03 TiDB**数据库分布式事务**

[03 TiDB**数据库分布式事务**](#_bookmark0)

[1**、分布式事务基本原理**](#_bookmark1)

* 1. [分布式事务知识](#_bookmark2)
     1. [事务的定义与特性](#_bookmark3)
     2. [隔离级别](#_bookmark4)
  2. [分布式事务⽅案](#_bookmark5)
     1. [伪分布式架构](#_bookmark7)
     2. [TCC架构](#_bookmark6)
     3. [SAGA事务架构](#_bookmark8)
     4. [2PC(分布式事务架构)](#_bookmark9)

[2**、**Percolator**事务模型原理**](#_bookmark10)

* 1. [Percolator简介](#_bookmark11)
  2. [Percolator三要素](#_bookmark12)
     1. [Snapshot Ioslation](#_bookmark13)
     2. [分布式时钟](#_bookmark16)
     3. [Percolator事务执⾏流程](#_bookmark15)
        1. [Percolator执⾏流程详解](#_bookmark14)
        2. [Percolator事务执⾏案例](#_bookmark17)
        3. [percolator的优缺点](#_bookmark18)

[3**、**TiDB**事务实现⽅式与优化**](#_bookmark19)

* 1. [TiDB 事务](#_bookmark20)
     1. [事务控制](#_bookmark21)
     2. [TiDB事务隔离级别](#_bookmark22)
  2. [TiDB数据库事务实现](#_bookmark23)
  3. [TiDB数据库锁](#_bookmark24)
     1. [事务执⾏流程介绍-乐观锁模式](#_bookmark25)
     2. [事务执⾏流程介绍-悲观锁模式](#_bookmark27)

[（1）悲观锁介绍](#_bookmark26)

[(2) 悲观锁的实现](#_bookmark28)

* + 1. [乐观锁和悲观锁的优缺点](#_bookmark29)
  1. [事务去中⼼化](#_bookmark30)
  2. [TiDB事务优化](#_bookmark31)
     1. [乐观锁和悲观锁设置](#_bookmark32)
        1. [乐观锁实验](#_bookmark33)
        2. [悲观锁实验](#_bookmark34)
     2. [写偏斜write skew](#_bookmark35)
        1. [写偏斜示例](#_bookmark36)
        2. [解决写偏斜](#_bookmark37)
     3. [事务⼤⼩限制](#_bookmark39)
     4. [事务错误处理](#_bookmark38)

1、分布式事务基本原理

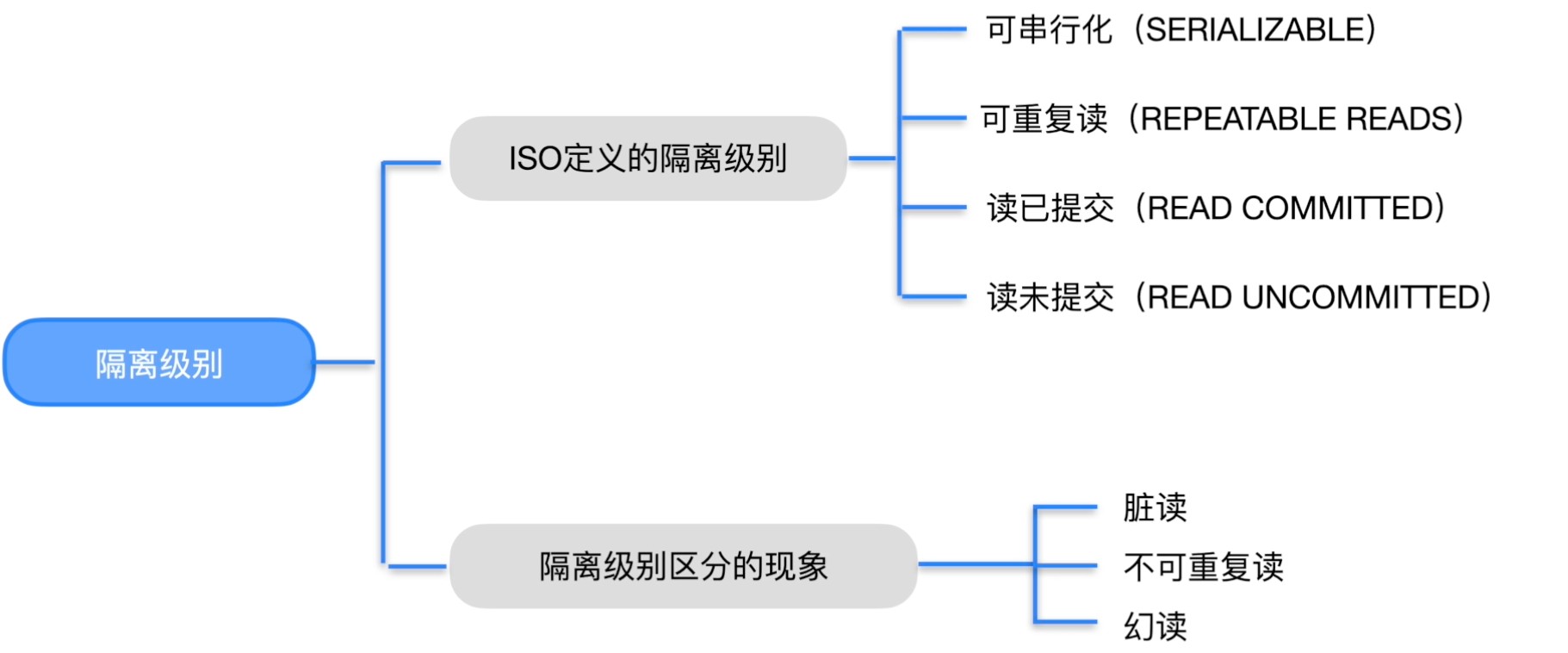
# 分布式事务知识

* + 1. **事务的定义与特性**

事务是数据库⼀系列操作集合,是数据改变的不可分割基本单位。

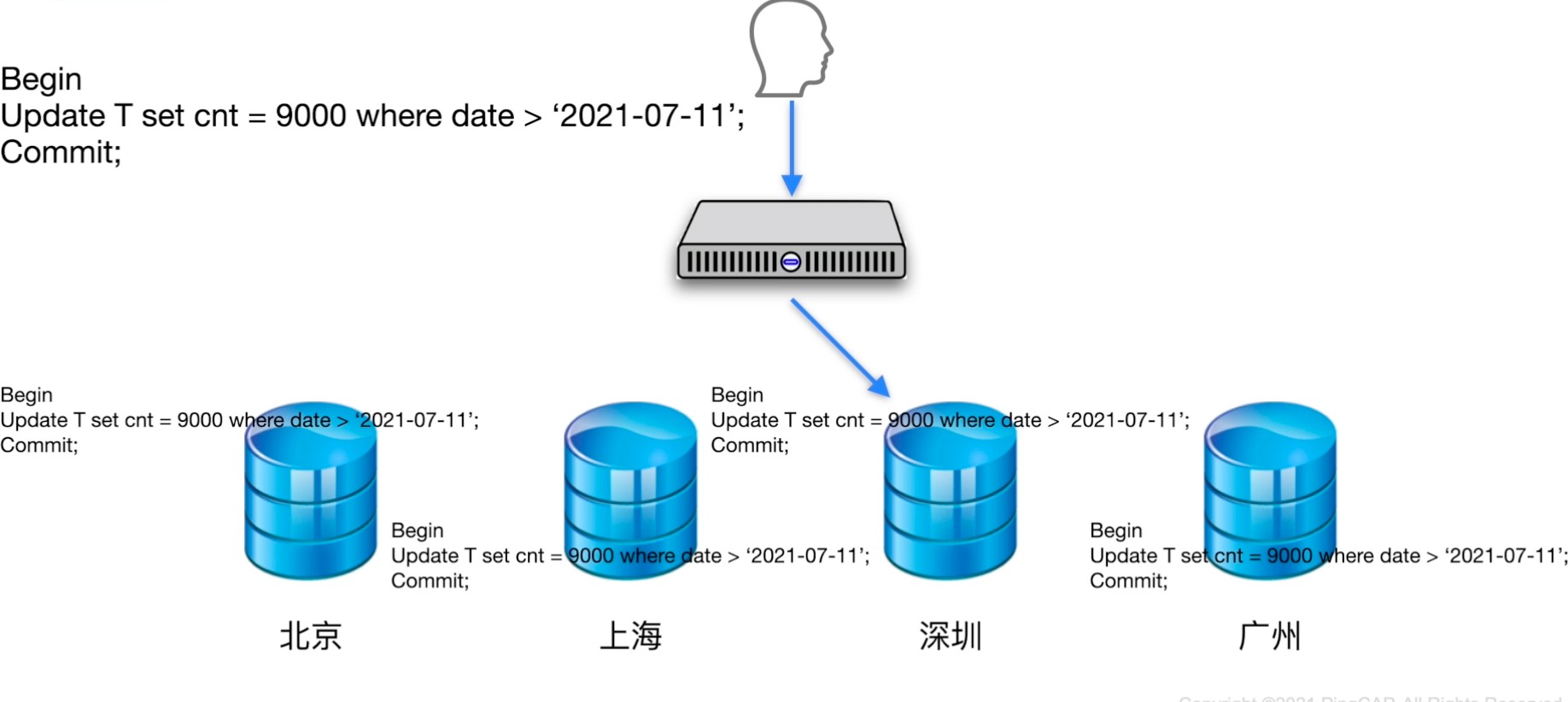
事务四⼤特性ACID:

* 原⼦性（Atomicity）：事务必须是原⼦⼯作单元，对于其数据修改，要么全执⾏，要么全不执⾏。
* ⼀致性（Consistency）：事务在完成时，必须使所有的数据都保持⼀致状态。
* 隔离性（Isolation）：由并发事务所做的修改必须与任何其他并发事务所作的修改隔离（简⽽⾔之：⼀个事务执⾏过程中不应受其它事务影响）。
* 持久性（Durability）：事务完成之后，对于系统的影响是永久性的。
  + 1. **隔离级别**



# 分布式事务⽅案

* + 1. **伪分布式架构**



伪分布式系统-mycat等中间件

* ⼀张表按地区进⾏拆分，分成了4个库
* 提交⼀个sql将date⼤于2021年7⽉11号的数据的cnt字段的值修改为9千
* ⼀个sql在3个地区修改成功,第四个地区失败,⽐如⼴州没变,则破坏⼀致性,事务应该回滚
  + 1. TCC**架构**

事务架构特点: 完全由程序控制事务, 对业务侵⼊极⾼

Try：

1.⼤事务在两个分⽚做修改,到commit之前

Commit:

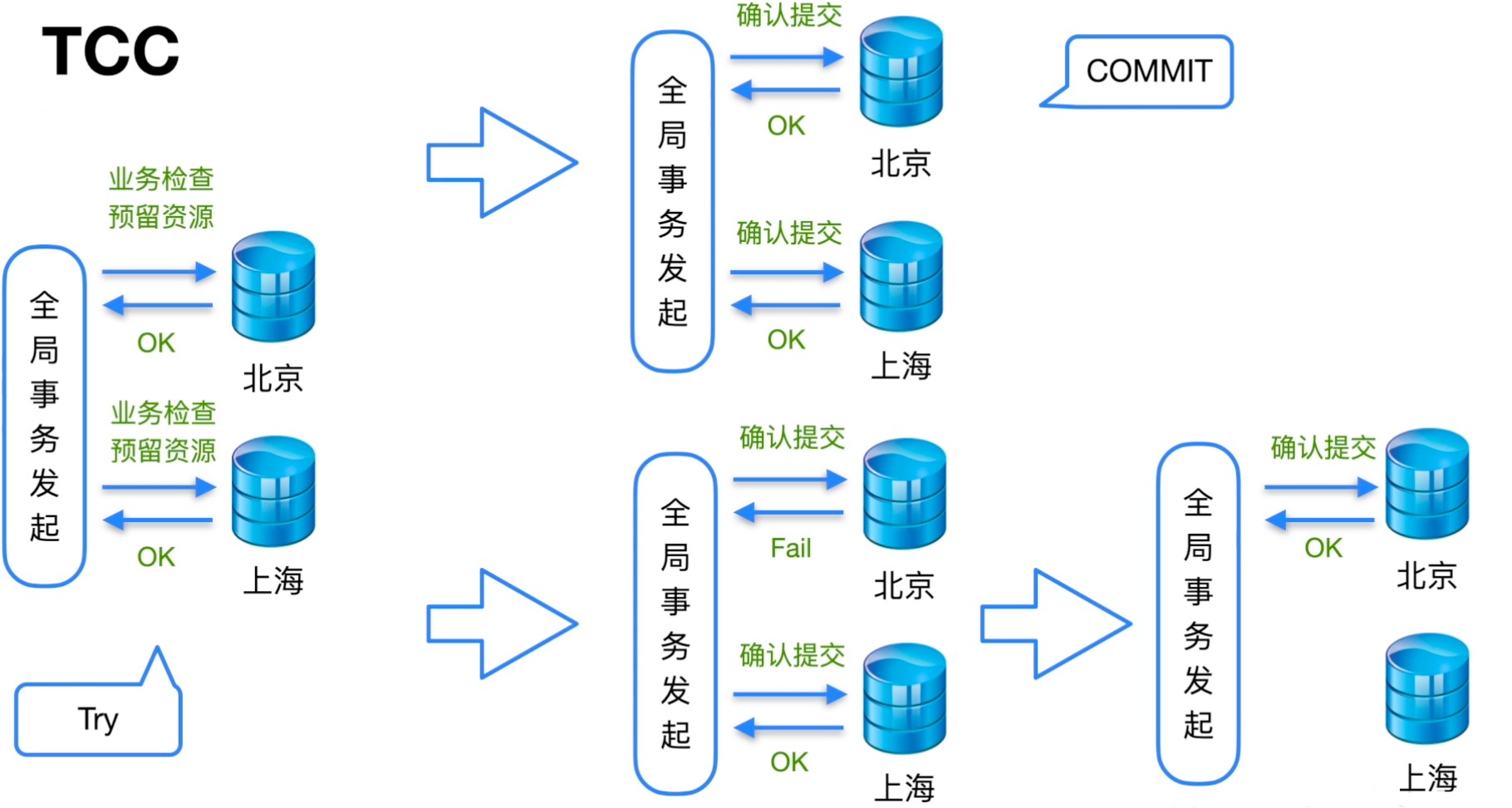
1.两个分⽚确认提交,则事务commit成,事务成功

第⼆个C:

