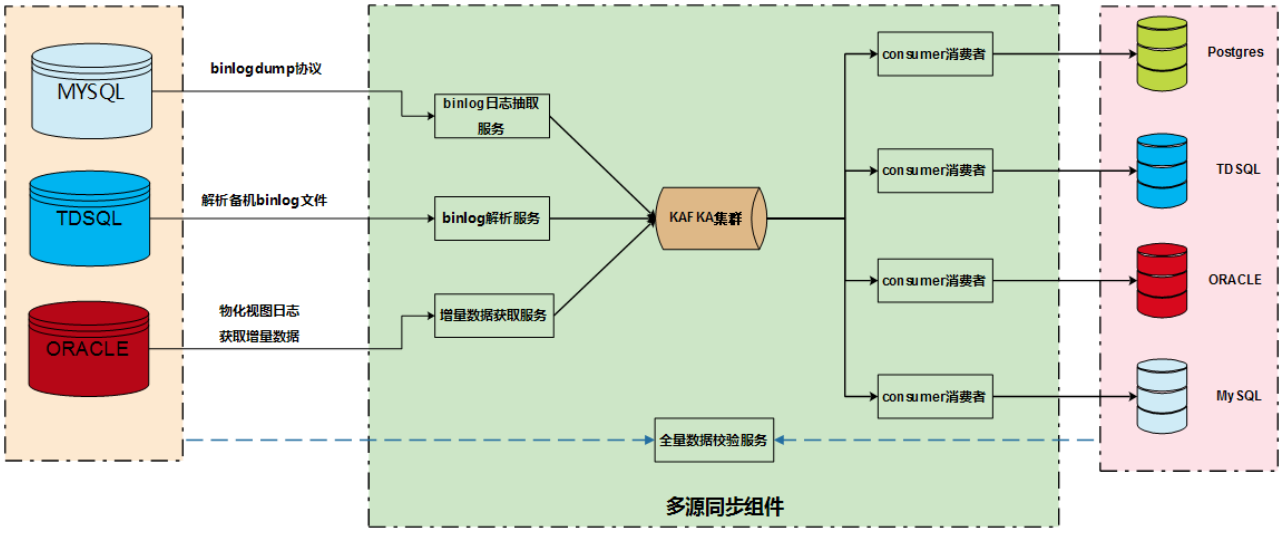
基于总分的数据汇总架构

数据实时同步分析架构

TDSQL-MULTISRCSYNC（下文简称多源同步）模块正是为了应对这样的需求所开发的高性能，高一致，支持多种异构数据平台的数据分发服务。该支持支持以TDSQL作为源端的数据实时同步分发至MySQL,ORACLE,POSTGRES,KAFKA等平台，同时也支持以TDSQL作为目标端，将mysql或者oracle的数据实时同步至TDSQL中,并且部署灵活，支持一对多，多对一等多种复制拓扑结构。

二、系统架构

多源同步模块典型的基于日志的DCD复制技术



从上图我们可以看到，整个系统可以大致的分成三个部分：producter，store，consumer。

producter：增量日志获取模块，主要负责解析获取源端的增量数据改动日志，并将获取到的日志解析封装为JSON协议的消息体，投送至kafka消息队列。当源端是MySQL或者TDSQL时，获取的增量日志为binlog事件，这里要求binlog必须是row格式且为full-image。当源端是ORACLE，producter从oracle的物化视图日志中获取增量数据并进行封装和投送。这里producter在向kafka生产消息时，采用at-least-once模式，即保证特定消息队列中至少有一份，不排除在队列中有消息重复的情况。

store:这里采用kafka作为中间存储队列，因为数据库系统日志有顺序性要求，因此这里所有的topic的partition个数均为1，保证能够按序消费。

consumer：日志消费和重放模块，负责从kafka中将CDC消息消费出来并根据配置重放到目标实例上。这里因为producter端采用at-least-once模式生产，因此消费者这里实现了幂等逻辑保证数据重放的正确。

