

# 数据库事务 V2

2021.10.29

# 主要内容

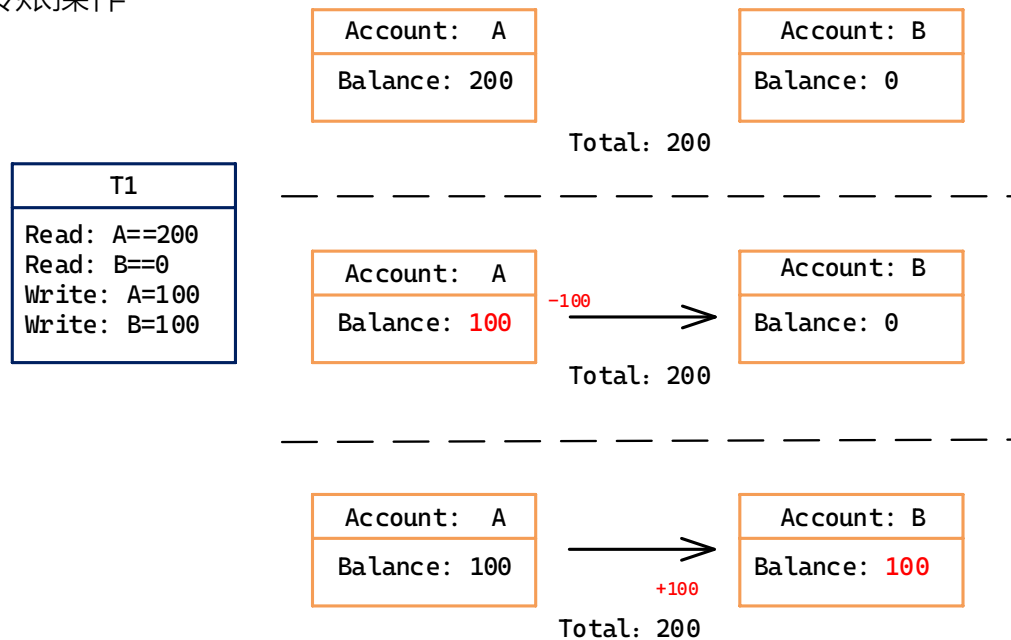
---

- 事务引入
- 单机事务实现及优化
- 分布式事务实现及优化

# 事务基础

## ➤ 事务的引入:

经典例子：两个人之间转账操作



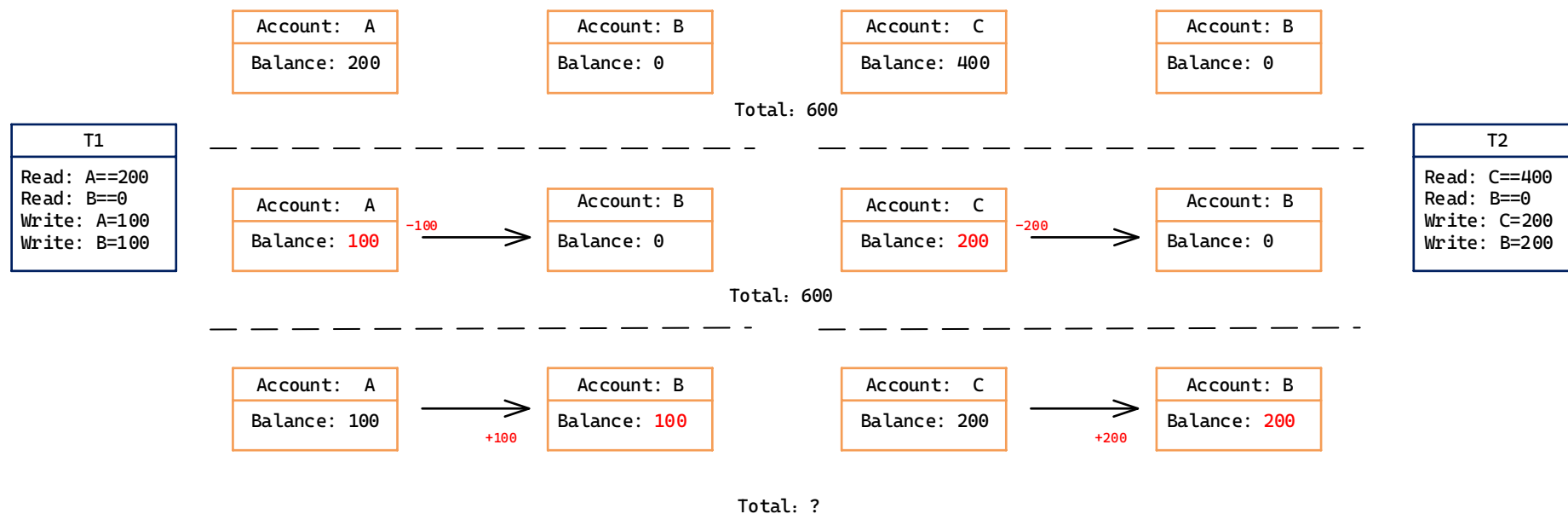
如果要保证正确性，需要满足什么条件？

1. 步骤1和2必须同时成功 或 同时失败
2. 转账操作必须是永久性的

# 事务基础

## ➤ 事务的引入:

例子: 多人同时转账



如果要保证正确性, 需要满足什么条件?

1. 步骤1和2必须同时成功 或 同时失败
2. 转账操作必须是永久性的
3. 多个同时进行的操作之间不能相互影响

—— 一致性 C

—— 原子性 A

—— 持久性 D

—— 隔离性 I

# 事务基础

---

- 事务：是数据库执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。
- 特性：ACID
- 作用
  - 保证数据库的正确性
  - 简化开发人员的工作

# 事务基础

---

## ➤ 保证ACID的方法

### 1. 原子性

Write Ahead Log(WAL)

### 2. 持久性

WAL、非易失性存储

### 3. 隔离性

并发控制算法

### 4. 一致性

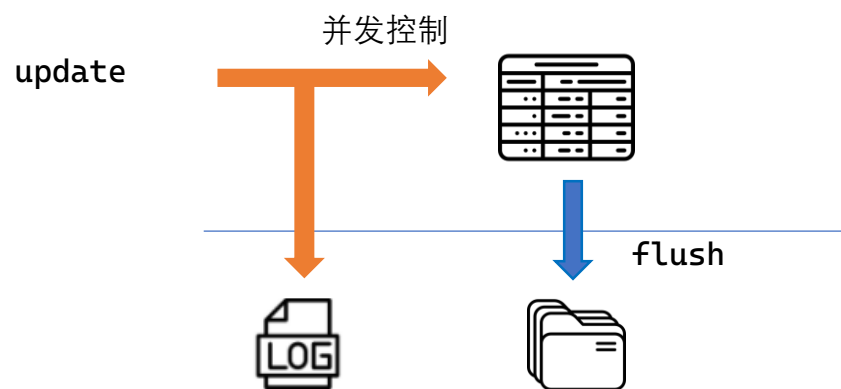
原子性 + 持久性 + 隔离性 ➡ 一致性

# 事务基础

---

因此，一个事务基本的处理流程为：

1. 先写日志
2. 并发控制
3. 修改buffer中的数据
4. 将buffer中的数据同步到disk上



# 事务的特性

---

## ➤ 原子性与持久性:

目前应用最广泛的Write Ahead Log (WAL) 机制

*ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*

基本思想:

1. 对任何更新操作, 先记录日志。将更新写回磁盘之前, 必须先把日志写到稳定的存储器上
2. 一个事务的所有日志 (包括commit日志) 写入磁盘后才可以被提交

LSN: [prevLSN, TransID, "update", pageID, redo Info, undo Info]

- LSN (Log Sequence Number) 每条日志拥有一个全局唯一单调递增的LSN。
- PrevLSN: 当前事务的前一条日志的LSN。如果已经是第一条日志, 那么PrevLSN为空
- TransID: 产生这条日志的事务ID
- PageID: 对应的更新操作作用的页面ID
- redo Info: 记录如何redo这次更更新的操作
- undo Info: 记录如何undo这次更新的操作



# 事务的特性

► 原子性与持久性:

## 目前应用最广泛的Write Ahead Log (WAL) 机制

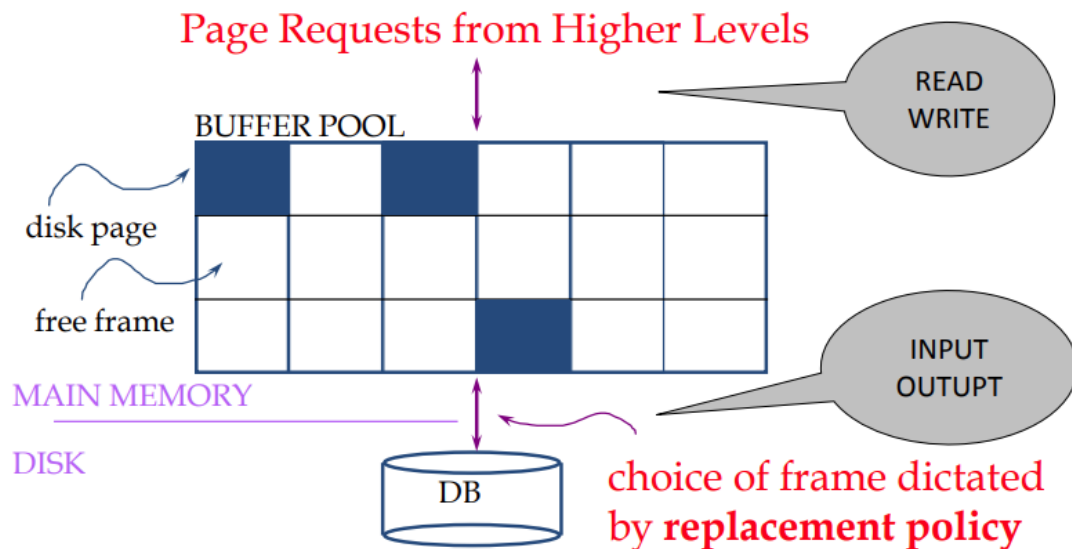
*ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*

每个page: 记录最后一次修改操作对应的LSN

记录当前已经刷盘的LSN: flushLSN

当且仅当:  $\text{pageLSN} \leq \text{flushLSN}$

该页面才能被写回到磁盘中



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained

# 事务的特性

## ➤ 原子性与持久性:

- undo和redo的设计, 与当前buffer采用的steal和no-force刷盘策略密不可分

	No Steal	Steal
No Force		Fastest
Force	Slowest	

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

# 事务的特性

---

## ➤ 并发控制

### ➤ 基本的并发控制协议

- 基于锁：2PL
- 时间戳：TO
- 乐观并发控制

### ➤ 单版本 → 多版本（读操作不会被阻塞）

- MV2PL
- MVTO：每个数据项有读、写两种时戳；每个事务有开始时戳
- SI/SSI/WSI：每个数据项有创建时戳，每个事务有开始、提交两种时间戳

# 事务的特性

---

## 快照隔离 SI

- 提出: A Critique of ANSI SQL Isolation Levels SIGMOD'95
- 检测并发执行事务之间的写写冲突, 存在写偏序异常
- 时间戳:
  - Start – Timestamp*: 事务中第一个操作执行之前的任意时间点
  - Commit – Timestamp*: 事务执行完所有操作准备提交时的时间戳
- 冲突: 检测写写冲突
  - 空间上: 两个事务 $T_i, T_j$ 修改同一数据项
  - 时间上:  $T_s(T_i) < T_c(T_j)$  且  $T_s(T_j) < T_c(T_i)$

## 事务的特性

## 隔离级别

[illegible]

# 单机事务优化

---

## ➤ 对单机事务的优化

### 1. 提高写盘速度，加快事务的提交

NVM的出现，影响了当前DRAM+HDD/SSD架构

### 2. 改进并发控制协议，加快事务的处理

2.1 对协议本身的优化

2.2 同时使用多个并发控制协议

2.3 减少MVCC带来的副作用

# 单机事务优化

---

## ➤ 基于NVM的优化

- 基本思路：

- 1、 NVM直接替换之前的DRAM+ssd/hdd

1. 直接避开 WAL+Buffer

2. 成本高

3. 使用寿命有限

4. NVM速度慢于DRAM

- 2、 与当前结构结合

Write-behind logging

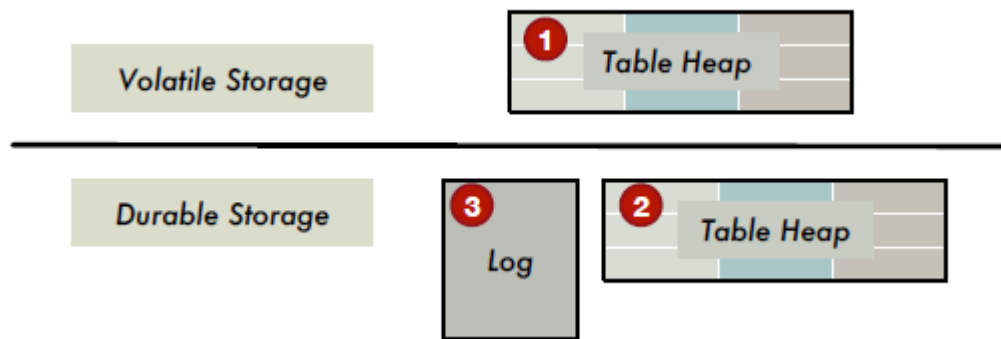
# 单机事务优化

## ➤ Write-behind logging

### ● 基本设计:

1. 使用NVM设备替换磁盘, 保留DRAM
2. 使用MVCC并发控制, NVM上保留多个版本
3. 提高刷盘频率, 数据先于日志刷盘
4. 简化log, 只保留时间戳, 减少写入数据量

Checksum	LSN	Log Record Type	Persisted Commit Timestamp( $C_p$ )	Dirty Commit Timestamp( $C_d$ )
----------	-----	-----------------	-------------------------------------	---------------------------------





# 单机事务优化

## ➤ Write-behind logging

### ● 日志格式



Checksum	LSN	Log Record Type	Persisted Commit Timestamp( $C_p$ )	Dirty Commit Timestamp( $C_d$ )
----------	-----	-----------------	-------------------------------------	---------------------------------

$C_p$ : 该时戳之前的事务均已提交

$C_d$ :  $(C_p, C_d)$ 时戳范围内的事务, 均未提交

} 每次提交时计算

### ● 设计思想:

1. 数据先于日志刷盘  日志中不需要保留redo信息
2. 为每个数据保存多个版本  宕机恢复时, 可通过时间戳屏蔽未提交的版本, 实现undo操作
3. 按批提交, 本次提交 $C_p$ 之前的事务,  $(C_p, C_d)$ 之间已完成的事务, 等到下一批次提交

# 单机事务优化

---

## ➤ 对并发控制协议优化

### 1. 优化协议的实现方式:

1. 乐观并发控制: SI -> SSI -> WSI
2. 悲观并发控制: Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking

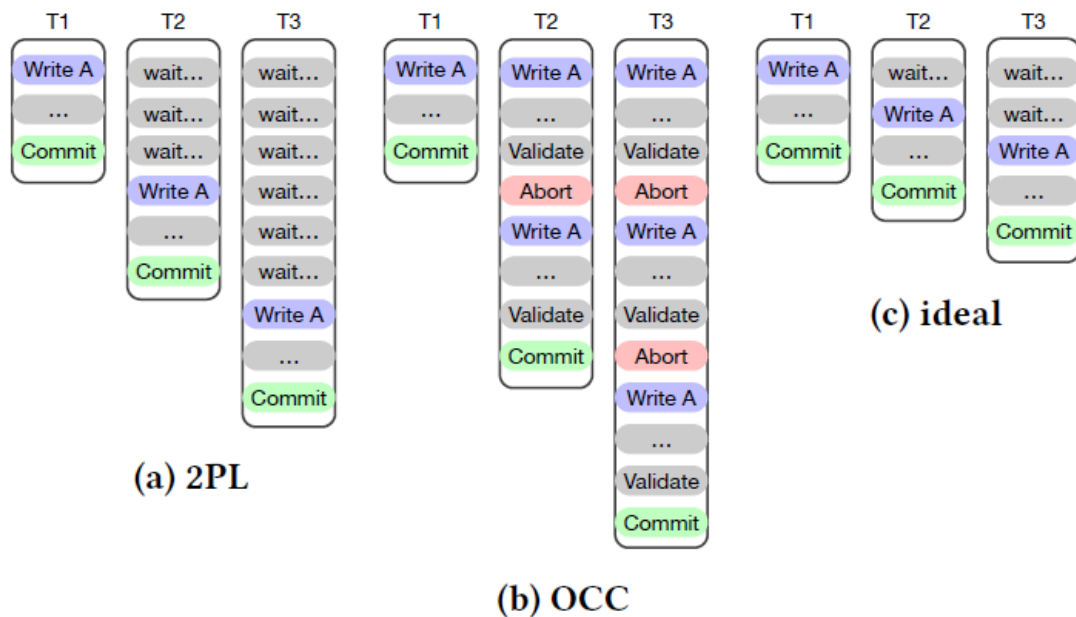
### 2. 不同的负载使用不同的协议: **Adaptive Optimistic Concurrency Control for Heterogeneous Workloads**