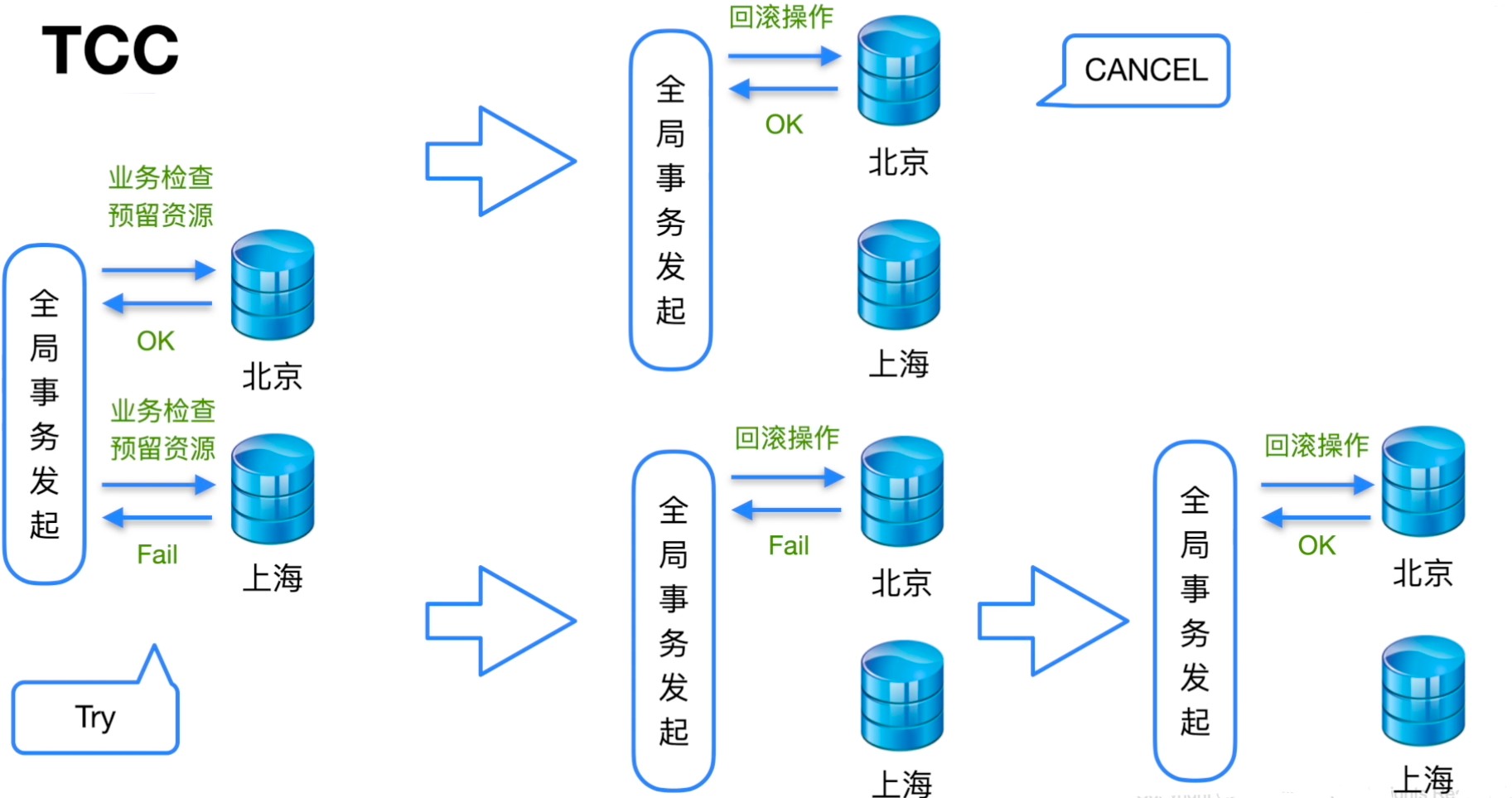
if 北京 commit失败

上海先提交,北京不停重试直到commit

其中⼀个commit失败



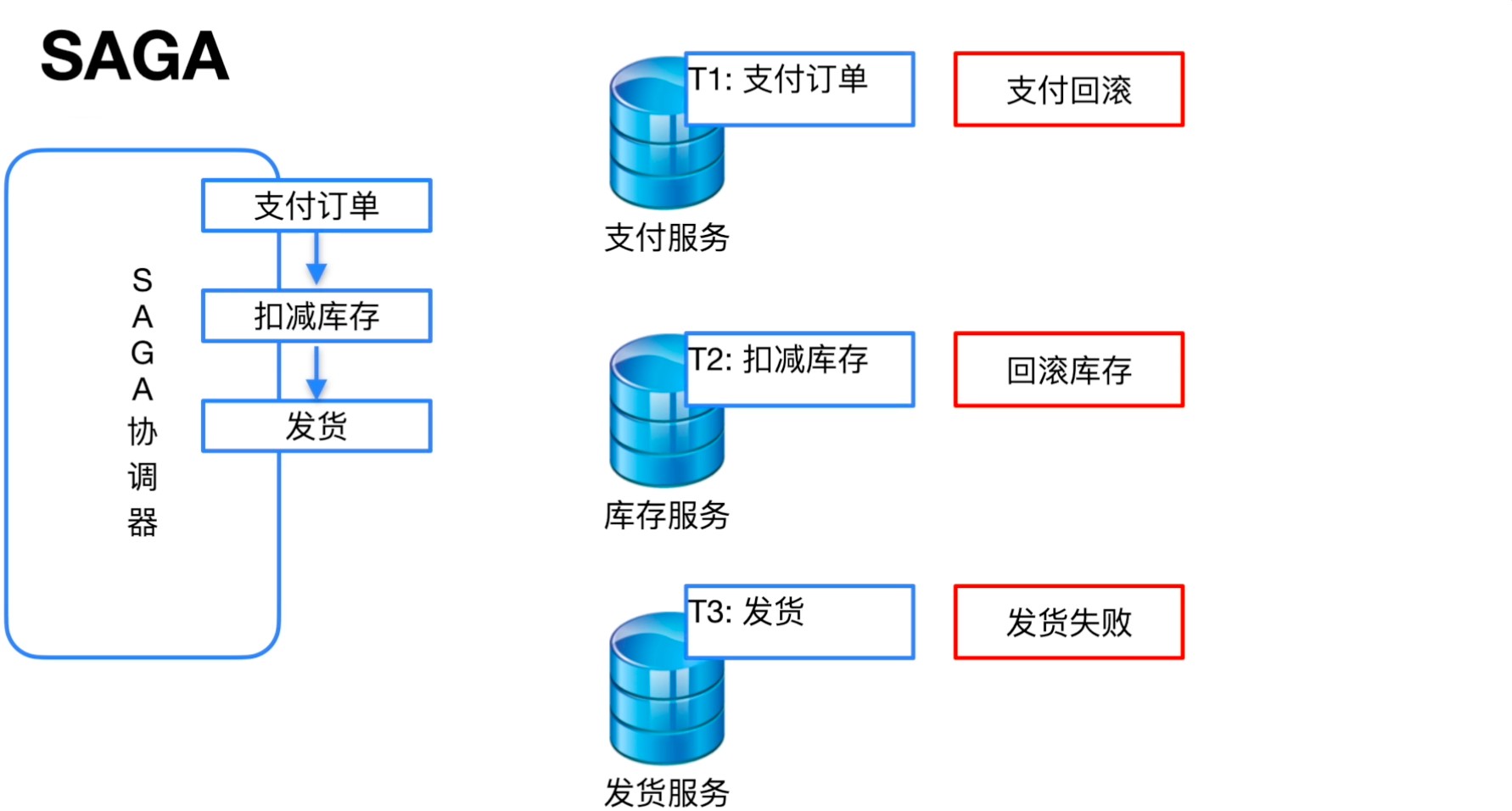
Try失败阶段



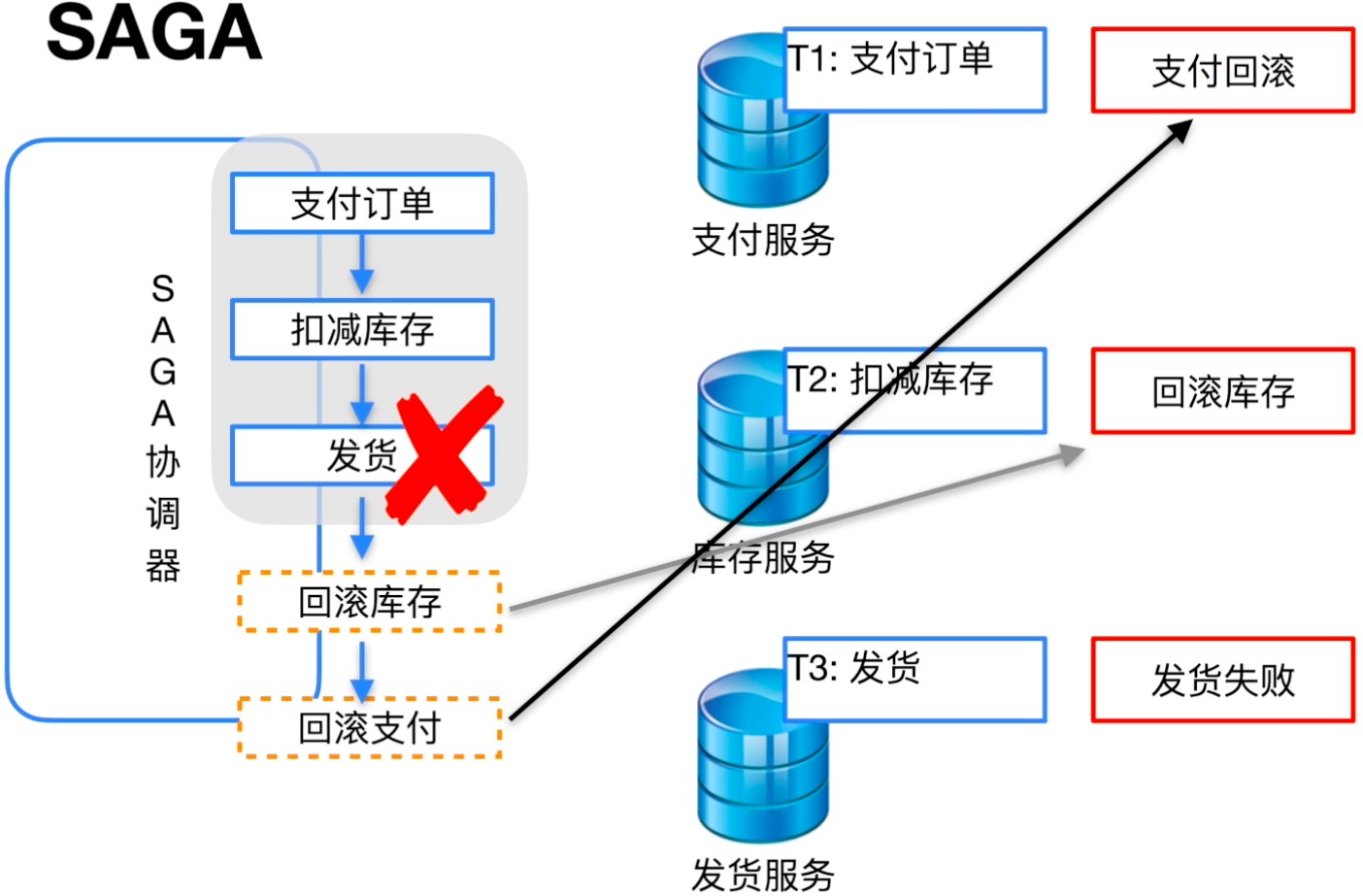
直接进⾏回滚操作,不断重试☞回滚成功

* + 1. SAGA**事务架构**

核⼼理念: ⼤事务拆解成⼩事务,每个步骤失败则进⾏回滚,任何⼀步失败则前⾯成功的事务进⾏回滚特点: 对业务的侵⼊性相对较低，⽐较适合微服务类型的架构



回滚已完成事务步骤

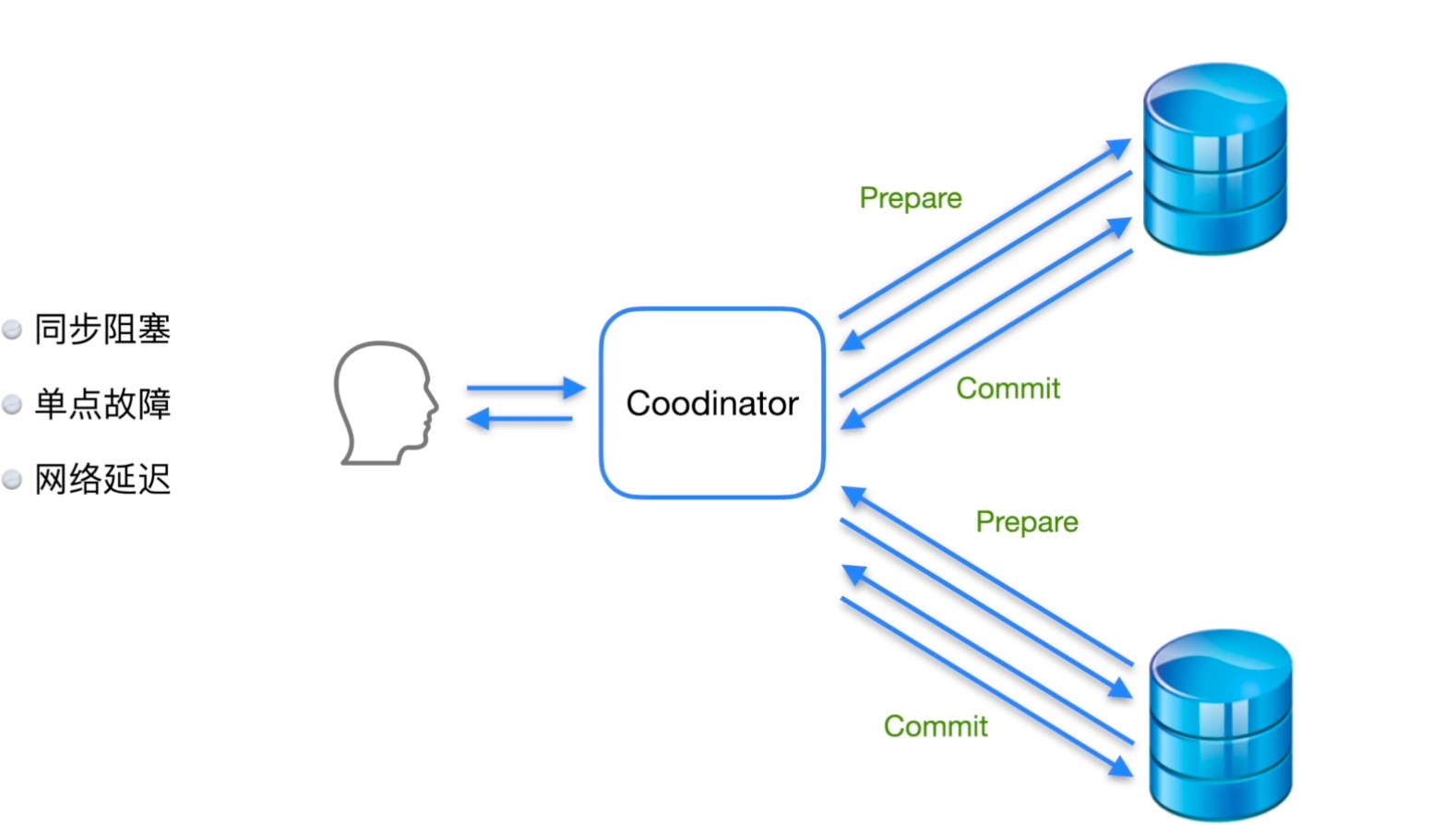


* + 1. 2PC(**分布式事务架构**)

2PC(Two-PhaseCommit)