三、核心特性

3.1、基于行的哈希并发策略

金融业务场景中，往往对数据的实时性要较高，因此对数据同步的性能提出了比较高的要求。为了解决这样的问题，consumer采用了基于行的哈希并发策略实现并行重放。下面以binlog消息为例来说明该策略的实现。

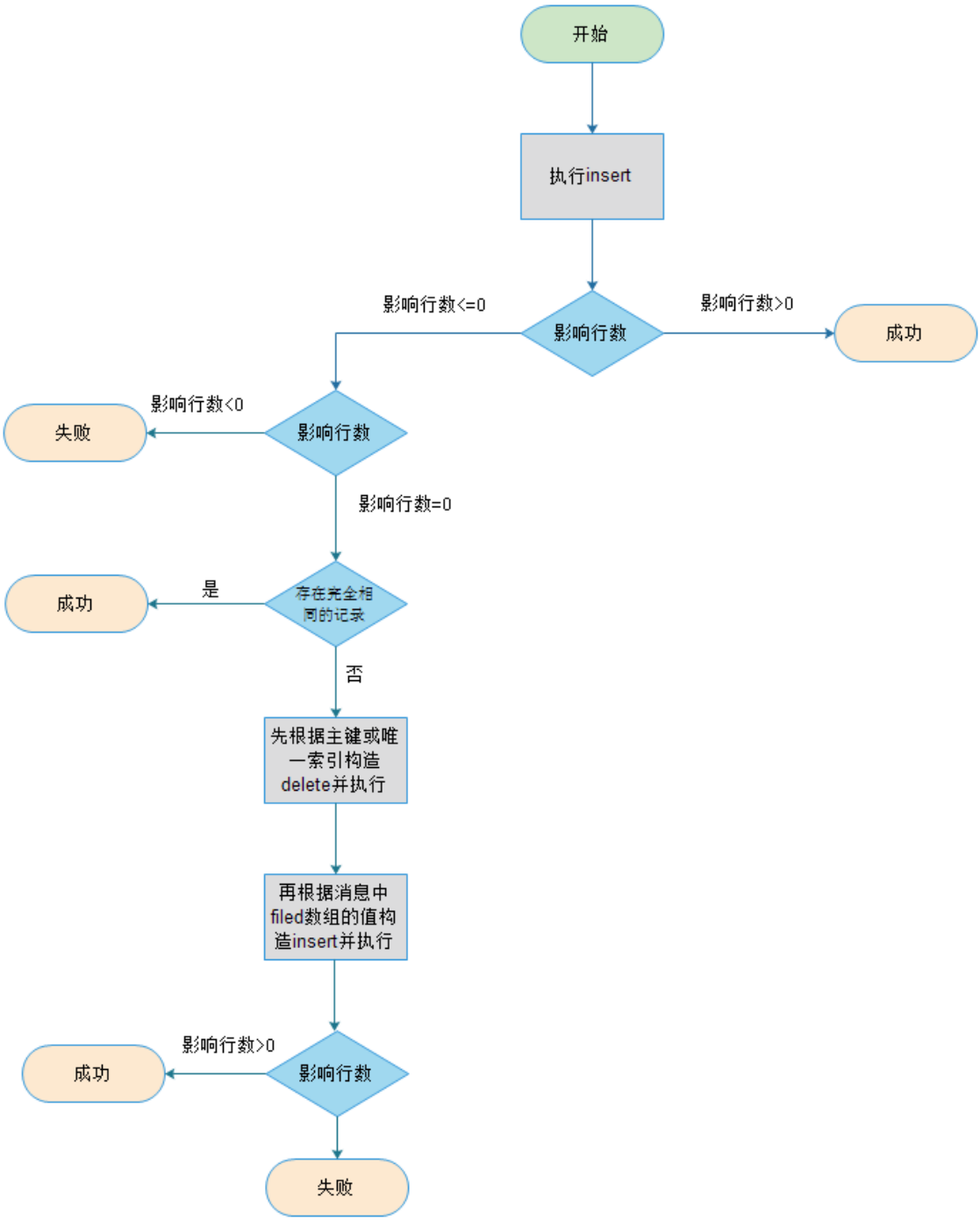
MySQL在记录binlog时，按照事务的提交顺序将行的改动写入binlog文件，因此按照binlog文件记录事件的顺序进行串行重放，源端和目标端数据库实例状态一定会达到一致。但是串行重放因为速度慢，在遇到如批量更新等大事务时，容易产生较大的同步时延，适应不了对数据实时性较高的同步场景。为了提高并发度，这里consumer按照每个行记录的表名和主键值进行hash，根据hash值将消息投送到对应的同步线程中。这样乱序的重放会导致数据不一致吗？答案是不会的，因为虽然是将顺序的消息序列打乱了，但是同一行的所有操作都是在同一个线程中是有序的，因此只要每个行的改动执行序列正确，最终数据是会一致。这个过程如下图所示