### 3. 对MVCC的过期版本回收方式进行优化: Scalable Garbage Collection for In-Memory MVCC Systems

# 单机事务优化

## ➤ 优化协议：提前解锁的时机

Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking



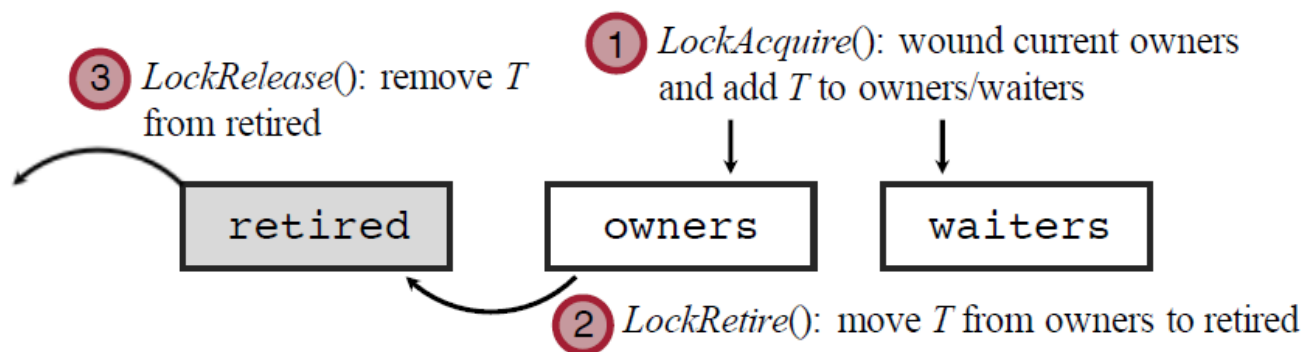
问题现象：对数据库中的某些热点数据，每个事务可能只做一次修改，但该item上加的锁，还是得等事务提交后才能释放，增加了其他事务的等待时间

基本思想：通过追踪脏读依赖关系，允许部分的脏读，提前释放锁

# 单机事务优化

## ➤ 优化协议：提前解锁的时机

Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking



每个lock拥有三个list:

1. owners: 记录当前持有锁的事务id
2. waiters: 记录等待锁的事务id
3. retired: 提前释放锁的事务id, 按插入顺序排序

依赖关系:

1. retired list中, 后插入的事务依赖先插入的事务
2. owners list中的事务依赖retired list中的事务

通过信号量机制, 保证有依赖关系的事务按顺序提交

# 单机事务优化

---

## ➤ 使用多种协议

Adaptive Optimistic Concurrency Control for Heterogeneous Workloads

提出OCC的两种验证方法，根据不同负载，使用不同的协议

1. Local read-set validation(LRV):

检查读集合是否被其他事务修改

2. Global write-set validation(GWV):

检查其他事务的写集合是否满足该事务的查询谓词

# 单机事务优化

## ➤ 使用多种协议

Adaptive Optimistic Concurrency Control for Heterogeneous Workloads

提出OCC的两种验证方法，根据不同负载，使用不同的协议

1. Local read-set validation(LRV): ——适合单点查询

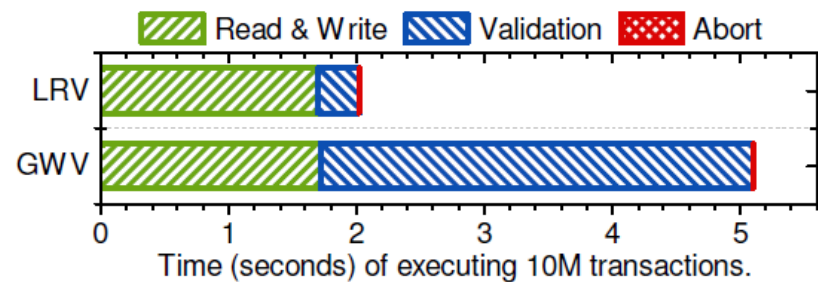
检查读集合是否被其他事务修改

2. Global write-set validation(GWV): ——范围扫描

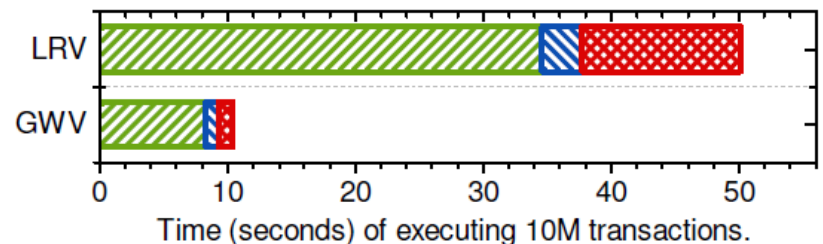
检查其他事务的写集合是否满足该事务的查询谓词

- 两种验证方式的切换

事务在执行前确定其验证类型



(a) 80% reads, 20% writes, low contention ( $\theta = 0.6$ ), 32 worker threads.



(b) 80% reads, 10% writes, 10% scans (length = 800), high contention ( $\theta = 0.9$ ), 32 worker threads

# 单机事务优化

## ➤ 使用多种协议

Adaptive Optimistic Concurrency Control for Heterogeneous Workloads

- 每个事务维护四个集合：

- ReadSet —— LRV  $\langle \text{ros}, \text{ts} \rangle$
- PredicateSet —— GWV  $\langle \text{type}, \text{COLs}, \text{set} \rangle$
- ReadSet and PredicateSet —— LRV 和 GWV
- Writeset

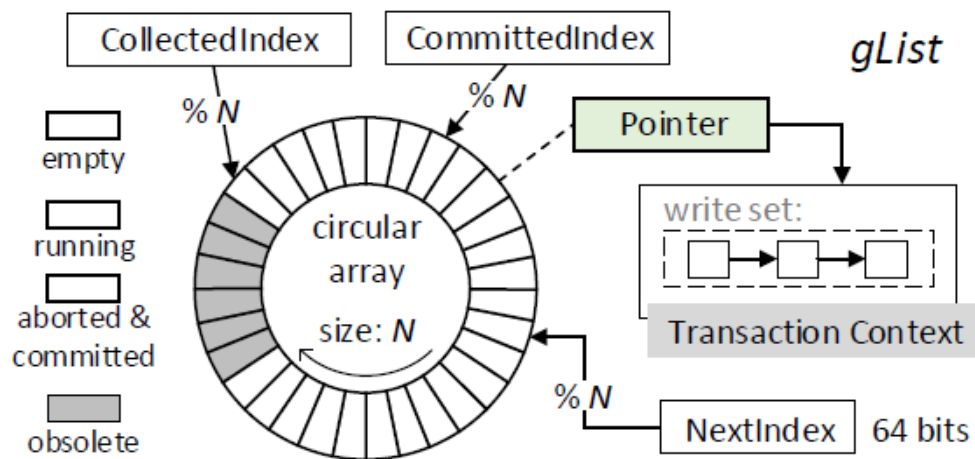
- 使用环形数组 gList 记录并发执行的事务

- $[\text{sIndex}, \text{eIndex}]$

```
SELECT * FROM t WHERE (CA>=a and CB<=b) or CD=d;
```

$\langle P_{\text{no\_readset}}, \{ \langle C_A, a, \text{MAX} \rangle, \langle C_B, \text{MIN}, b \rangle \} \rangle, \text{null} \rangle$   
 $\langle P_{\text{no\_readset}}, \{ \langle C_D, d, d \rangle \}, \text{null} \rangle$