核⼼理念: 事务的提交和回滚都交给数据库端，对业务0⼊侵。

应⽤:Percolator模型



2**、**Percolator**事务模型原****理**

* 1. Percolator**简介**

#Percolator产⽣背景

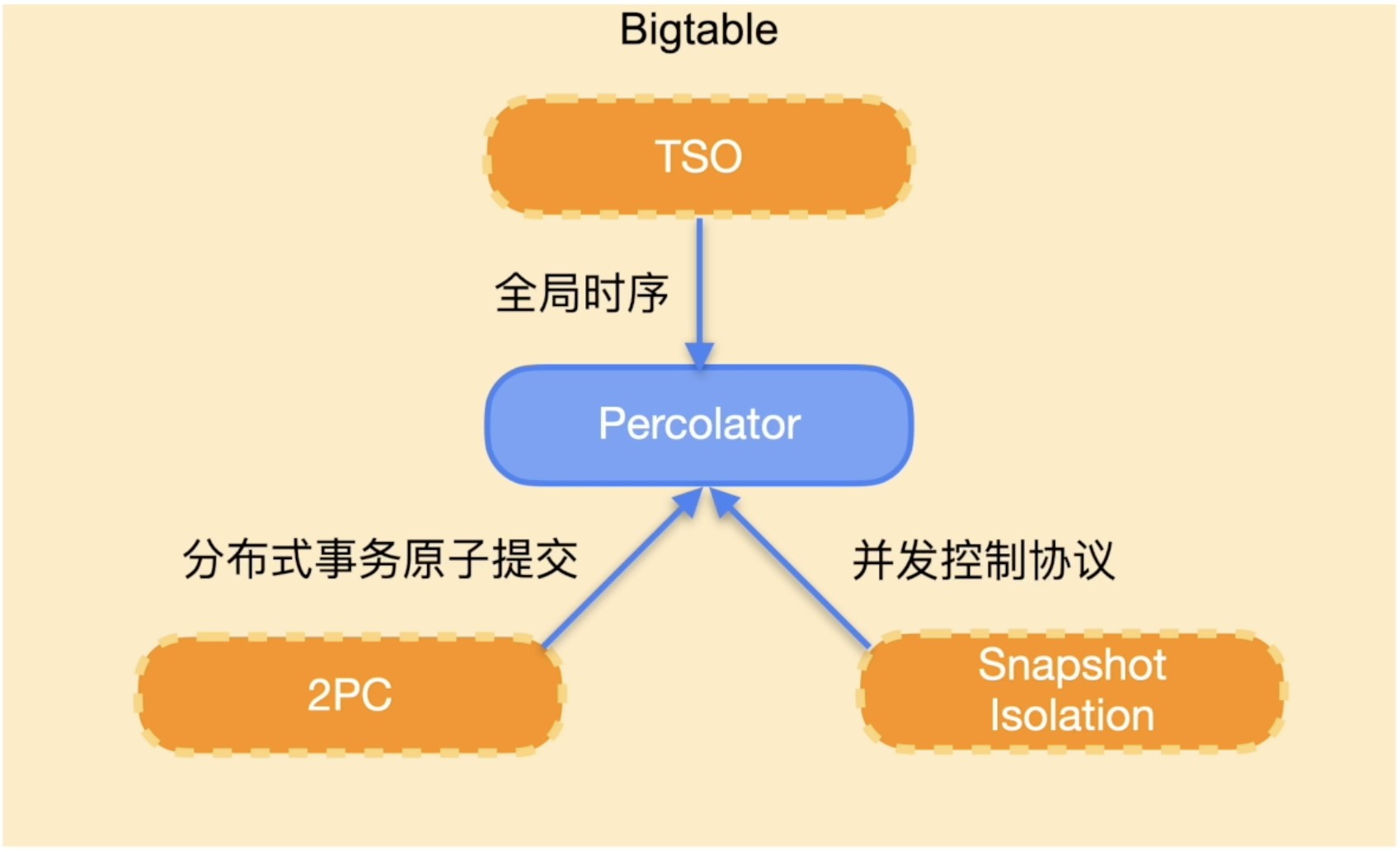
Google提出，应⽤在Bigtable上，解决数千个node的原⼦性事务.

#Percolator三要素

1.TSO 全局时间戳

2.2PC 原⼦提交

3.SI 并发控制协议



* 1. Percolator**三要****素** 
     1. Snapshot Ioslation

定义：

事务只能看到早于它开始时间之前提交的事务

eg:

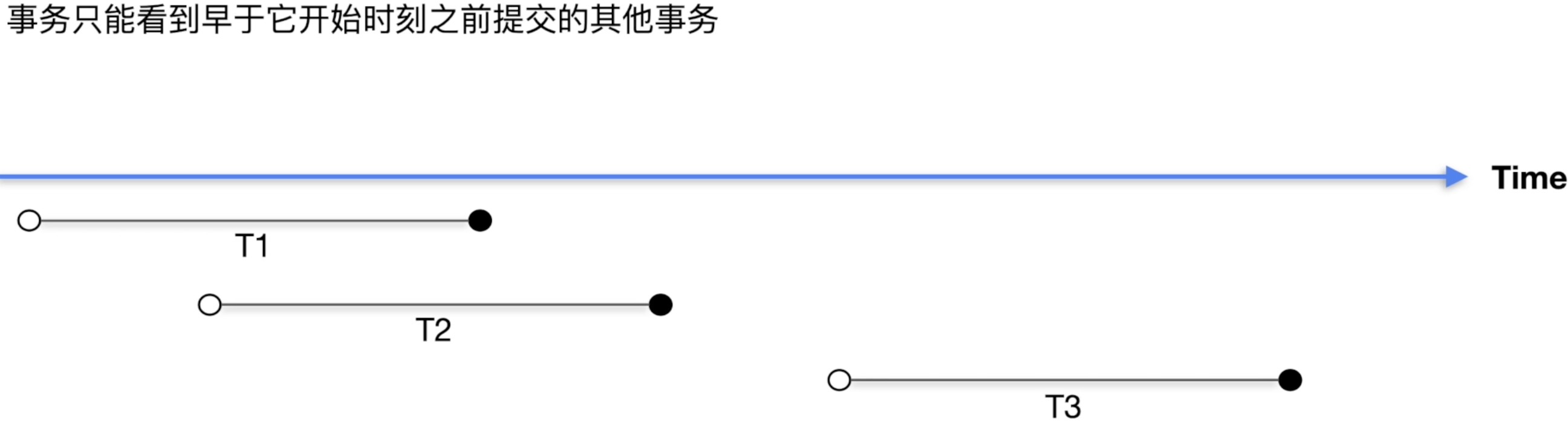
T2看不到T1修改

T3可以看到T1和t2的修改

SI+悲观锁:等于Mysql RR

SI+乐观锁:TiDB4.0默认,事务修改中没有冲突

\*/



* + 1. **分布式时钟**



1. True Time: 真实时钟.对硬件要求⾼,多点校准时间,每个点按照区域时间差校准
2. HLC：使⽤NTP,有误差
3. TSO:单点,有全局授时器发送时钟(PD),单点故障+性能瓶颈

混合:物理+逻辑时钟

* + 1. Percolator**事务执⾏流程**

1. Percolator**执⾏流程详解**

举例:⼀个事务修改三个节点的三个key1 2 3

组件:

1. Coodinator: 协调器,事务提交 TiDB服务
2. TSO: 时间戳,PD
3. 数据: TiKV

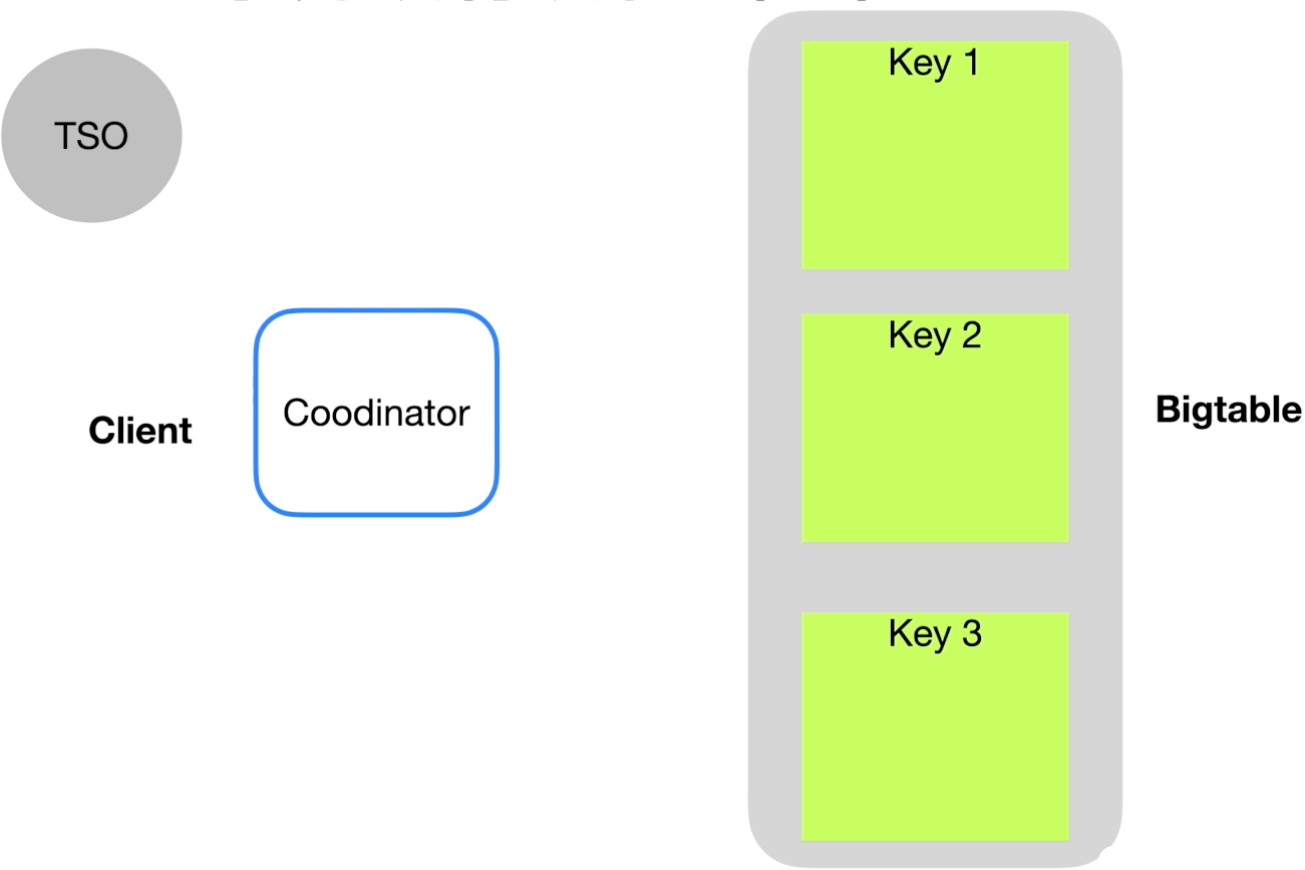
流程:两阶段,pre+com

1. Prewrite:修改数据+lock

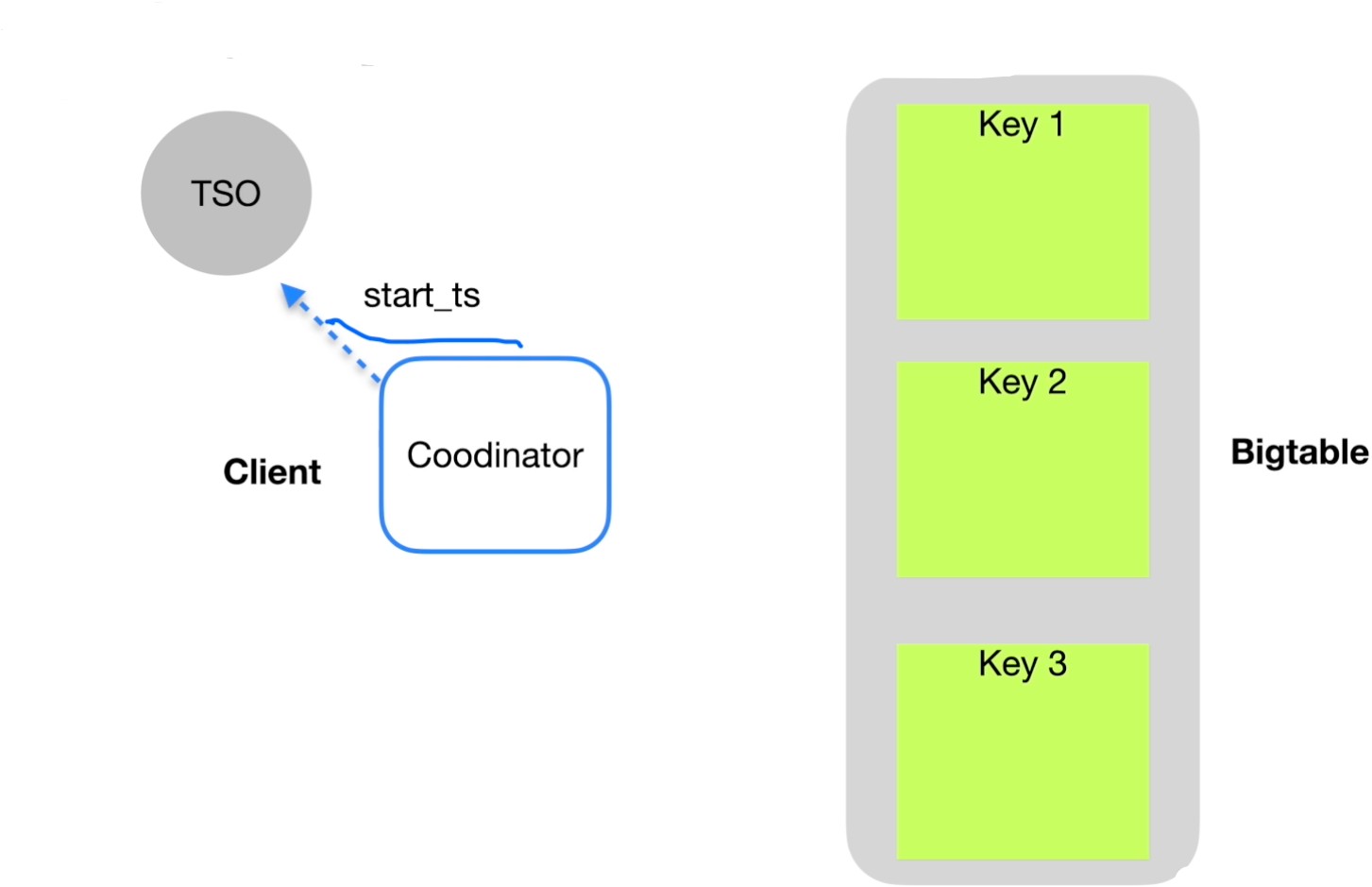
* 找TSO要时间戳
* 在修改的多⾏数据中,⼀⾏pri,其他sec,pri + key(lock cf)，然后给sec⾏加lock,sec指向pri的lock