3.2、row格式binlog事件的幂等容错

实现幂等逻辑的动机有两个：1、因为生产者实现的是at-lease-once模式进行消息生产，因此consumer这里必须要能否处理消息重复的问题。2、支持幂等逻辑后，便于数据的修复，且在数据同步的过程中不需要记录镜像点，便于运维。这里幂等逻辑的设计原[illegible]og事件的意图去对目标实例进行修改。如insert事件，其意图就是要在数据库中有一条new值标识的记录，update事件的意图就是，数据库中没有old值标识的记录，只有new值标识的记录；delete操作也是同样，其结果就是要求目标数据库中，不包含old值标

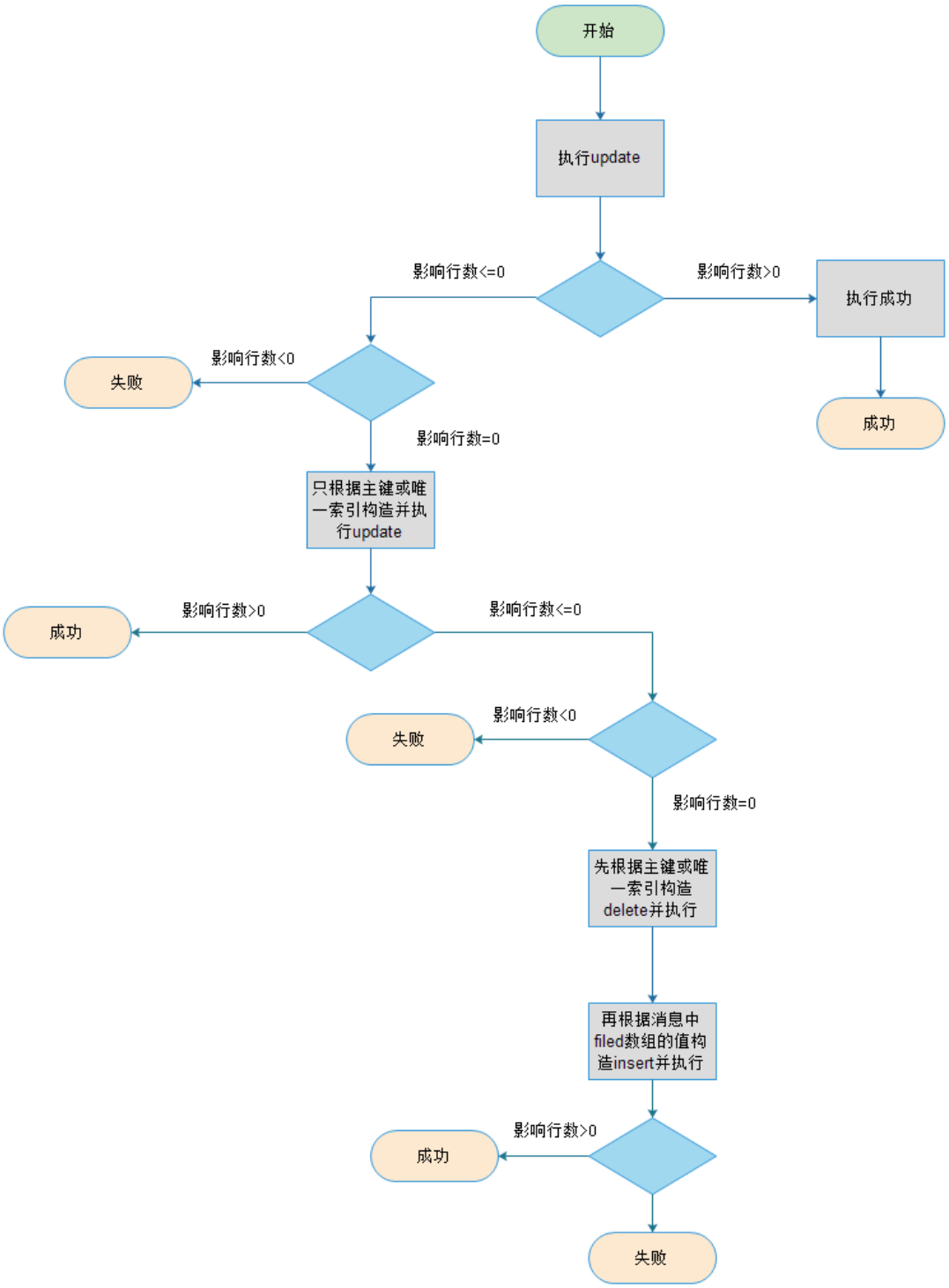
识的记录。因此针对insert，update，delete操作，其幂等逻辑如下。