$P_{\text{no\_readset}}$

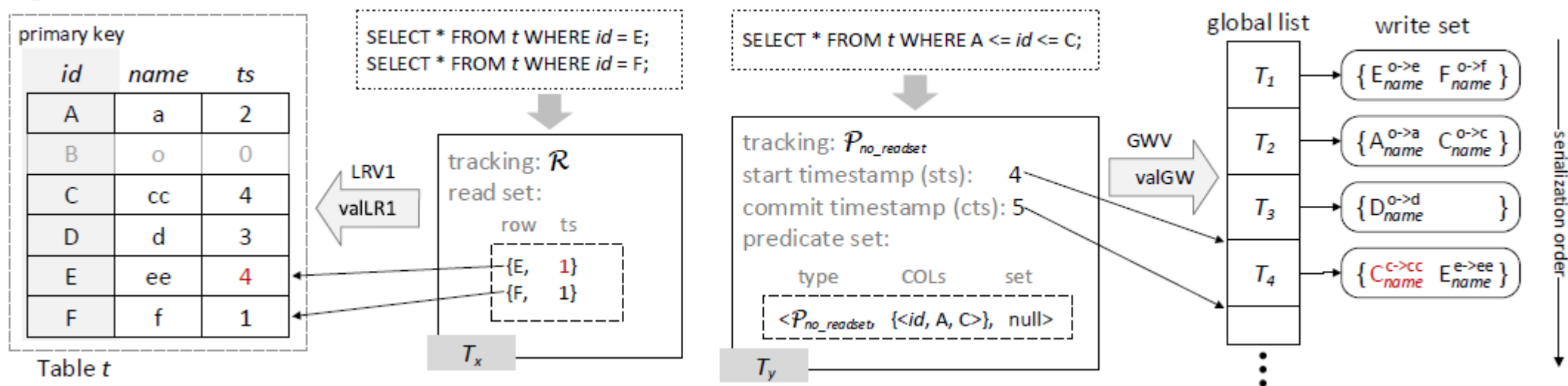


# 单机事务优化

## ➤ 使用多种协议

Adaptive Optimistic Concurrency Control for Heterogeneous Workloads

Example:





# 单机事务优化

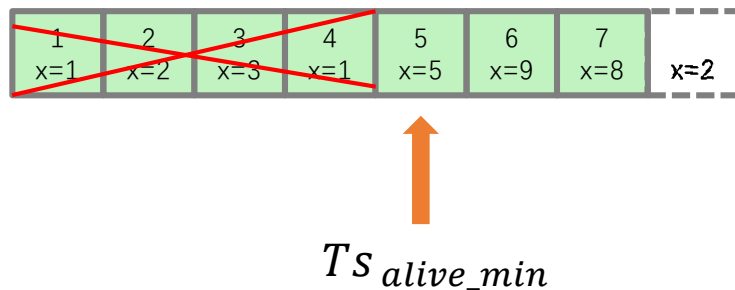
## ➤ 降低协议副作用

MVCC 在执行过程中，会创建多个数据版本，随着事务的运行，过期的版本需要被回收

MVCC过期版本回收：

1. 基本原则：当前版本不会被任何事务访问到时，便可进行垃圾回收
2. 基本方法：

如果存在早于当前运行中的事务最小开始时戳的数据版本，则该版本之前的所有版本均可清理

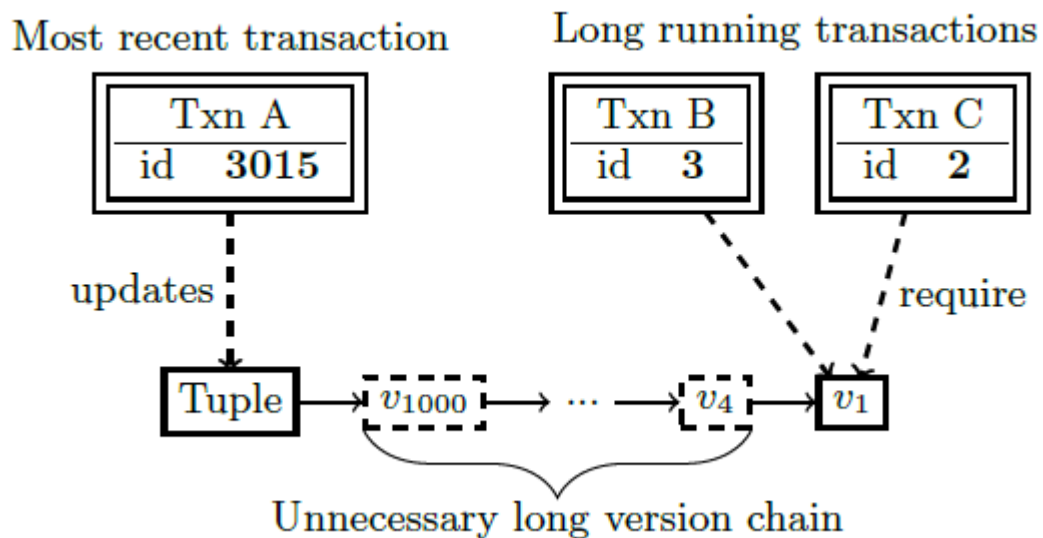


# 单机事务优化

## ➤ 降低协议副作用

回收基本方法：如果存在早于当前运行中的事务最小开始时戳的数据版本，则该版本之前的所有版本均可清理

存在的问题：长事务阻止垃圾回收，导致空间浪费



# 单机事务优化

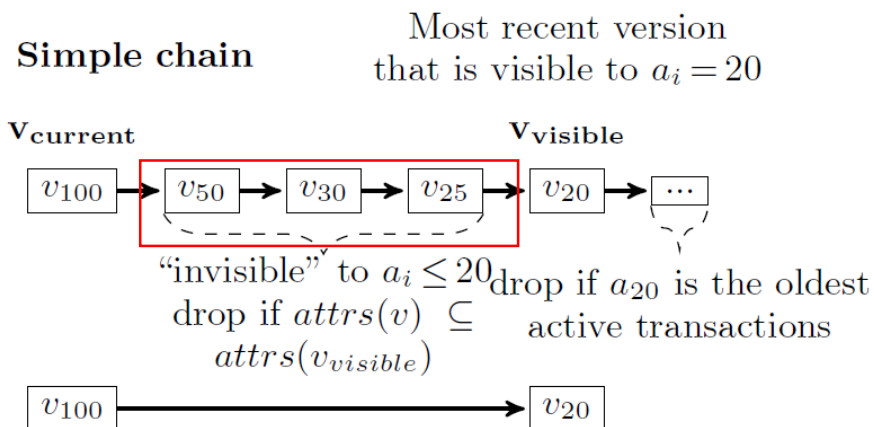
## ➤ 降低协议副作用

Scalable Garbage Collection for In-Memory MVCC Systems

改进思想：在扫描操作过程中，检查当前版本是否会被访问，若没有事务可以访问到，则立即清除

通过两个链表管理事务，按事务开始时戳排序：

1. active transactions list
2. committed transactions list



---

```
input: active timestamps  $A$  (sorted)
output: pruned version chain

 $v_{current} \leftarrow getFirstVersion(chain)$ 
for  $a_i$  in  $A$ 
     $v_{visible} \leftarrow retrieveVisibleVersion(a_i, chain)$ 
    // prune obsolete in-between versions
    for  $v$  in  $(v_{current}, v_{visible})$ 
        // ensure that the final version covers all attributes
        if  $attrs(v) \not\subseteq attrs(v_{visible})$ 
            merge( $v, v_{visible}$ )
            chain.remove( $v$ )
    // update current version iterator
     $v_{current} \leftarrow v_{visible}$ 
```