- 中间有其他事务修改pri+sec key,看到有lock则返回报错

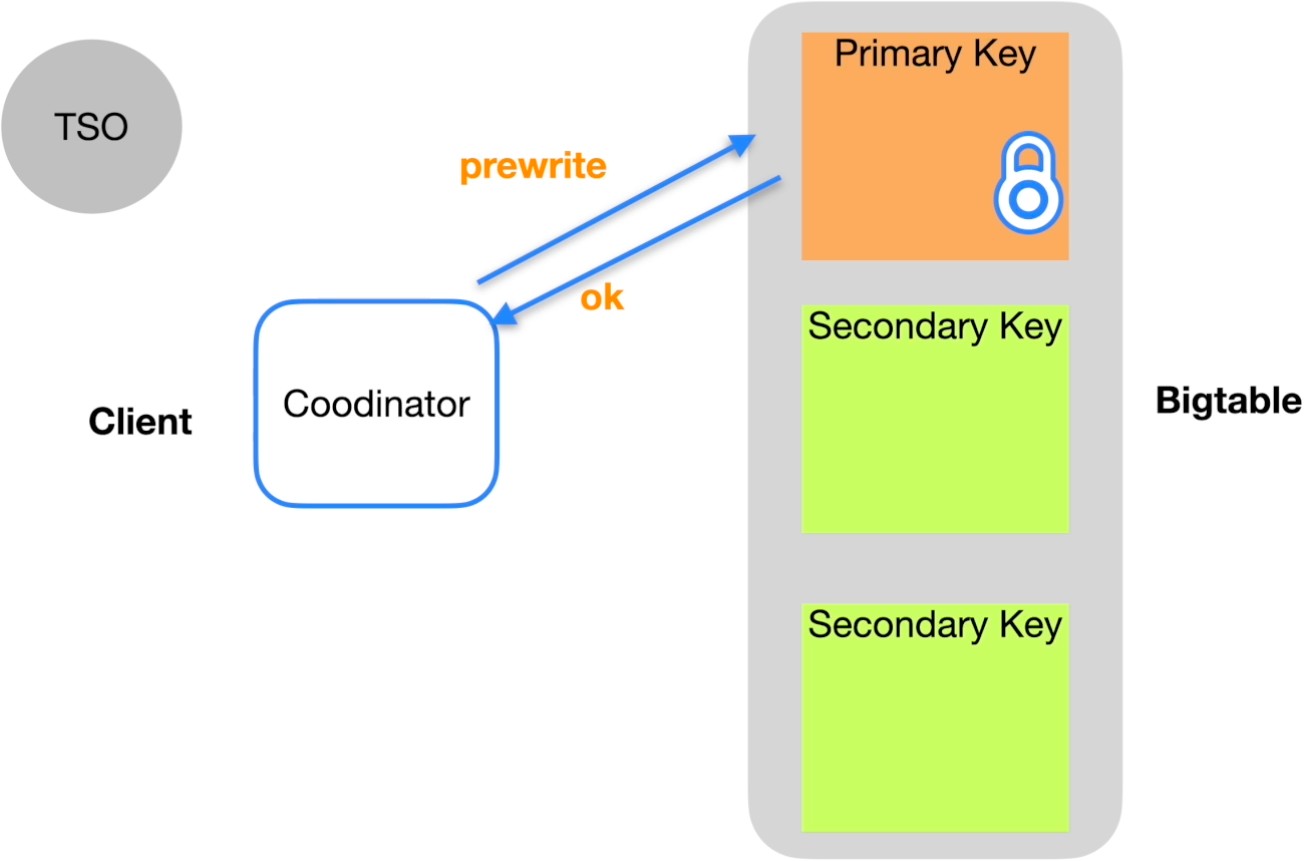
2.Commit阶段:pri sec lock清理+commit



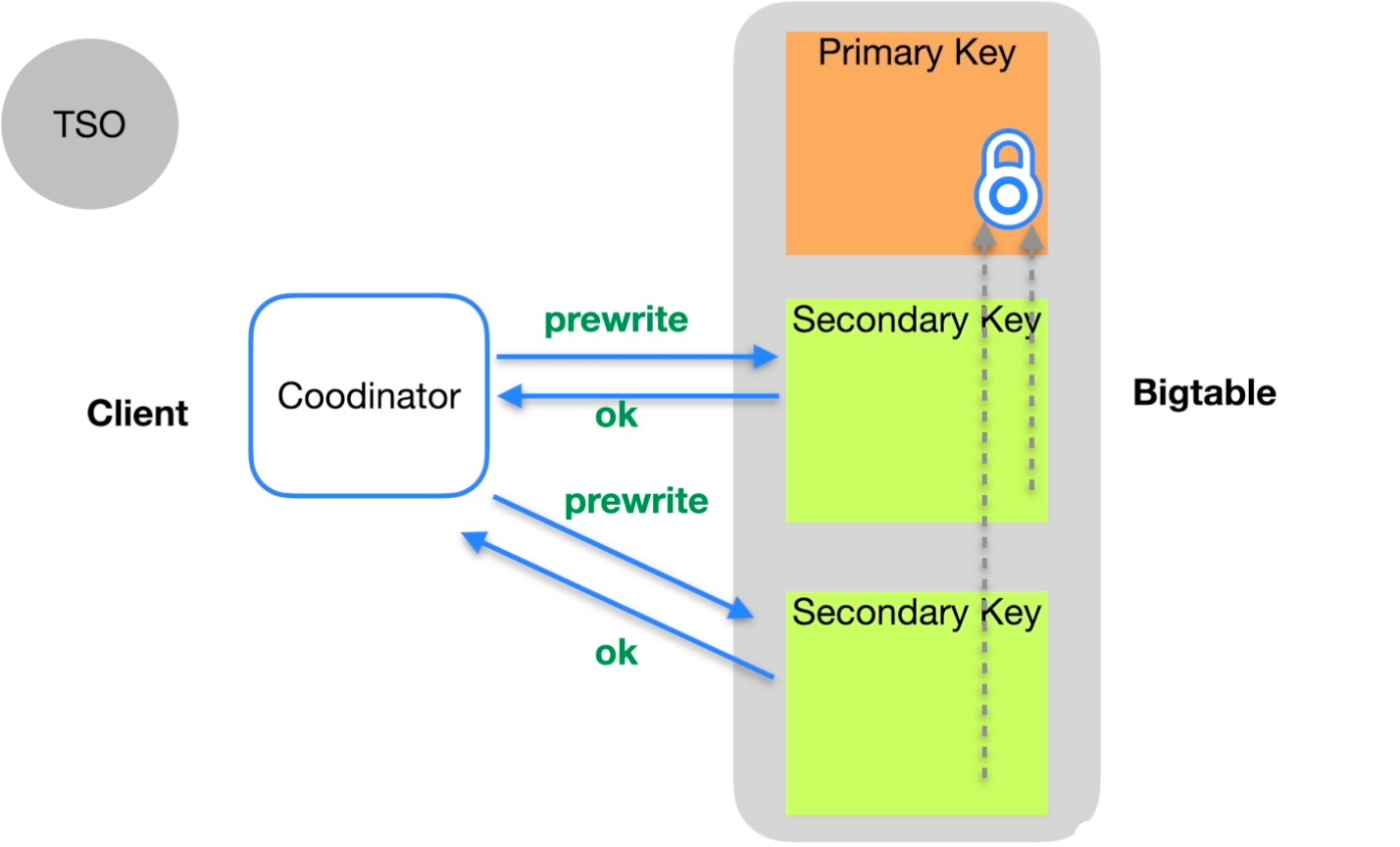
prewrite阶段——获取事务的开始时间



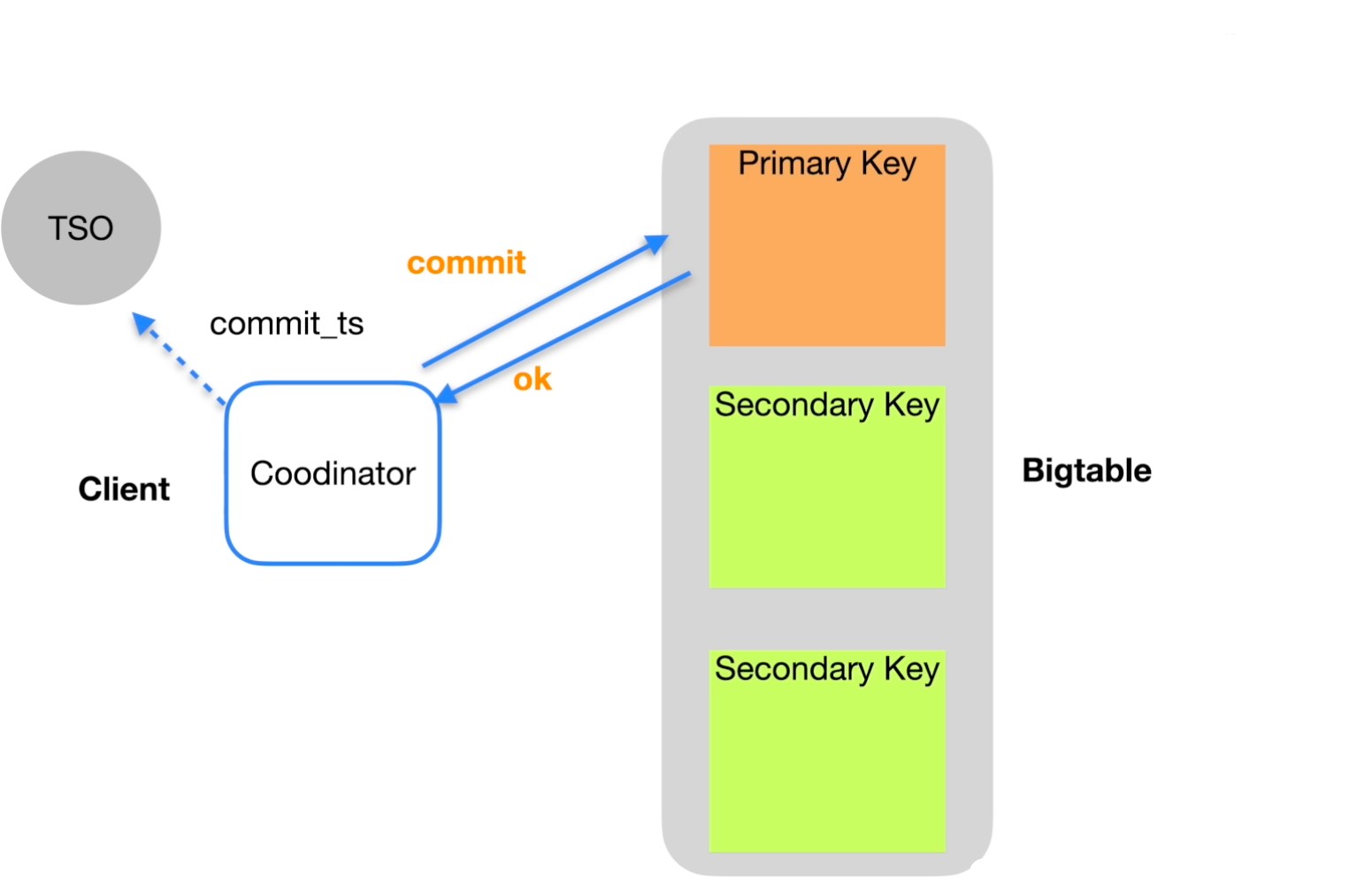
prewrite阶段——主⾏加锁



prewrite——secondary key加锁



commit阶段



1. Percolator**事务执⾏案例**

例⼦:bob-->joe 7 美⾦,在不同节点的数据,

1. data: 5数据时间戳+value
2. wrtie: 6(读取时间) @5(⽬前最新的版本数据,作为索引指向)
3. data lock write分别是rocksdb的cf

流程:

* 1. Prewrite:原⼦操作
     + Bob:版本7 转⾛7美元.pri+7 lock
     + Joe:版本7 加sec lock @Bob -->7：primary

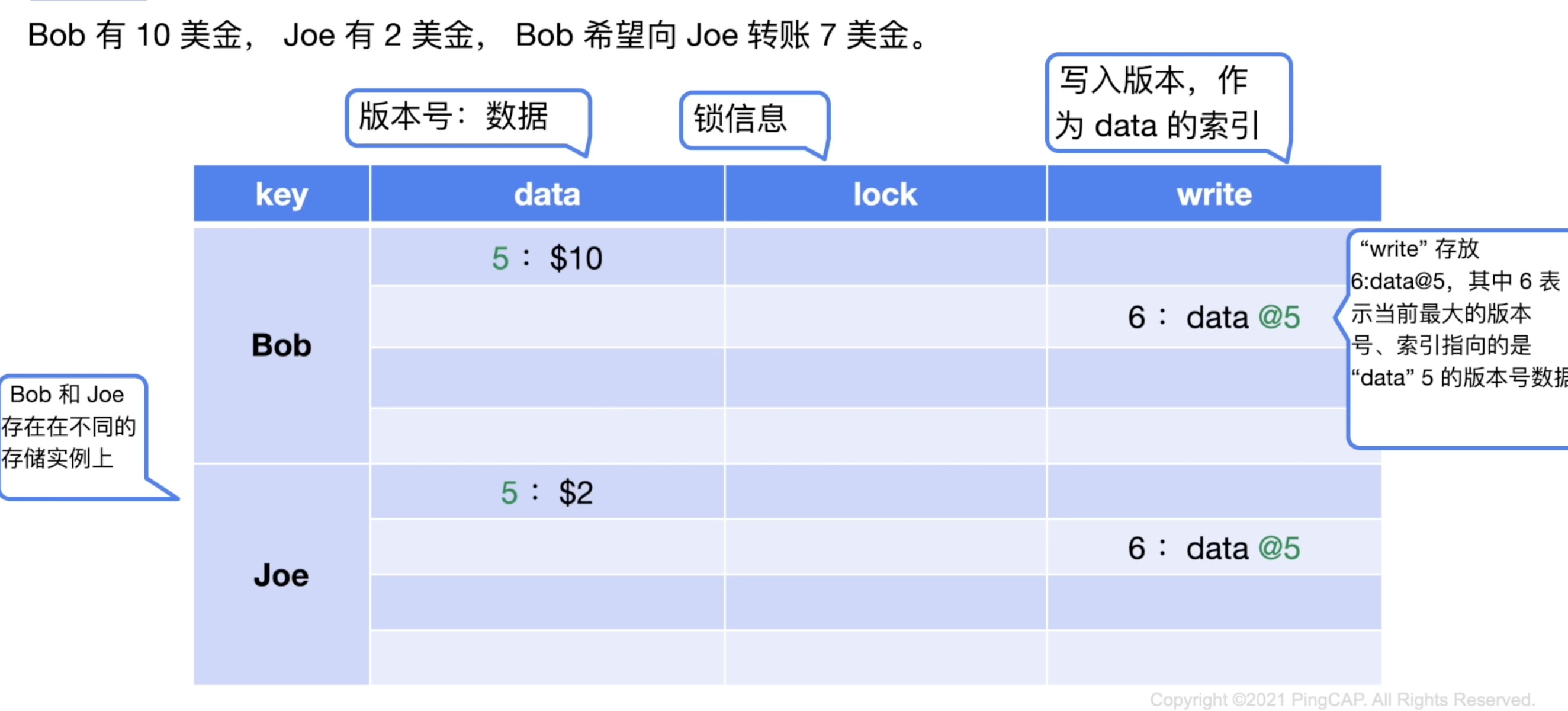
1. commit：
   * 清理7 pri lock,write写⼊最新的版本8,--->@7指向data 版本7
   * 清理7 sec lock，write写⼊最新的版本8,--->@7指向data 版本7