- INSERT



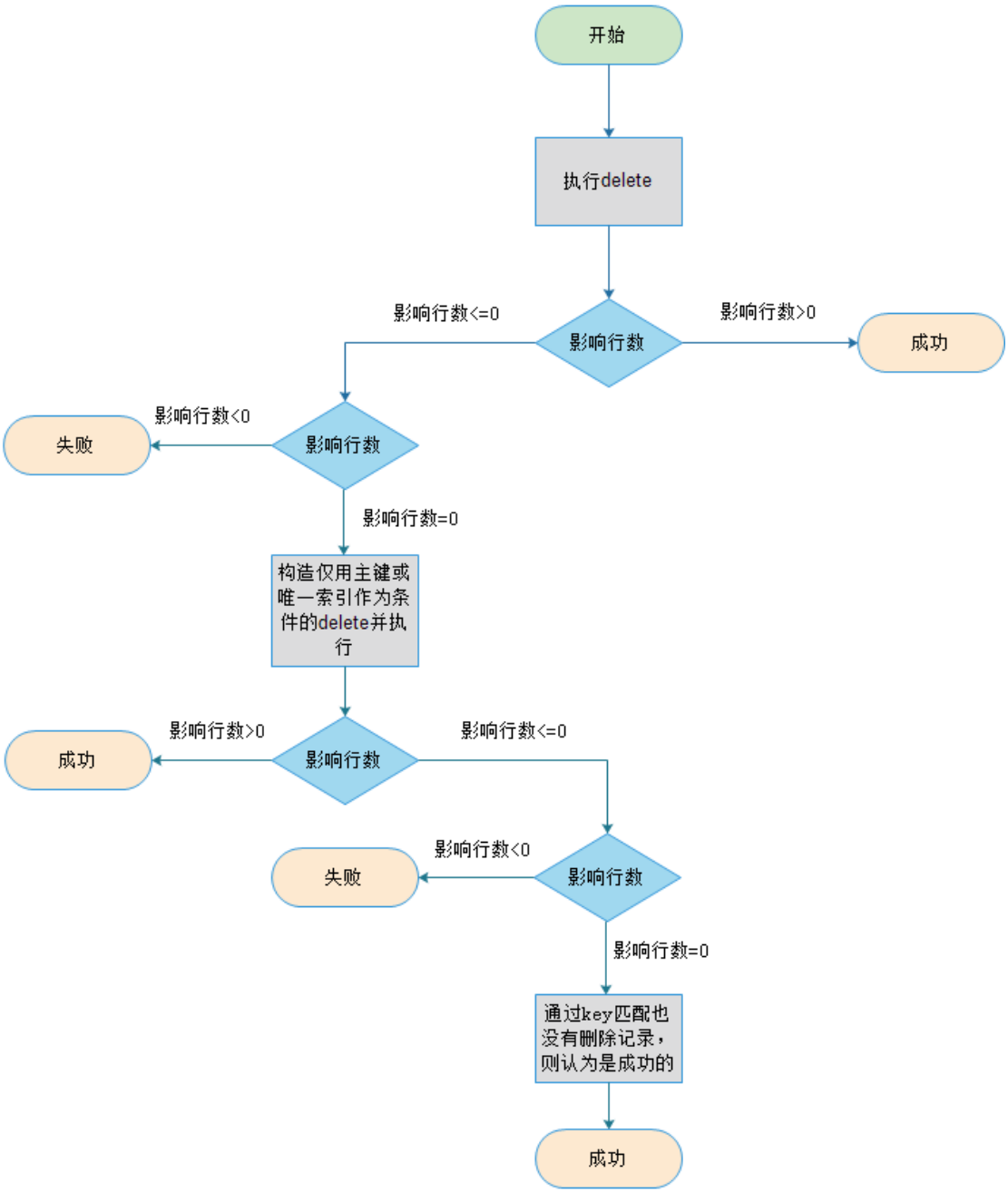
根据上图可以看到，当出现主键冲突时，insert操作会转变成delete+insert操作来保证insert动作执行成功。另外图中的影响行数小于0或者等于0标识执行SQL出错和主键冲突。