---

# 小结

---

## ➤ 对单机事务的优化

### 1. 提高写盘速度，加快事务的提交

NVM的出现，影响了当前DRAM+HDD/SSD架构

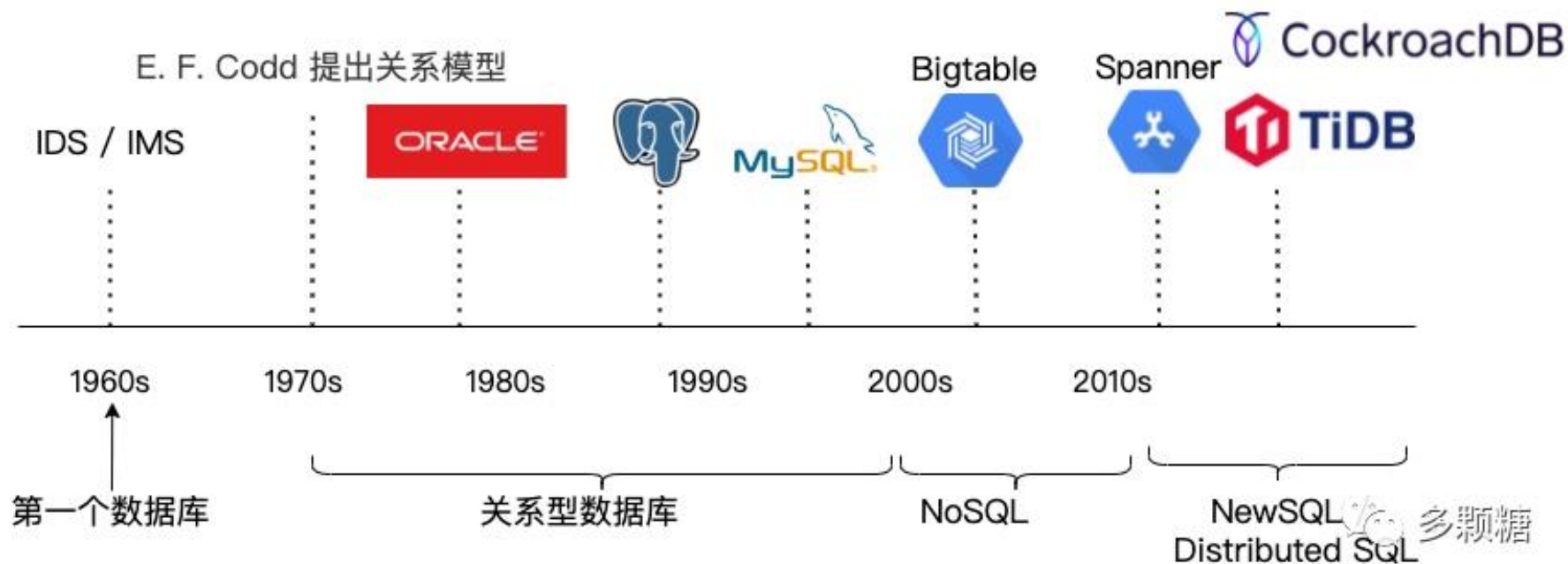
### 2. 改进并发控制协议，加快事务的处理

2.1 对协议本身的优化

2.2 同时使用多个并发控制协议

2.3 减少MVCC带来的副作用

# 分布式数据库



分布式关系型数据库



分布式事务

# 分布式事务基础

---

## ➤ 分布式数据库带来的改变

分布式数据库：

1. 数据分散在多台机器

数据如何划分、管理

2. 分布式系统的CAP定理

如何保证节点可用性，还有数据的一致性

分布式事务：

1. 一个事务可能涉及到多台server上

多台机器的情况下，事务的原子性如何满足

2. 多台机器同时进行事务处理

分布式场景下，如何保证事务之间的偏序关系

# 分布式事务基础

## ➤ 解决方法

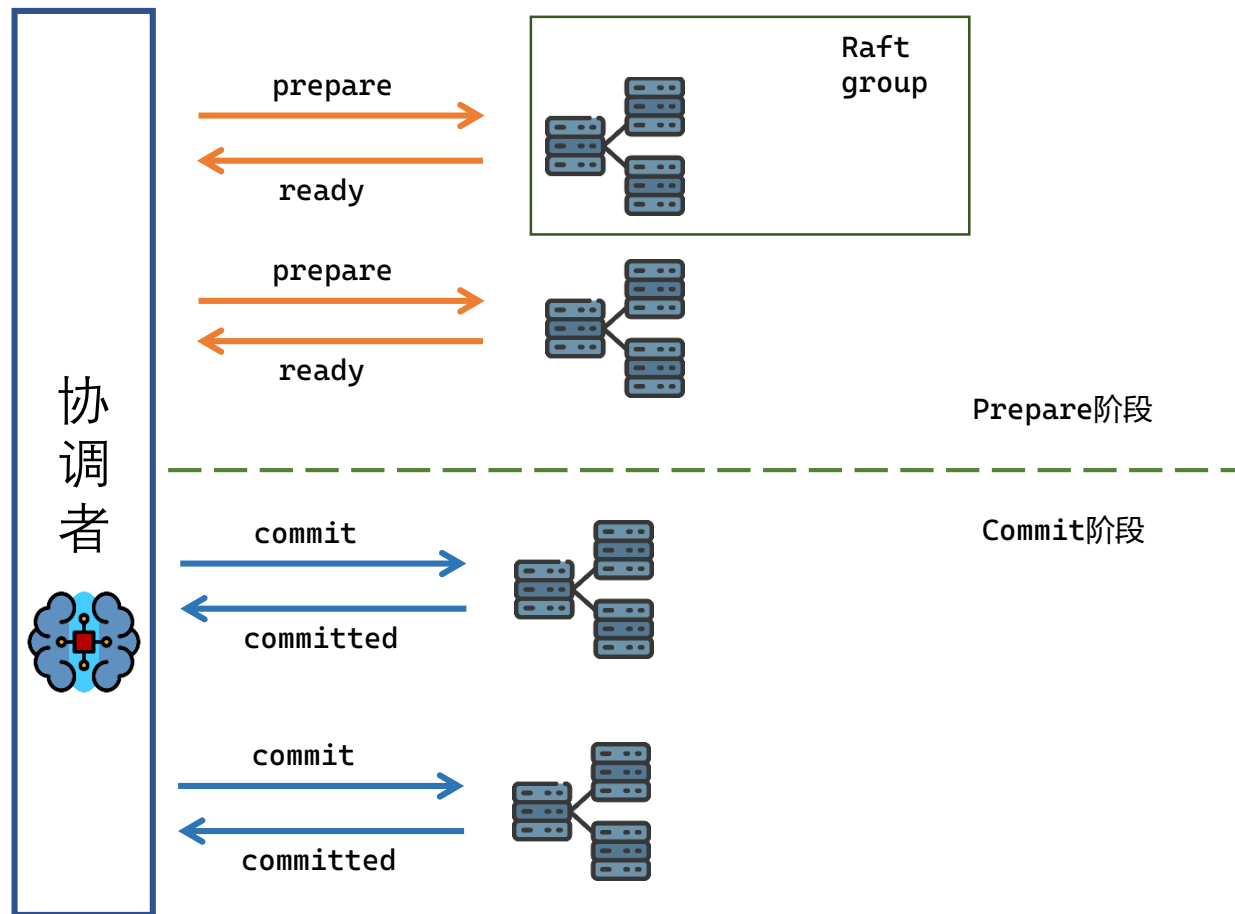
分布式数据库：

1. 数据分散在多台机器  
基于数据key的range划分数据
2. 分布式系统的CAP定理  
Raft协议

分布式事务：

1. 一个事务可能涉及到多台server上  
两阶段提交协议
2. 多台机器同时进行事务处理  
中心时间服务器 TSO

两阶段提交协议(2PC) + Raft



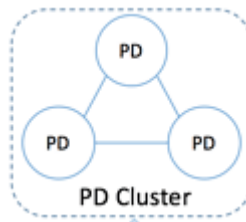
# 分布式事务

## ➤ 时间戳问题

### 1. 使用全局的中心时戳服务器、

TiDB: PD负责全局的时钟管理

- PD使用raft保证高可用性
- 以3s为周期，持久化当前时间+3s的时间戳



### 2. 硬件支持：原子钟

Spanner——存在较短的不确定性窗口，等待不确定性窗口时间过去

### 3. 混合逻辑时钟：HLC

CockroachDB——不需要特殊的硬件，但不确定窗口较大



# 分布式事务

## ➤ 提交协议

2PC存在的问题:

需要多次网络通信, 增加事务的延迟

改进的方向:

1. 由于数据分散到多台机器, 所以需要网络通信进行协调

优化数据分布, 使经常访问的数据集中在一台机器

2. 减少通信次数

以epoch为单位进行提交

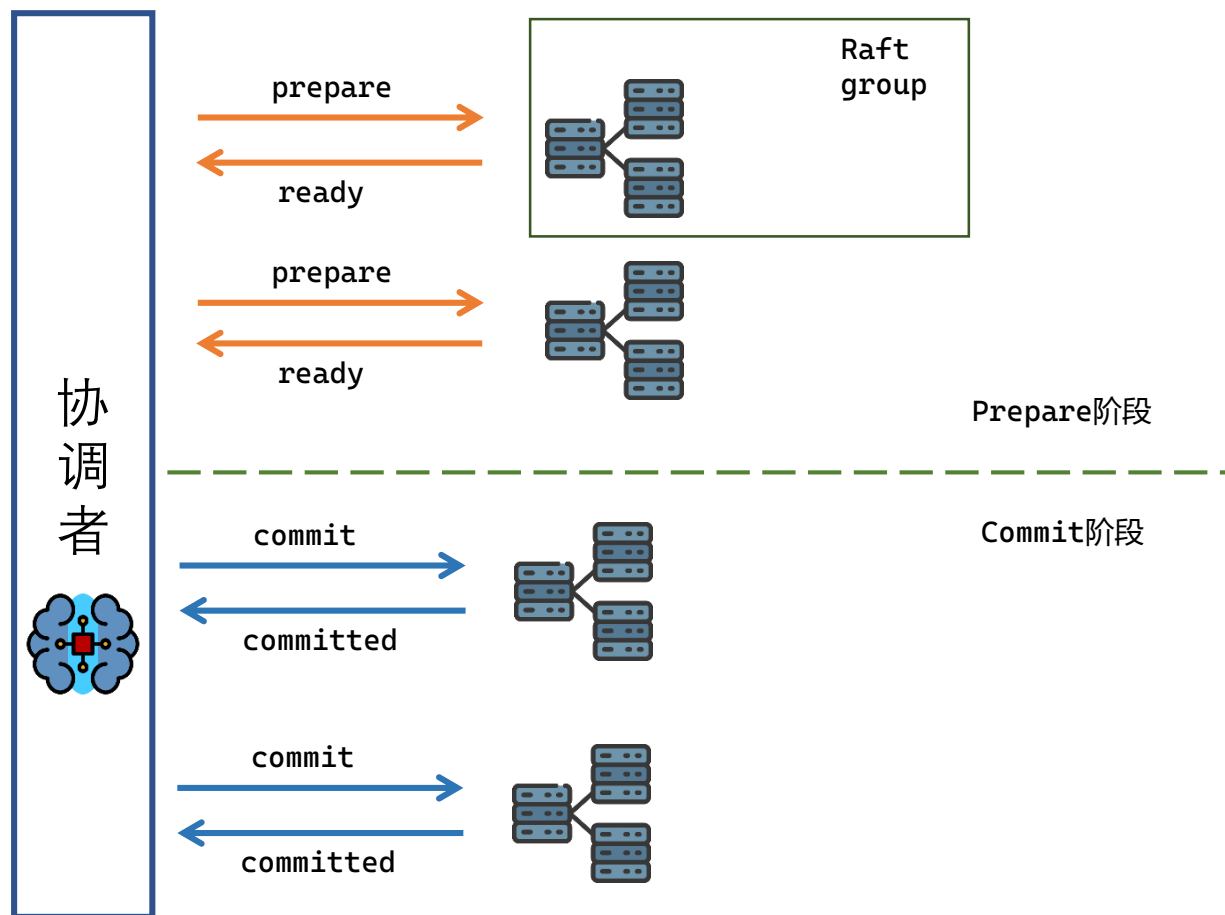
确定性事务

3. 优化提交协议

异步提交技术

4. 减少网络通信开销

RDMA



# 分布式事务

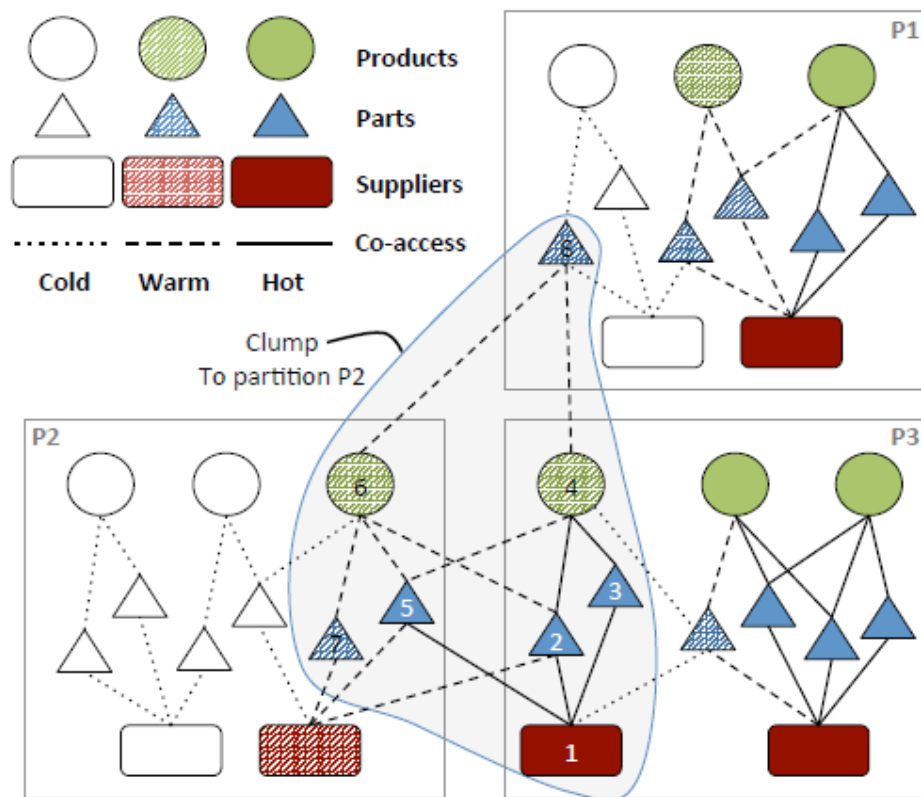
## ➤ 2PC优化：优化数据分布

基本思想：将经常一起访问的数据放到同一台机器上，同时兼顾负载均衡

Clay: FineGrained Adaptive Partitioning for General Database Schemas

- 将数据库建模为图：
  - 所有tuples 水平划分到各个server
  - 同一类数据，划分到同一server，建模为一个顶点
  - 用顶点对应数据的访问频率，作为顶点权重
  - 一个事务同时访问两个点的数据，则添加一条边
  - 用共同访问的次数，作为边的权重。

通过边的权重和移动数据的代价，决定是否要进行一次动态划分



# 分布式事务

## ➤ 2PC优化：优化协议

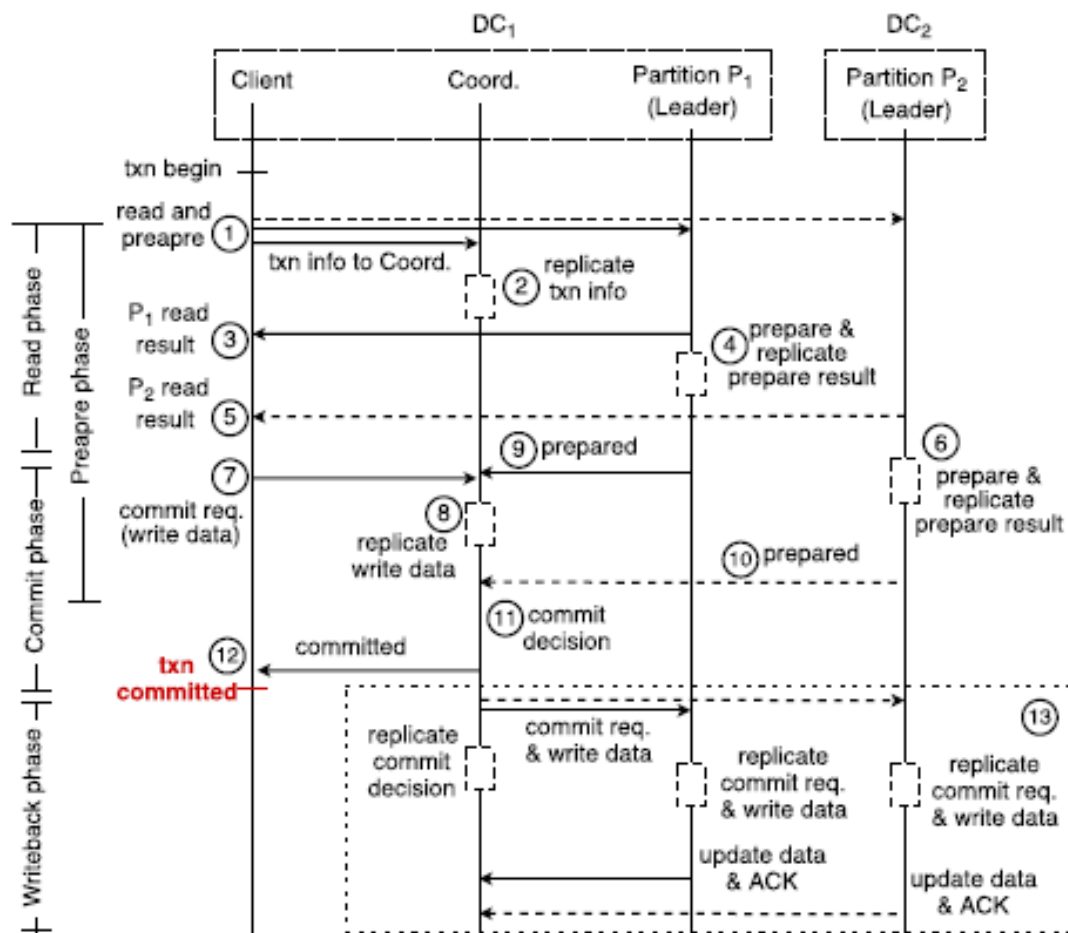
Carousel: Low-Latency Transaction Processing for Globally-Distributed Data