3.故障1,主锁commit，sec lock未清理,主库宕机

- 发现主锁lock，猜测宕机,事务主动补全

故障2:prewrite宕机:直接回滚

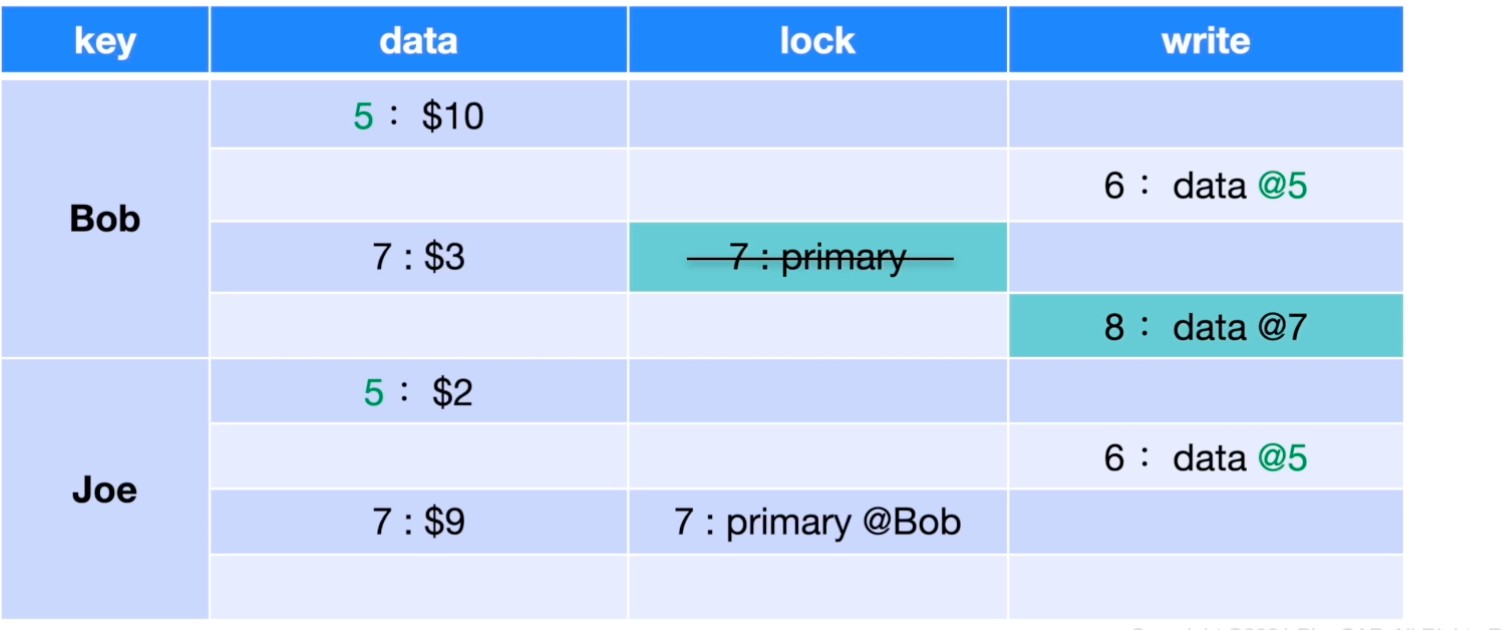
初始数据存储



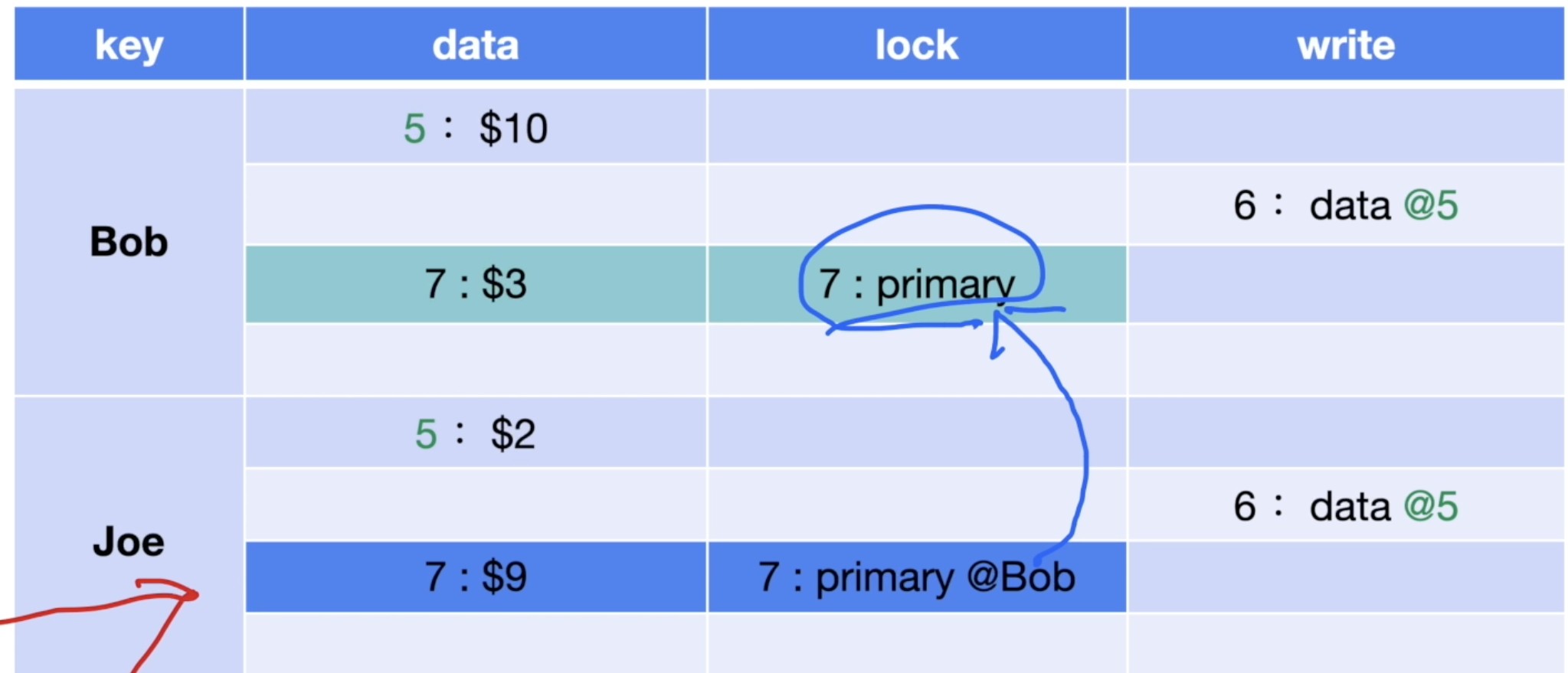
Prewrite阶段：



commit primary



故障



1. percolator**的优缺点**

优点:

1. 简单
2. ⽤单⾏实现跨⾏事务
3. 去中⼼化锁管理

缺点:

* 1. 中⼼化版本号TSO
  2. ⽹络交互多

3**、**TiDB**事务实现⽅式与优****化**

* 1. TiDB **事务**

提供两种事务模式

* 乐观事务
* 悲观事务
  + 1. **事务控制**

1、事务⾃动提交模式

默认情况下，tidb集群在AUTOCOMMIT模式下运⾏

* 只要你执⾏DML，它们就会被⾃动提交
* 每条语句都被当做⼀个事务对待

检查⾃动提交设置：select @@autocommit;

禁⽤⾃动提交模式：

* set autocommit=0;
* begin;
* 使⽤commit/rollback⽅式结束事务。

2、事务控制语句

* START TRANSACTION or begin：显式地开启⼀个新事务
* commit：使当前事务的更改永久化
* rollback：取消当前事务的更改
* set autocommit：控制⾃动提交模式

3、隐式commit

* start transaction，begin，set autocommit=1
* 权限管理操作，例如grant、revoke
* DDL
  + 1. TiDB**事务隔离级别**

TiDB⽀持以下事务隔离级别：

* Read committed
* repeatable read

悲观事务模式⽀持：

* repeatable read
* Read committed

乐观事务模式⽀持：

* repeatable read

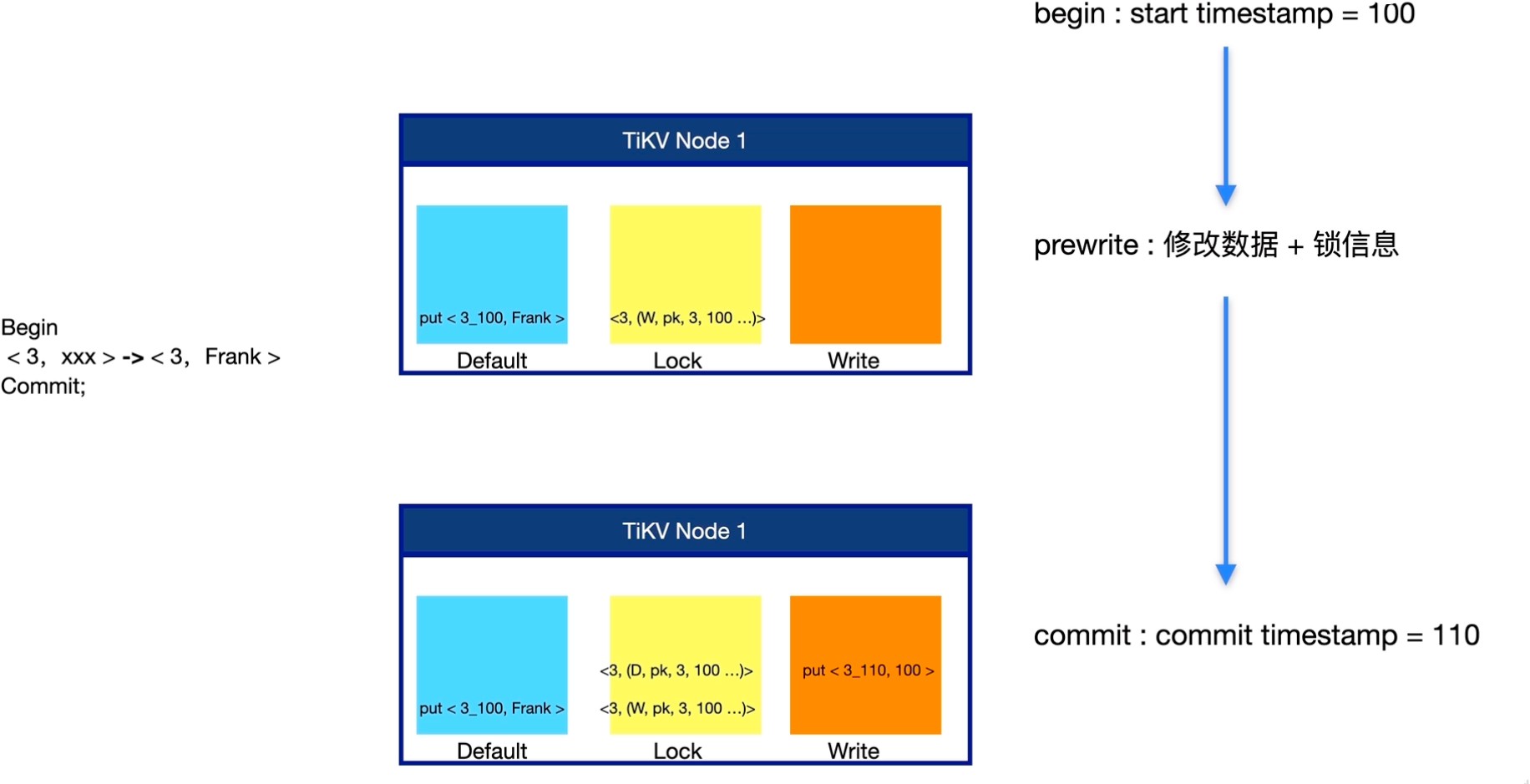
设置会话隔离级别：

set transaction\_isolation='read-committed |repeatable-read';

* 1. TiDB**数据库事务实****现**

事务在tidb中存储-CF+2PC

读取:先读Write CF确认TSO,根据TSO的记录信息找到3\_100 Frank.



* 1. TiDB**数据库****锁** 
     1. **事务执⾏流程介绍**-**乐观锁模式**
        1. ⽤户发出请求到tidb server
        2. 去PD获取TSO
        3. sql语句解析 编译 优化
        4. 执⾏--》PD获取路由信息
        5. 执⾏器从rocksdb取出-->放进tidb server 的memBuffer修改数据
           + data有版本监测
           + 修改事务⼤需要更多内存,注意tidb内存
           + 此时宕机则事务丢失
        6. 事务在membuffer修改
           + startts = 120,数据进⾏修改
           + 数据内存修改完毕,进⾏2PC落盘
        7. TiDB的2PC
           + Prewrite阶段:

版本冲突监测:如果有事务已经提交,且是此⾏数据被修改,造成写写冲突(在本事务提交内其他事务提交),看

write是否有最新版本信息且⼩于本事务的start ts，⼩于start ts继续,否则回滚

锁冲突监测:看是否有锁信息在lock cf

如果有D锁 or ⽆锁信息,继续

如果没有D锁, 有W锁代表已经有事务,回滚

加锁:w

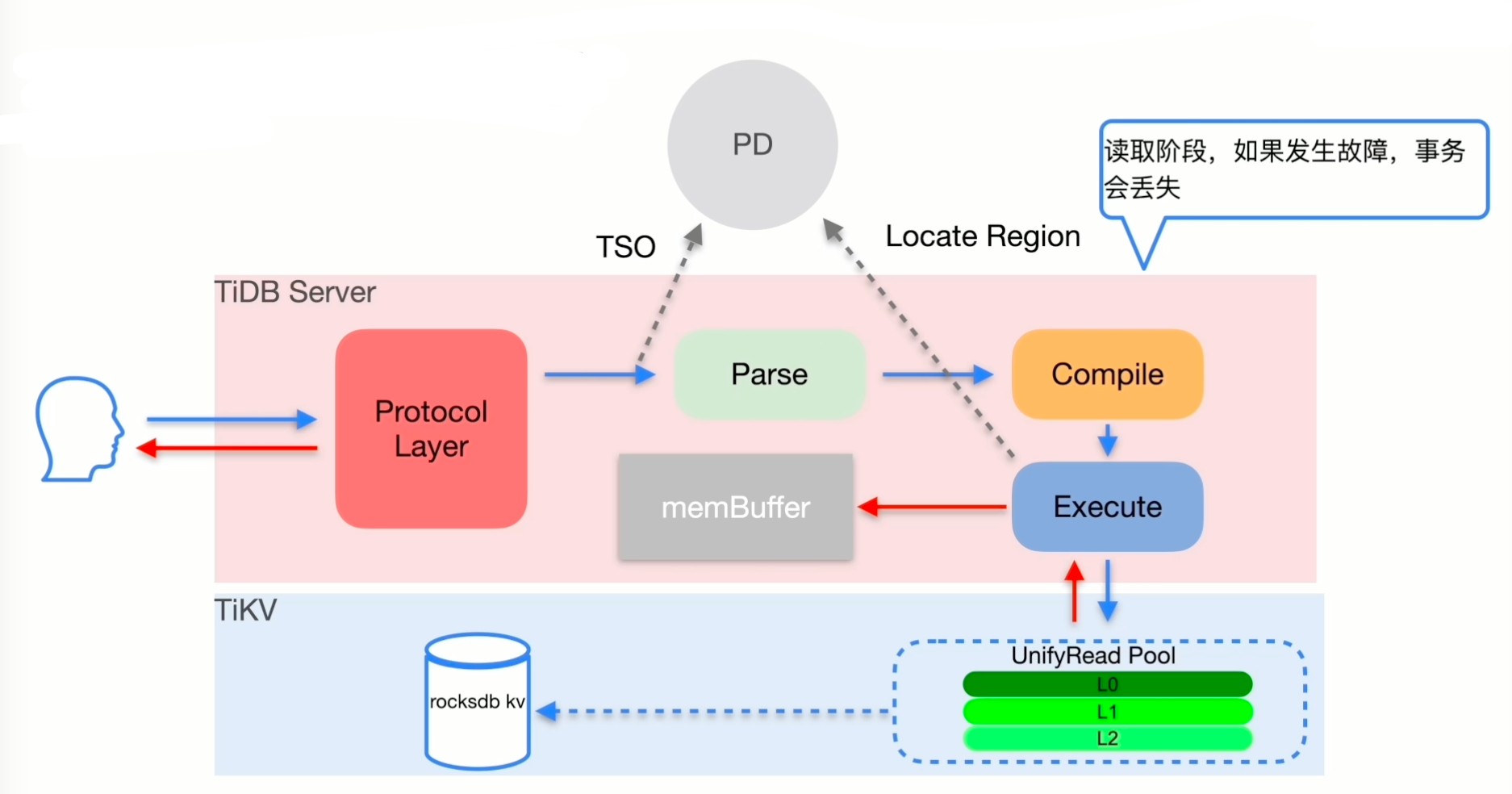
* + - * + Commit锁检查和提交

检查锁 获得提交时间TSO

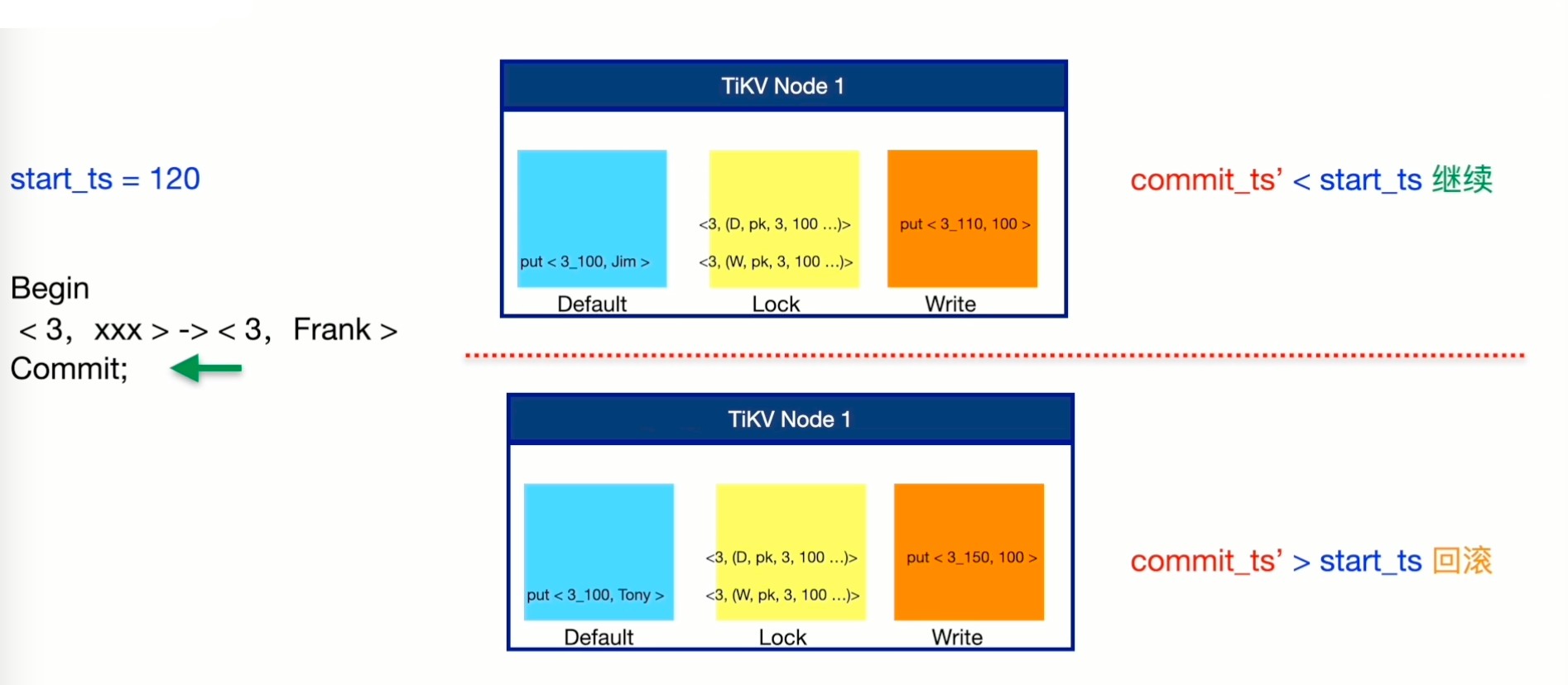
记录最新版本 write cf 3\_160,120,然后清理lock cf <3,D,pk,3,120>