- update

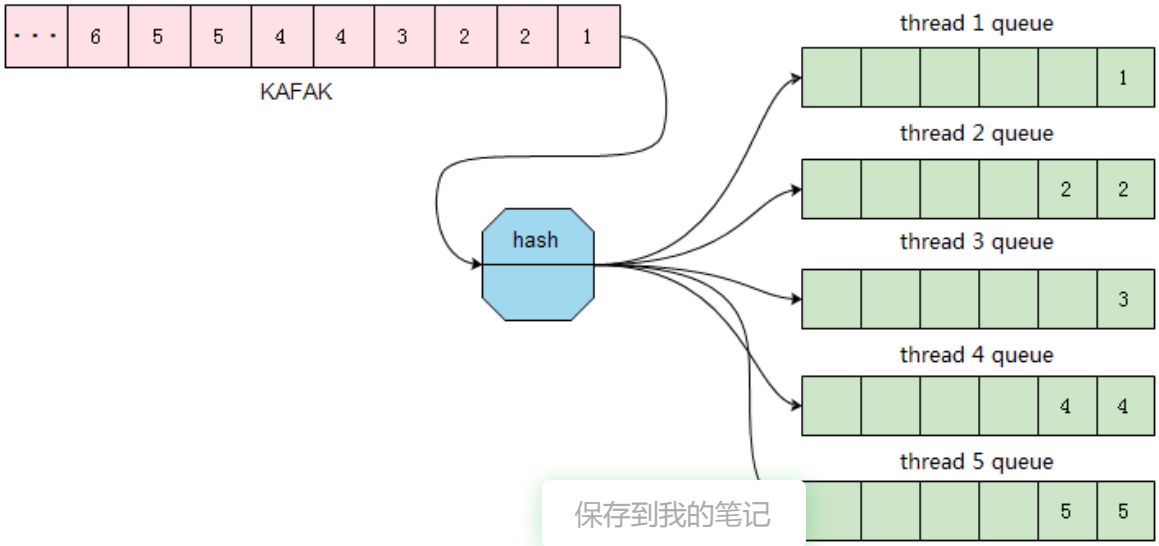


从上图我们可以看到，update操作的幂等处理，其实就是保证了在数据库中，只能有new值产生的记录。

- delete



3.3、多唯一约束条件下的并发控制



从上面的原理图可以看出，在kafka队列中，具有相同主键值的记录会被投送到相同的线程，且线程内是有序

的。 这样的并发方式在下面这样的场景中，会产生数据不一致的情况。以下是对该场景的详细描述。

表sync-table的表结构定义如下：

```
CREATE TABLE `sync_table` (
    `id` int(11) NOT NULL,
    `name` varchar(11) COLLATE utf8_bin DEFAULT NULL,
    `age` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`),
    UNIQUE `unq_1` (`name`)
) ENGINE = InnoDB CHARSET = utf8 COLLATE utf8_bin
```