当前存在的问题：2PC+共识协议 显著增加了通信次数

提出的方法：

1. 将事务的执行和2PC **并发执行**，最多两次往返时间便可向client返回结果（在没有conflict的情况下）
2. 2PC协议与共识协议融合，将prepare请求发送给所有节点，而非只发给leader
3. 将修改的数据先保存到client所在的datacenter，之后再异步的复制到原位置

预先条件：事务的read set 与 write set已知



# 分布式事务

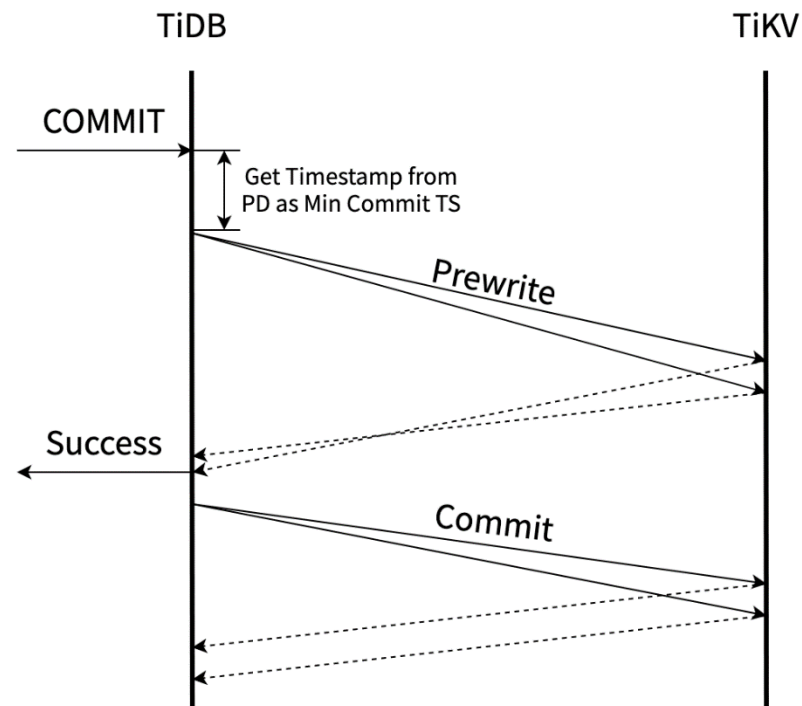
## ➤ 2PC优化：优化协议

TiDB & CockroachDB 异步提交技术

**基本思想：** 由于prewrite阶段完成之后事务的状态便可知，因此可在prewrite阶段后，便可向用户返回事务执行结果

**问题及解决方法：**

1. 如何在prewrite阶段完成时，确定事务的提交时戳
  1. 在prewrite阶段开始前，获取事务的提交时戳
  2. 记录数据项的 被读取的时间戳
  3. 综合原始的提交时戳和读时间戳，取最大值作为提交时戳