数据落盘rocksdb到sst⽂件

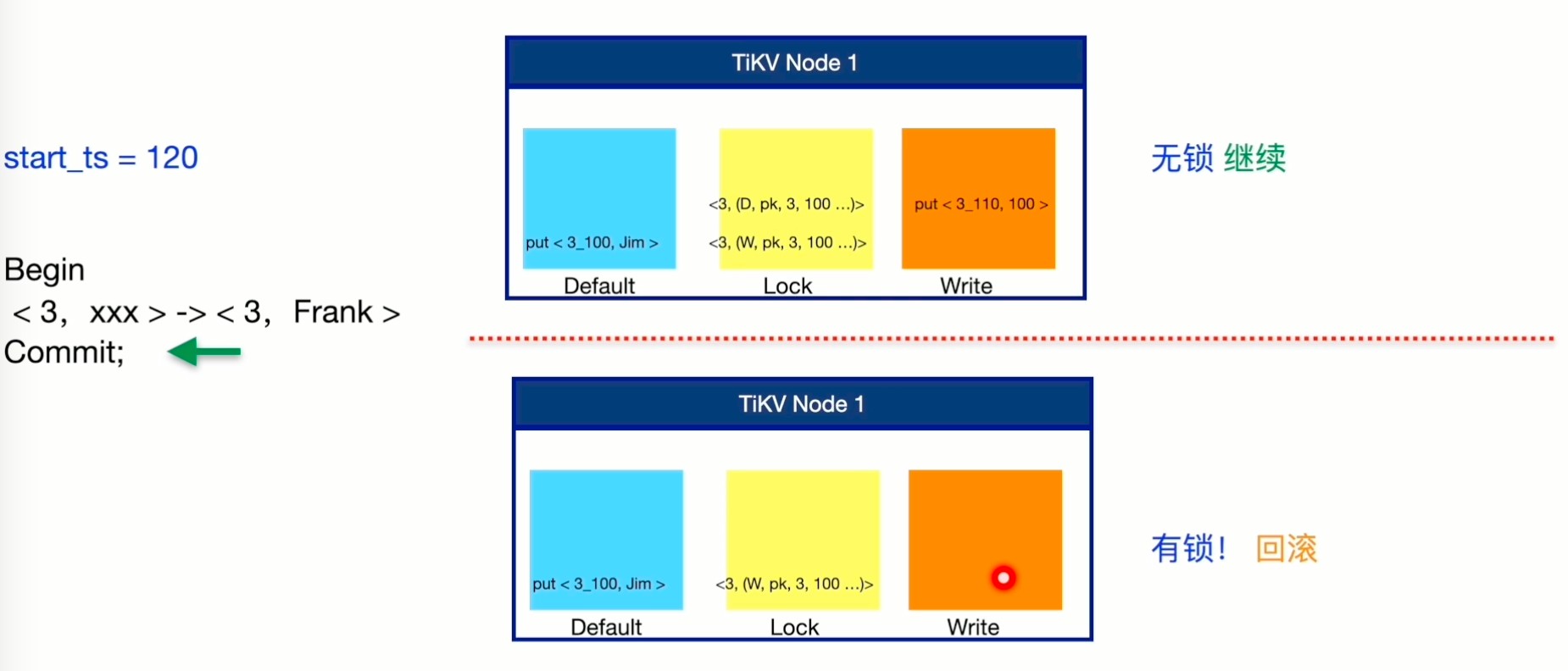
数据读取到memBuffer中



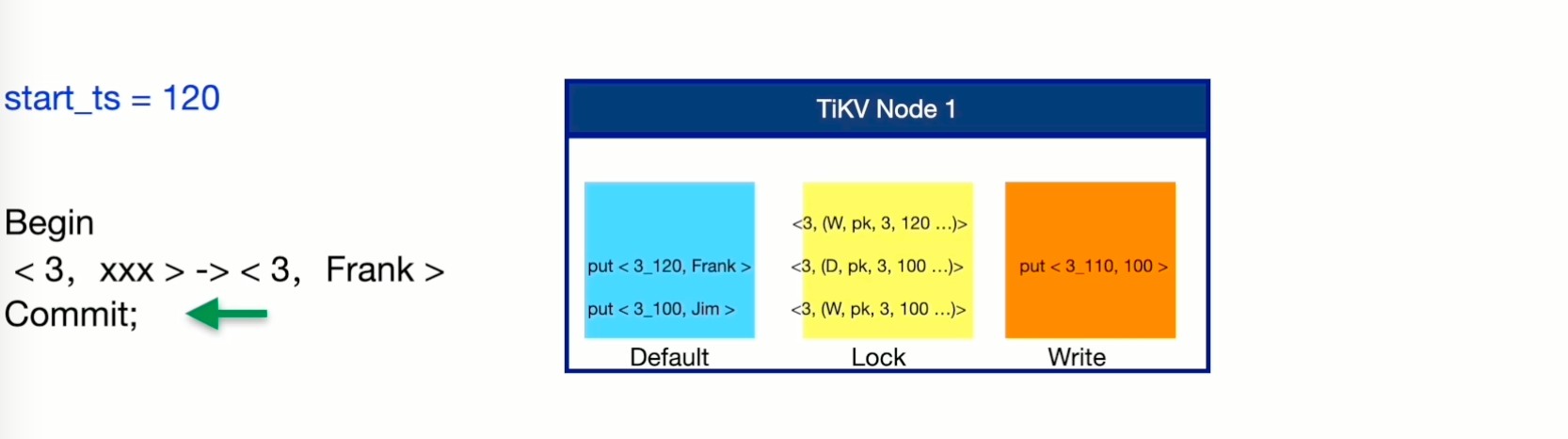
2PC落盘——Prewrite版本检查



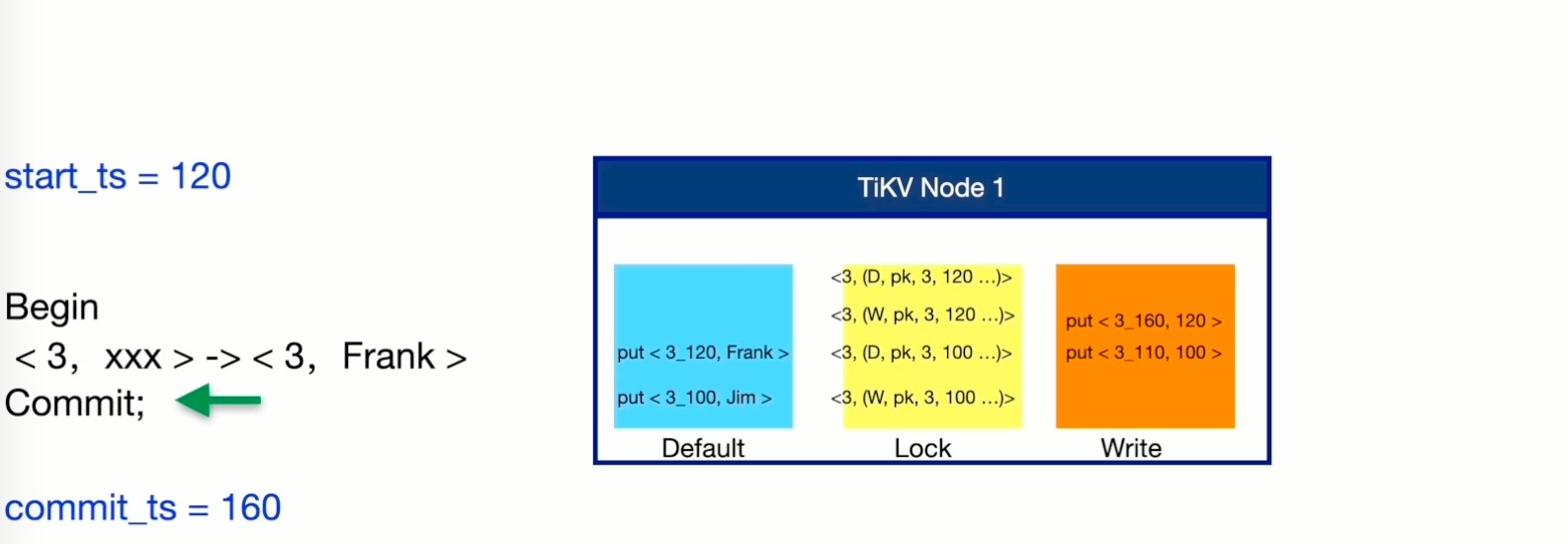
2PC——Prewrite锁冲突监测



2PC——Prewrite加锁

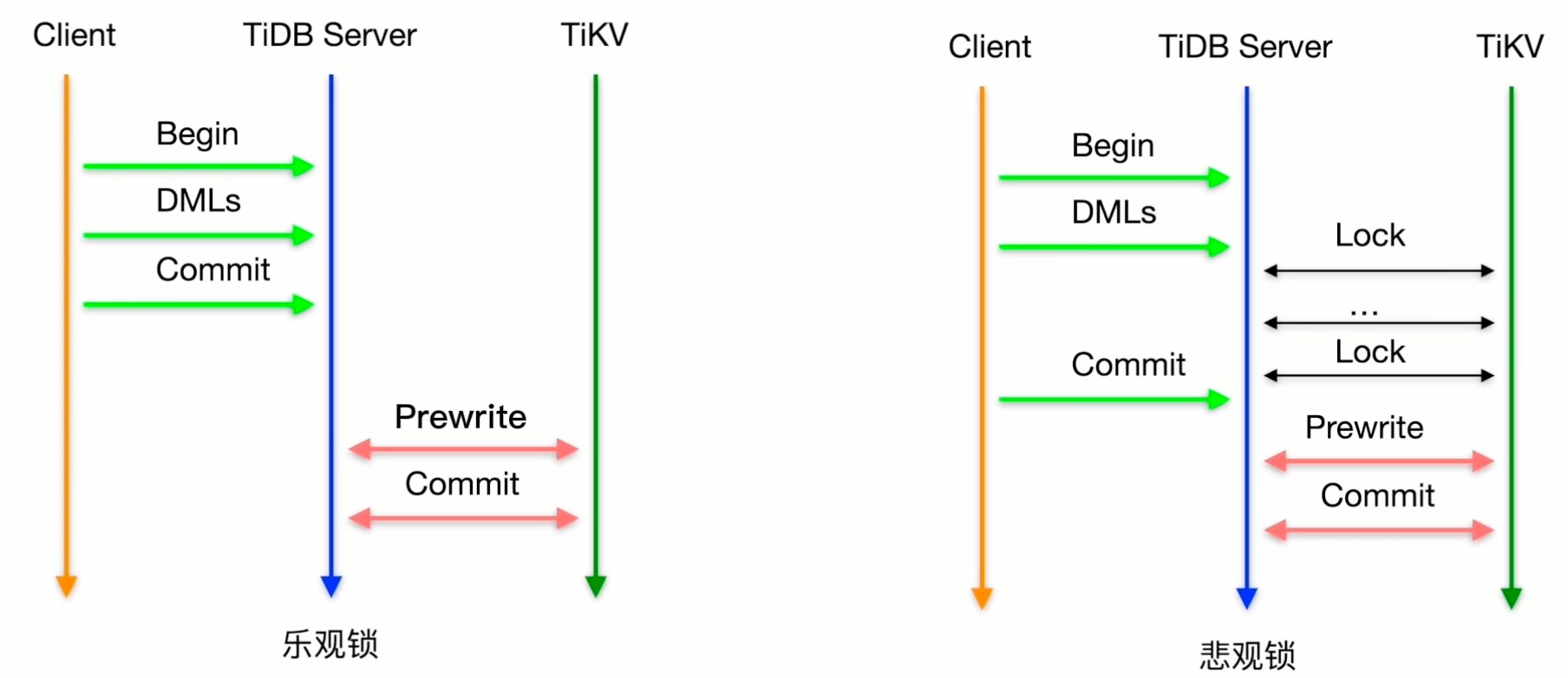


commit锁检查与提交



* + 1. **事务执⾏流程介绍**-**悲观锁模式**

## （1）悲观锁介绍



(2) **悲观锁的实现**

场景:⼤量事务修改在(⼏百万⾏)在mem中修改准备提交,发现已经有锁则回滚代价⾮常⾼,为了避免此代价使⽤悲观锁

特点:在事务数据修改每⼀⾏的时候都⻢上加DML锁,执⾏中发现被锁则⽴即回滚,不⽤等事务全部更改数据准备commit时候在回滚。

1. 在DMl阶段每次都先加lock 到 lock cf,写⼊⼀个占位锁
2. 发现任何写冲突则回滚
3. Prewrite阶段lock占位锁改成正常lock 锁

流程:

1. ⽤户发出请求到tidb server
2. 去PD获取TSO
3. sql语句解析 编译 优化
4. 执⾏--》PD获取路由信息
5. 执⾏器从rocksdb取出-->放进tidb server 的memBuffer修改数据。
   * write版本监测
   * startts = 120,数据进⾏内存修改同时,在lock cf中DML上锁占位符锁 <3,W,pk,0000>
   * 数据内存修改完毕,进⾏2PC落盘