现在在源端mysql执行下列操作序列

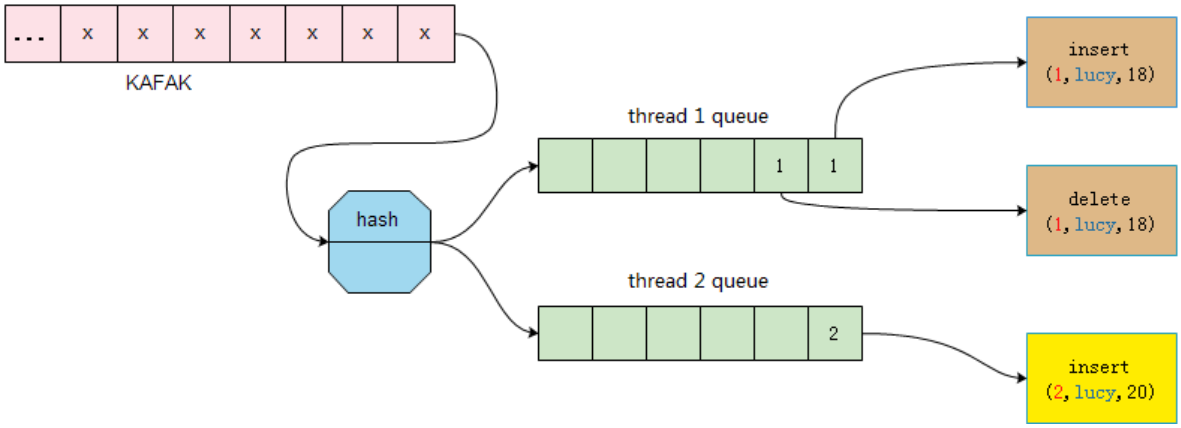
```
mysql> insert into sync_table (id ,name ,age) values (1,"lucy",18);
Query OK, 1 row affected (0.00 sec)

mysql> delete from sync_table where id = 1 and name = "lucy";
Query OK, 1 row affected (0.00 sec)

mysql> insert into sync_table (id ,name ,age) values (2,"lucy",20);
Query OK, 1 row affected (0.01 sec)

mysql> select * from sync_table;
+----+------+------+
| id | name | age  |
+----+------+------+
|  2 | lucy |   20 |
+----+------+------+
1 row in set (0.00 sec)
```

该操作序列会产生三条binlog，分别是insert (1,lucy,18)，delete(1,lucy,18)，insert(2,lucy,20)。那么这三条binlog事件会按照下图所示的方式分发到不同的同步线程。



因为线程间的执行顺序是完全并发的，因此这三个操作在两个线程间的执行顺序可能为以下几种情况。

1）线程1比线程2执行的早

thread1	thread2
insert (1,lucy,18)	
delete(1,lucy,18)	
	insert(2,lucy,20)

目标实例这种执行顺序与源实例的执行顺序完全一致，不会造成数据的不一致。

2）线程1和线程2执行时序有重叠

thread1	thread2
insert (1,lucy,18)	
	insert(2,lucy,20)
delete(1,lucy,18)	