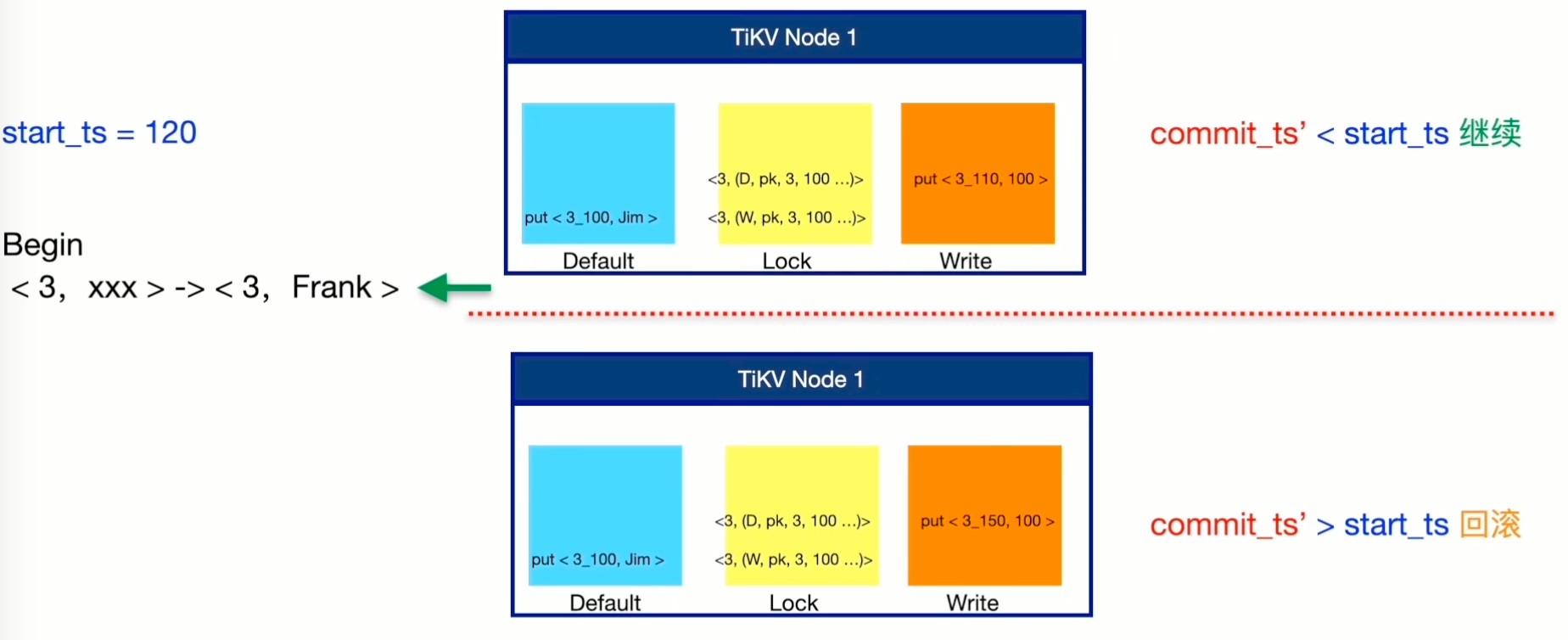
7.TiDB的2PC

* Prewrite阶段:
  + 版本冲突监测:如果有事务已经提交,且是此⾏数据被修改,造成写写冲突(在本事务提交内其他事务提交),看

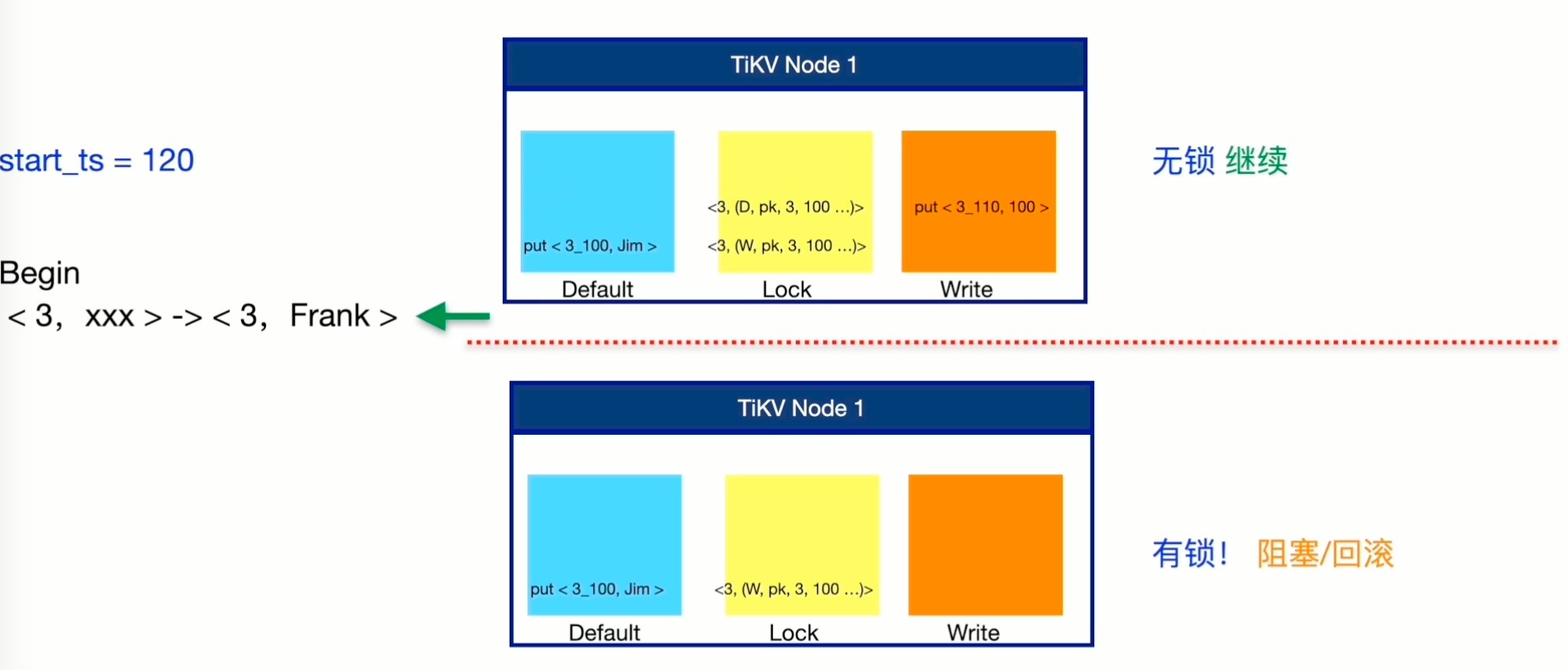
write是否有最新版本信息且⼩于本事务的start ts，⼩于start ts继续,否则回滚

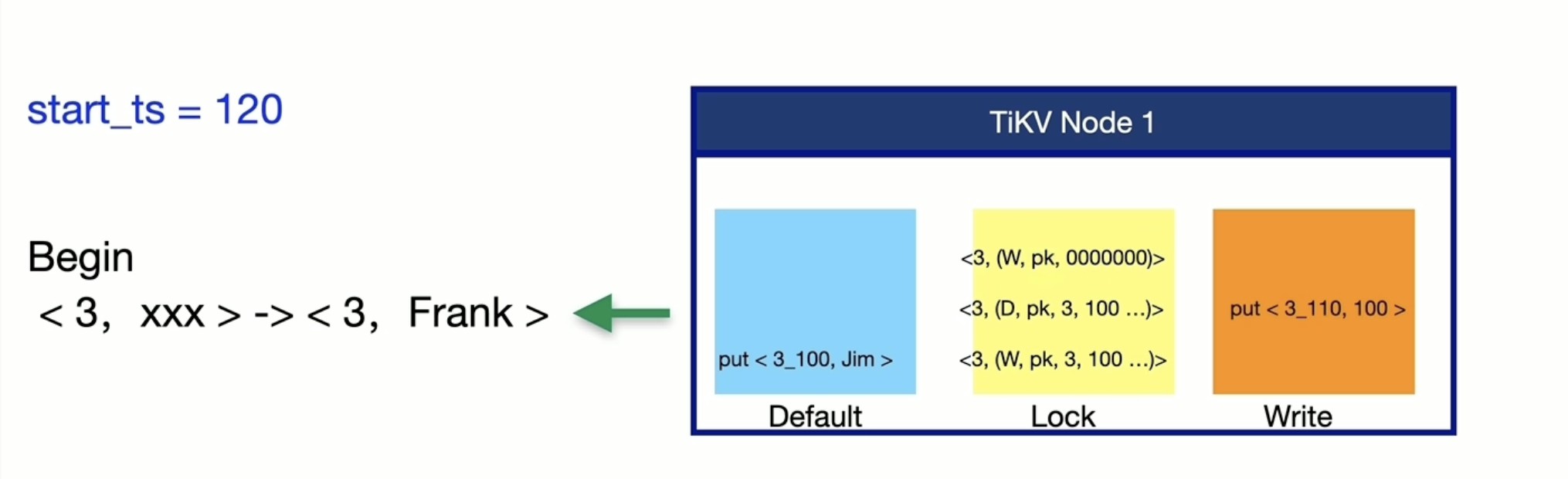
* + 占位锁修改为正常的cf lock 锁
  + 加锁:w
* Commit锁检查和提交
  + 检查锁 获得提交时间TSO
  + 记录最新版本 write cf 3\_160,120,然后清理lock cf <3,D,pk,3,120>
  + 数据落盘rocksdb到sst⽂件

悲观锁——DML上锁:版本检测

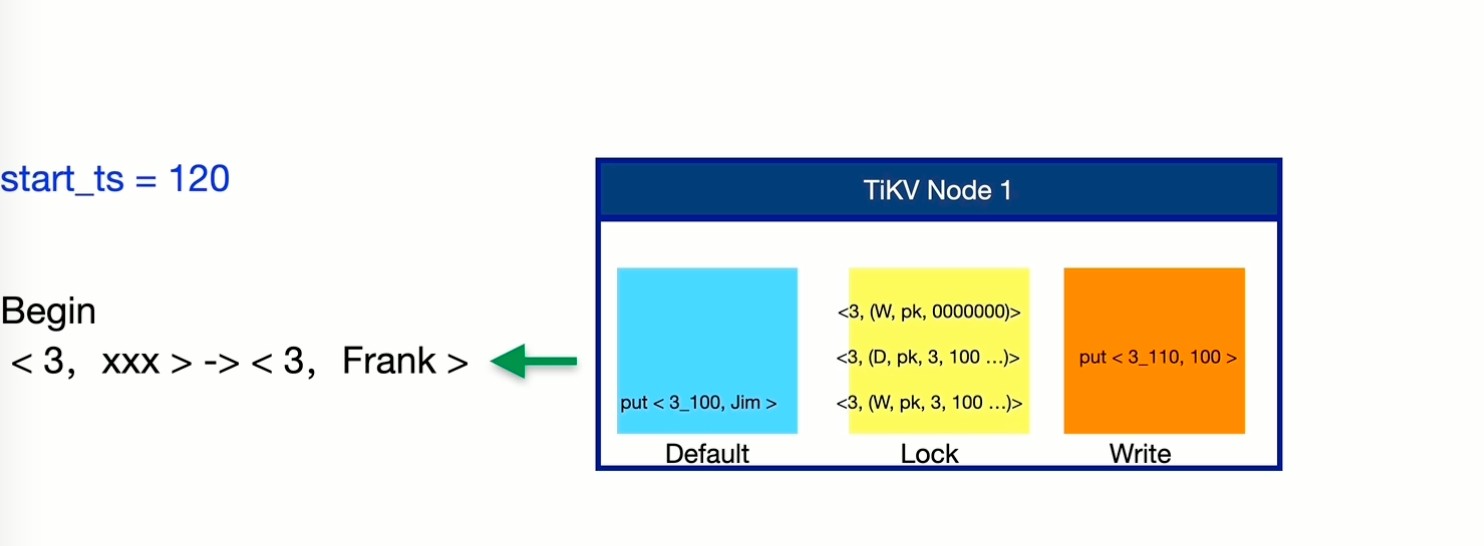


悲观锁——DML上锁:锁检测

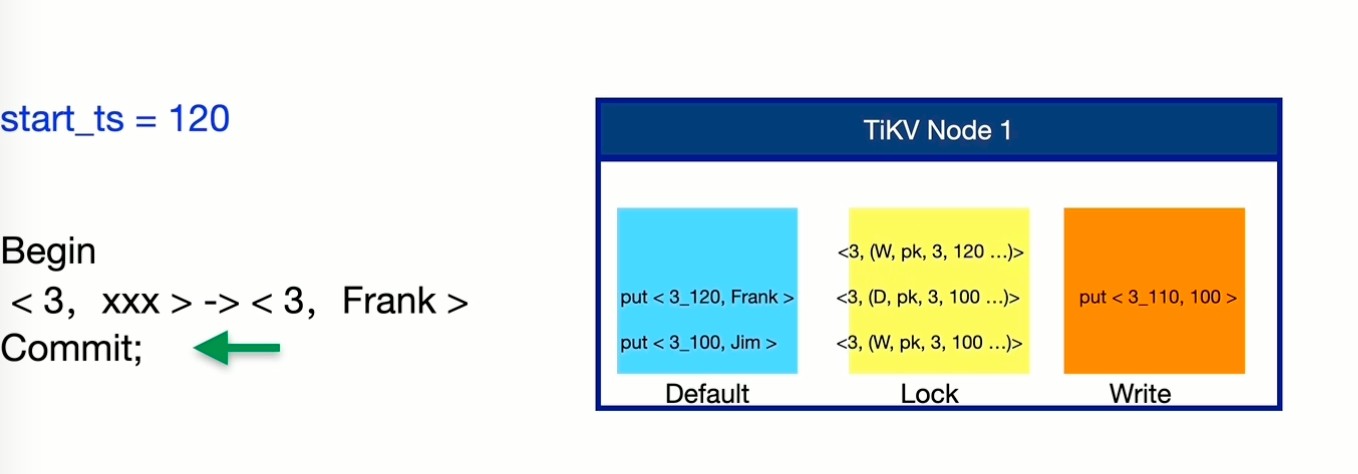




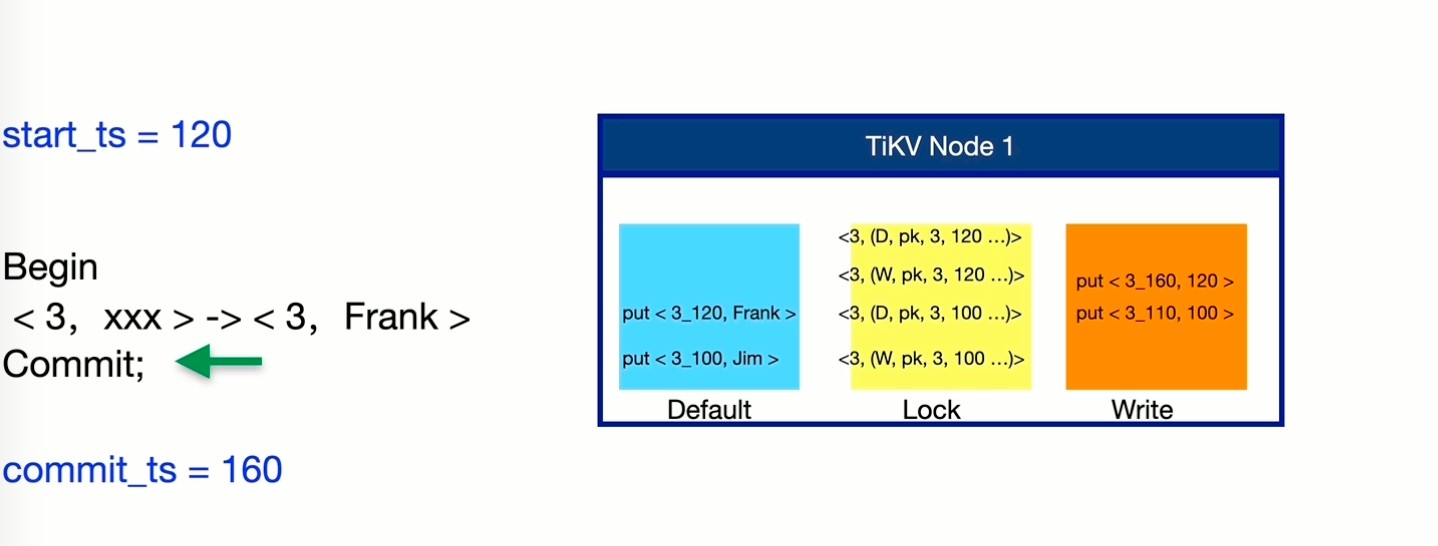
悲观锁——DML上锁



悲观锁——2PC提交,prewrite（补充占位符信息）



悲观锁——2PC, commit



* + 1. **乐观锁和悲观锁的优缺点**

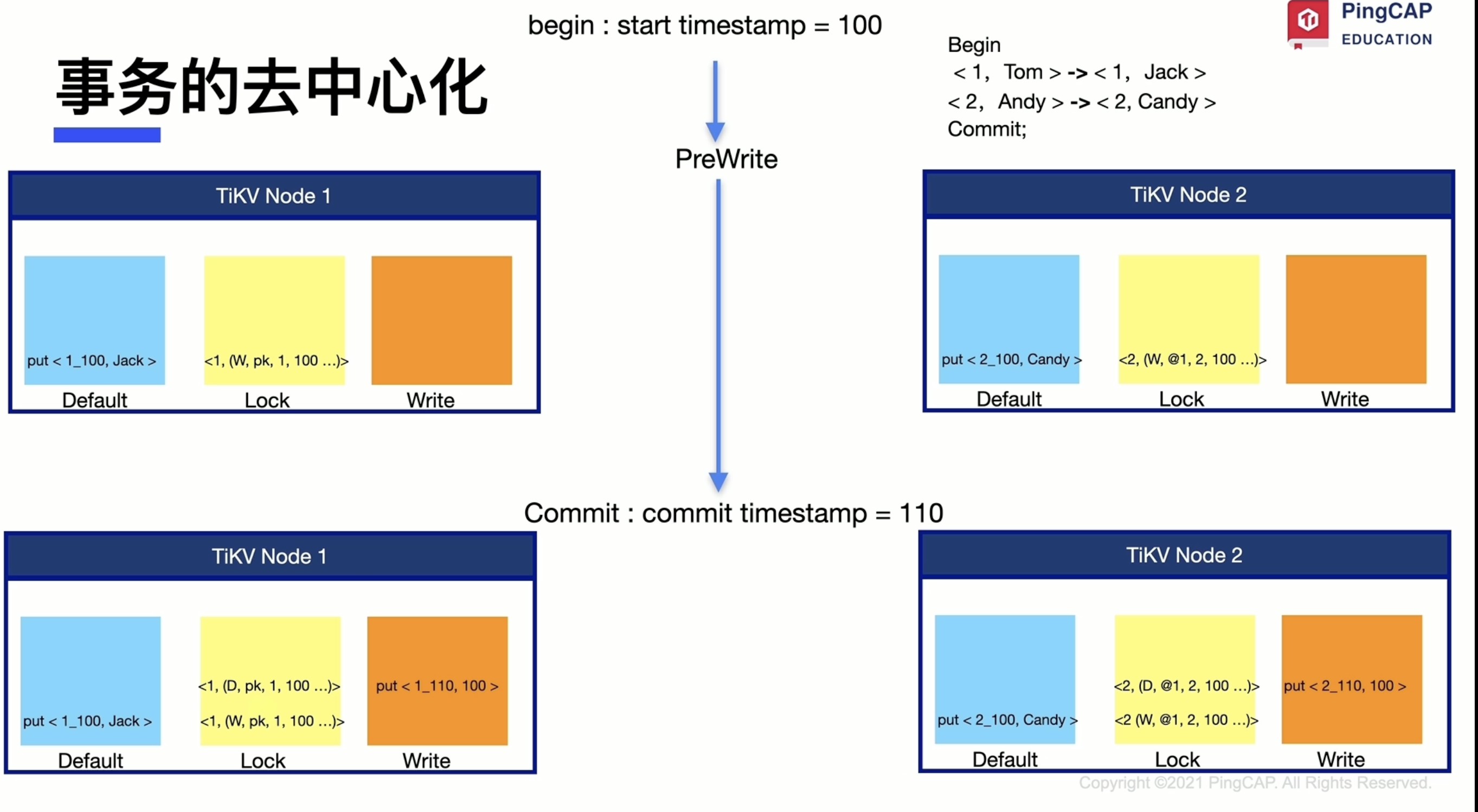
乐观锁：

* 适合冲突率不⾼的场景，在冲突率不⾼的场景处理⾼效
* 需要复杂的应⽤端重试机制

悲观锁：

* + 适合冲突率⾼的场景，提前上锁的代价⼩于事后回滚的代价
  + 能以⽐较低的代价解决多个并发事务互相冲突导致谁也成功不了的场景
  + 悲观事务更直观，更容易实现
  + 在冲突率不⾼的场景并没有乐观事务处理⾼效

# 事务去中⼼化



⼀旦pri key commit,事务则⼀定成功

* 剩下事务异步提交
* 宕机则随时恢复
  1. TiDB**事务优****化** 
     1. **乐观锁和悲观锁设置**

#事务相关参数：

show variables like '%transaction\_isolation%'; show variables like '%tidb\_txn\_mode%';

乐观锁适合⾼并发⽆冲突场景

悲观锁适合写冲突场景业务

#设置全局事务模式 (1)设置悲观锁

SET GLOBAL tidb\_txn\_mode = 'pessimistic';

#显式地开启悲观事务 BEGIN PESSIMISTIC;

BEGIN /\*T! PESSIMISTIC \*/;

（2）设置乐观锁

SET GLOBAL tidb\_txn\_mode = 'optimistic';

#切换事务模式

* 以乐观模式启动事务
  + begin optimistic;
  + set tidb\_txn\_mode='optimistic';
  + begin;
* 在悲观模式下启动事务
  + begin pessimistic;
  + set tidb\_txn\_mode='pessimistic';
  + begin;
* 悲观模式是默认模式

## 乐观锁实验

##参数设置

set global transaction\_isolation='read-committed'; set global tidb\_txn\_mode='optimistic';

use test

create table test\_lock(id int(11) auto\_increment primary key,name varchar(32)); insert into test\_lock values (null,'aaa'),(null,'bbb'),(null,'ccc');

##窗⼝1：

mysql -uroot -p000000 -h 192.168.1.150 -P4000 -D test mysql> begin;

mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 1 | aaa |

| 2 | bbb |

| 3 | ccc |

+----+ +

3 rows in set (0.00 sec)

mysql> update test\_lock set name='www' where id=1; Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from test\_lock; ##本事务内可以看到修改

+----+ +

| id | name |

+----+ +

| 1 | www |

| 2 | bbb |

| 3 | ccc |

+----+ +

3 rows in set (0.00 sec) mysql> commit;

ERROR 9007 (HY000): Write conflict, txnStartTS=437478447686090753, conflictStartTS=437478452889911298, conflictCommitTS=437478483088637954, key=

{tableID=87, handle=1} primary={tableID=87, handle=1}, reason=Optimistic [try again later]

##窗⼝2：

mysql -uroot -p000000 -h 192.168.1.150 -P4000 -D test mysql> begin;

mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 1 | aaa |

| 2 | bbb |

| 3 | ccc |

+----+ +

3 rows in set (0.00 sec)

mysql> delete from test\_lock where id=1; Query OK, 1 row affected (0.00 sec) mysql> commit;

Query OK, 0 rows affected (0.01 sec)

mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 2 | bbb |

| 3 | ccc |

+----+ +

2 rows in set (0.00 sec)

|  |  |
| --- | --- |
| **事务**1 | **事务**2 |
| begin; | begin; |
| update test\_lock set name='www' where id=1; |  |
|  | delete from test\_lock where id=1; |
|  | commit; |
| commit;  ERROR 9007 (HY000): Write conflict, txnStartTS=437478447686090753, conflictStartTS=437478452889911298, conflictCommitTS=437478483088637954, key=  {tableID=87, handle=1} primary={tableID=87, handle=1}, reason=Optimistic [try again later] |  |

## 悲观锁实验

##参数设置

set global transaction\_isolation='read-committed'; set global tidb\_txn\_mode='pessimistic';

use test

create table test\_lock(id int(11) auto\_increment primary key,name varchar(32)); insert into test\_lock values (null,'aaa'),(null,'bbb'),(null,'ccc');

##窗⼝1：

mysql -uroot -p000000 -h 192.168.1.150 -P4000 -D test mysql> begin;

mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 2 | bbb |

| 3 | ccc |

+----+ +

3 rows in set (0.00 sec)

mysql> update test\_lock set name='www' where id=2; Query OK, 1 row affected (0.02 sec)

Rows matched: 1 Changed: 1 Warnings: 0 mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 2 | www |

| 3 | ccc |

+----+ +

2 rows in set (0.00 sec)

##窗⼝2：

mysql -uroot -p000000 -h 192.168.1.150 -P4000 -D test mysql> begin;

mysql> select \* from test\_lock;

+----+ +

| id | name |

+----+ +

| 2 | bbb |

| 3 | ccc |

+----+ +

3 rows in set (0.00 sec)

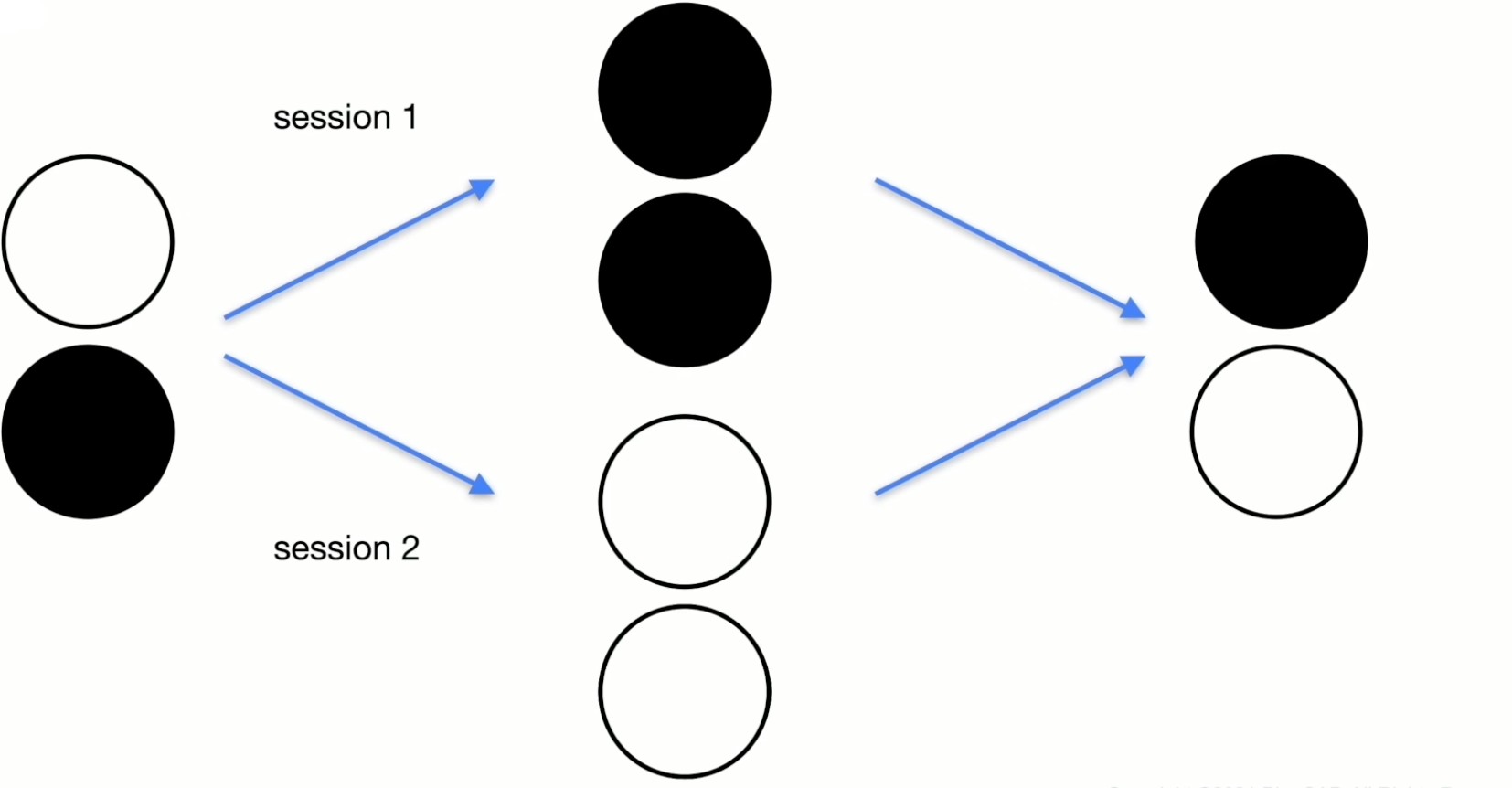
mysql> delete from test\_lock where id=2; -- 阻塞，直到窗⼝1的执⾏完成，则成功，若等待时间⻓，则返回超时报错。

|  |  |
| --- | --- |
| **事务**1 | **事务**2 |
| begin; | begin; |
| update test\_lock set name='www' where id=2; |  |
|  | mysql> delete from test\_lock where id=2; --阻塞 |
| commit; | 冲突解除，delete执⾏成功 |
|  | commit; |

* + 1. **写偏斜**write skew

定义：

SI 隔离级别下，两个并发的事务读取了不同但相关的记录，接着这两个事务各⾃更新了⾃⼰读到的数据，并最终都提交了事务，如果这些相关的记录之间存在着不能被多个事务并发修改的约束，那么最终结果将是违反约束的。



## 写偏斜示例

## 设置隔离级别

set global transaction\_isolation='repeatable-read'; use test

create table test\_ws(id int(11) auto\_increment primary key,name varchar(32)); insert into test\_ws values (null,'⽩⾊'),(null,'⿊⾊');

##session 1:

begin;

mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⽩⾊ |

| 2 | ⿊⾊ |

+----+ +

2 rows in set (0.00 sec) #session1吧⽩⾊改为⿊⾊

mysql> update test\_ws set name='⿊⾊' where id=1;

Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⿊⾊ |

| 2 | ⿊⾊ |

+----+ +

2 rows in set (0.00 sec)

##session2:

begin;

mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⽩⾊ |

| 2 | ⿊⾊ |

+----+ +

2 rows in set (0.00 sec)

#session1提交后，在session2 查看表的信息依然是⼀⿊⼀⽩

#session2吧⿊⾊改为⽩⾊

mysql> update test\_ws set name='⽩⾊' where id=2;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⽩⾊ |

| 2 | ⽩⾊ |

+----+ +

2 rows in set (0.00 sec) mysql> commit;

Query OK, 0 rows affected (0.01 sec)

mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⿊⾊ |

| 2 | ⽩⾊ |

+----+ +

2 rows in set (0.00 sec)

|  |  |
| --- | --- |
| **事务**1 | **事务**2 |
| begin; | begin; |
| select \* from test\_ws; | select \* from test\_ws; |
| update test\_ws set name='⿊⾊' where id=1; |  |
| commit; | select \* from test\_ws; |
|  | update test\_ws set name='⽩⾊' where id=2; |
|  | commit; |

1. **解决写偏斜**

SELECT FOR UPDATE 可以克服写偏斜问题

## session1： begin;

mysql> select \* from test\_ws;

mysql> update test\_ws set name='⿊⾊' where id=2; mysql> select \* from test\_ws;

+----+ +

| id | name |

+----+ +

| 1 | ⿊⾊ |

| 2 | ⿊⾊ |

+----+ +

2 rows in set (0.00 sec) mysql> commit;

## session2：

begin;

mysql> select \* from test\_ws for update; #使⽤for update来读取当前的值

+----+ +

| id | name |

+----+ +

| 1 | ⿊⾊ |

| 2 | ⿊⾊ |

+----+ +

2 rows in set (0.00 sec)

|  |  |
| --- | --- |
| **事务**1 | **事务**2 |
| begin; | begin; |
| select \* from test\_ws; | select \* from test\_ws; |
| update test\_ws set name='⿊⾊' where id=2; |  |
| commit; |  |
|  | select \* from test\_ws for update; |
|  | commit; |

* + 1. **事务⼤⼩限制**
* 最⼤单⾏记录容量为 120MB
* ⽀持的最⼤单个事务容量为 10GB
  + 1. **事务错误处理**

1、死锁

ERROR 1213: Deadlock found when trying to get lock; try restarting transaction

解决⽅案：

* 调整业务逻辑或者 Schema 设计，从应⽤层⾯避免死锁
* 减⼩事务粒度
* 使⽤乐观事务
* 重试