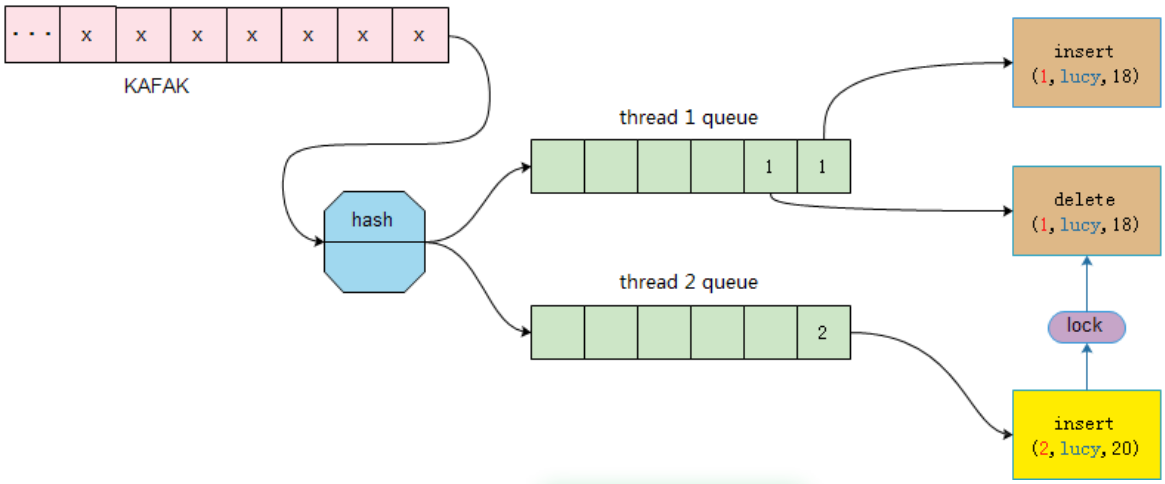
当线程2执行insert时，因为在这之前线程1已经将唯一索引为lucy的记录写入了DB，因此线程2的操作会失败（唯一索引冲突），从而进入幂等流程。这里insert的幂等逻辑是会根据记录中的唯一索引字段先进行一次删除操作，即执行delete where id = 2和delete where name = 'lucy'；然后再将记录插入到数据库，执行insert (2,lucy,20)，保证插入的成功。之后线程1再执行删除操作时，也会进入幂等流程（因为（1，Lucy，8）不存在，delete的影响行数为0），最终目标实例的状态是存在记录（2，lucy，20）。结果正确，不会造成不一致。

3）线程2先与线程1执行完毕

thread1	thread2
	insert(2,lucy,20)
insert (1,lucy,18)	
delete(1,lucy,18)	

线程2执行完insert后，线程1执行insert会因为唯一索引约束冲突而报错失败，从而进入幂等流程。线程1会执行delete where id = 2 和 delete where name = 'lucy'；之后在执行insert(1,lucy,18)。最终当线程1全部执行完后，目标实例内不存在（2，lucy，20）这条记录。造成了数据的不一致。

经过上述的问题描述，我们可以发现，产生数据不一致的原因其实是当前的数据并发策略在多唯一约束的条件下不能按照正确的时序来进行重放。因此在处理这种既有主键又包含一个或多个唯一索引表的数据时，我们就需要额外的手段来保证分布在多个线程中的binlog事件按序执行。这里consumer采用的解决方案是在分发binlog事件到多个同步线程中的时候，同时下发一个锁结构，来协调多个线程中含有相同唯一约束值binlog事件的执行顺序。如下图所示

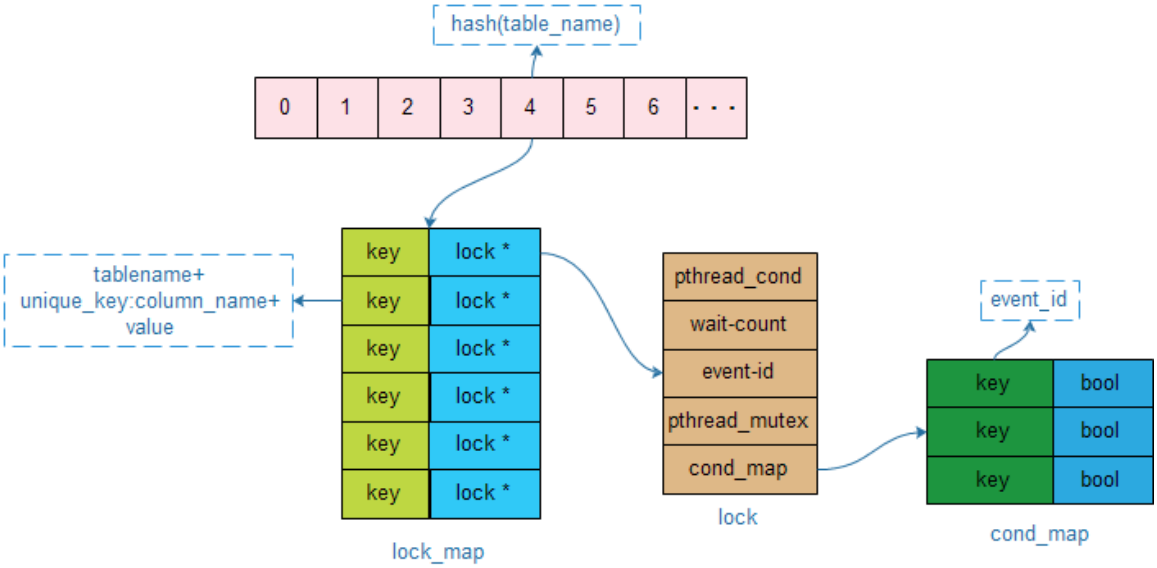


这里线程1的delete事件与线程2的insert事件 保存到我的笔记 如果线程1的delete事件没有执行结束，则线程2的insert事件不会执行。

- 锁的设计

- 锁的设计

根据上面的分析我们知道，当一个表的约束定义除了包含主键外，还包含唯一索引的话，则需要保证相同唯一索引的事件按照顺序来执行。因此这里锁的语义应该是，如果有前序包含相同唯一索引值的事件没有执行完，则需要等待，待其执行完后在执行当前事件。consumer的锁结构如下所示



所有的锁结构是存放在一个全局的数组中，在锁下发的时候，根据表名做一次hash，得到数组的下标。数组中的每一项包含了一个hash_map结构，其中key由表名+唯一索引列名+该列的值构成，类型为字符串；该key对应的value值为一个锁结构的指针lock*。

lock结构中包含下列成员

pthread_cond	条件变量
wait-count	计数变量，标识持有该锁的记录的数量
event-id	上次获取改锁结构的事件ID，全局自增
pthread_mutex	互斥量
cond_map	标识event_id所获取的对应的锁结构，是否释放；key为event_id，value为bool类型，标识当前状态是持有还是释放。

- 锁的下发

当consumer的dispatch线程对消息进行分发时，首先检测，该消息所对应的表是否包含初主键外的唯一约束，如果有的话，则需要在下发该条消息时，一并下发锁结构。开始时，会首先根据表名和唯一索引的信息，查询是否包含该锁结构，如果包含则直接进入下发流程，如果不包含，则创建一个锁，并将其写入lock_map中，然后开始锁下发：

1、自增锁结构中的wait-count。

2、保存event-id到变量wait_event_id，之后将自己的event_id写入锁结构中

3、更新cond_map,将自己event_id的键值对写入cond_map ,并标识为持有该锁。

4、将该锁结构句柄和在第二步保留下的event_id，随着消息体一并下发到同步线程。

- 锁的维护

同步线程在处理消息时，首先会检测改消息[illegible]果包含锁结构的话，处理流程如下所示。

1、check锁结构中的cond_map，检查key为wait_event_id的锁是否释放，如果没有释放则开始pthread_cond_

wait()。

2、当收到条件变量通知时，检测cond_map中wait_event_id的锁是否释放，如果没有释放则继续wait()。

3、当收到条件变量通知时，检测到cond_map中wait_event_id的锁已经释放，则开始对该消息进行重放。

4、重放该消息结束后，更新锁结构中的wait-count减1。如果更新后wait-count等于0，则说明该锁上已经没有任何消息持有。销毁改锁。

5、如果更新后wait-count大于0，则说明还有消息再等待该锁结构。更新cond_map，将自己event_id对应的value更新为释放状态，并且将wait_event_id对应的键值对删除。

6、执行完上述操作后执行broadcast()操作，通知其他等待线程。

四、优化与展望