# 分布式事务

## ➤ 避免2PC：确定性事务

Calvin: Fast Distributed Transactions for Partitioned Database Systems

基本思想：

1. 在事务执行之前，通过排序层对事务进行排序
2. 所有节点按照排好的顺序执行，节点之间不需要协调，由于操作执行的顺序一致，所以最终节点状态一致

预先条件：事务的read set 与 write set已知

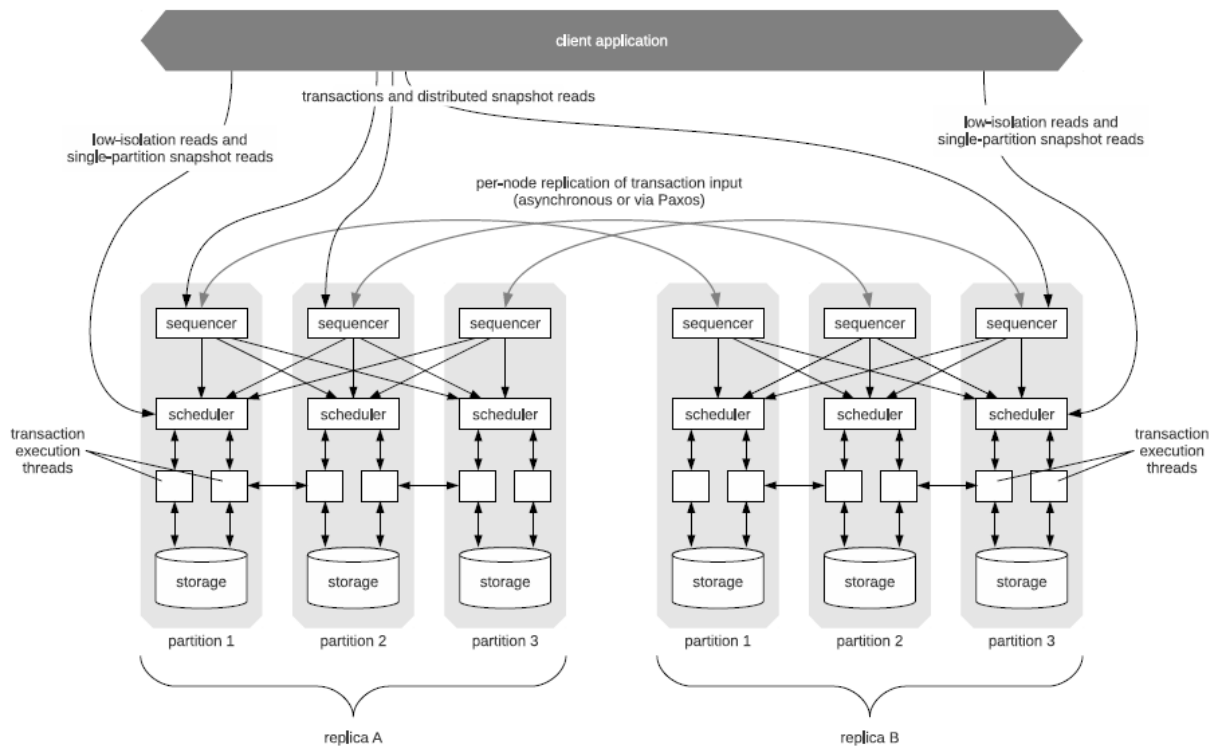


Figure 1: System Architecture of Calvin

# 分布式事务

## ➤ 避免2PC：类确定性事务

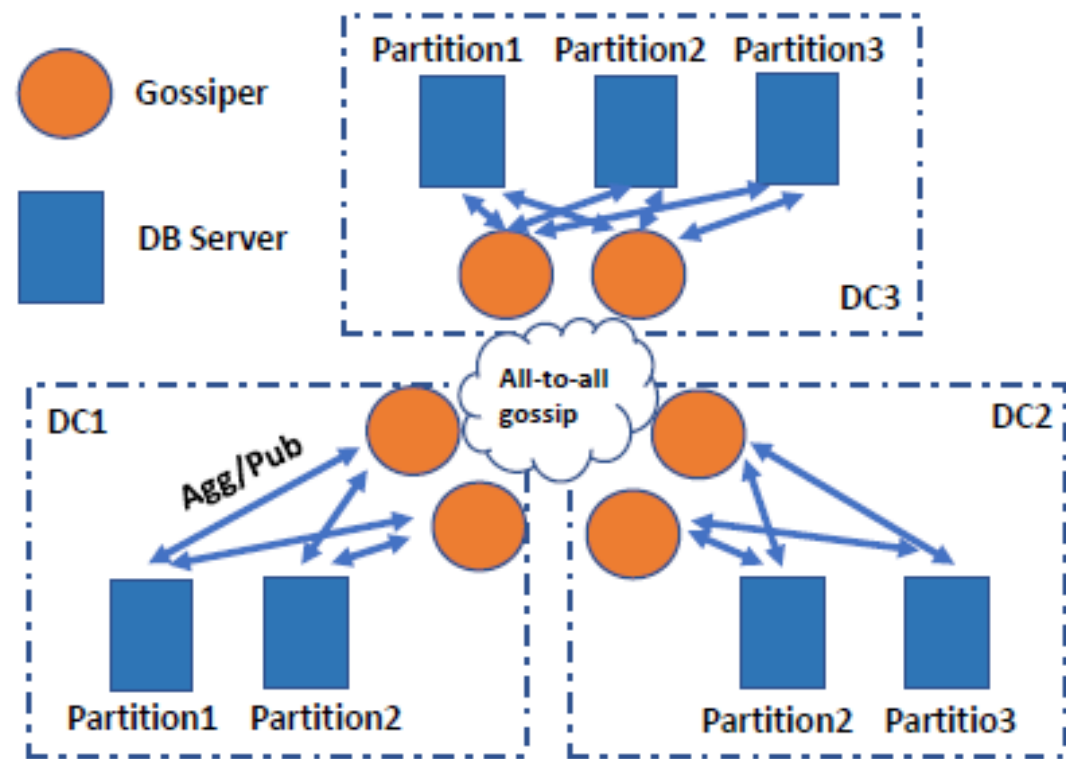
Ocean vista: gossip-based visibility control for speedy geo-distributed transactions

基本思想：

1. 以占位符的方式，在事务执行之前，就写入新的版本
2. 读数据的时候如果遇到了占位符则等待
3. 通过全局的时戳进行批量的提交

由于事务执行之前，已通过占位符的方式确定事务的依赖关系，所以类似于确定性事务，不需要节点之间的协调

预先条件：事务的read set 与 write set已知



# 分布式事务

---

## ➤ 优化2PC：降低网络通信开销

基本思想：使用RDMA

Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks SIGMOD'20

由于网络开销已经不是瓶颈，因此数据partition不是重点考虑的优化因素，首要目标减少事务之间的竞争

# 分布式事务

## ➤ 从共识算法入手

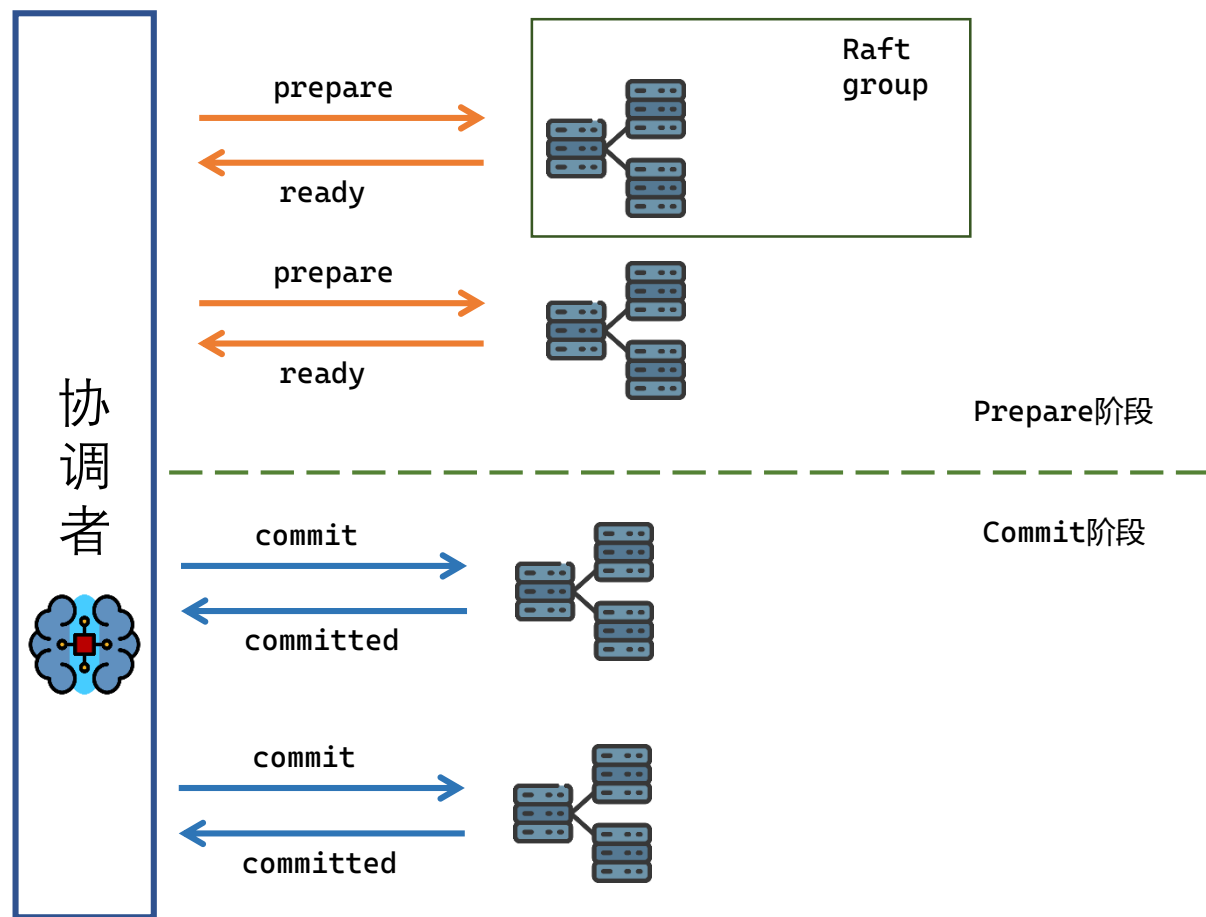
背景：

1. 当前主流分布式数据库均采用raft保证副本之间的一致性
2. Raft协议保证了所有的操作被线性的复制、commit、apply

问题：

1. 由于日志被线性的commit及apply，如果前边的日志由于网络等原因迟迟没有提交，那么其后的日志全部受影响
2. 数据库的一致性由并发控制算法保证，因此raft所保证的线性性对数据库来说是没有必要的。

1 x=3	1 y=1	1 z=9	2 x=2	3 C_y	3 y=2	3 z=1
----------	----------	----------	----------	----------	----------	----------



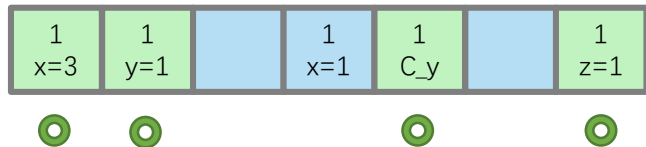
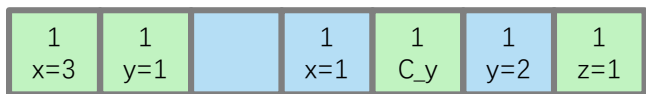
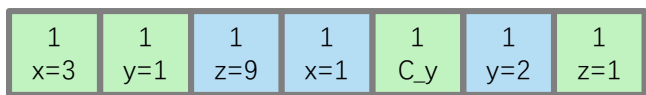


# 分布式事务

## ➤ 从共识算法入手

改进思路：

### 1. 允许raft 乱序提交顺序apply——ParallelRaft算法



### 1. 只有commit的日志才可以apply:

1. Key不冲突, 可以乱序apply
2. Key冲突, 依照log index顺序执行

### 2. look behind buffer:

1. 每个log项都附带
2. 存放前N个log项修改的key

代价：日志空洞问题，leader选举时，需要merge操作消除空洞

# 总结

---

## 单机事务：

### 1. 原子性与持久性：

NVM的出现，buffer和log设计均有较大改变

### 2. 隔离性：

1. 对锁的释放时机进行改进
2. 根据不用的负载使用不同的算法
3. 过期数据版本的回收策略

## 分布式事务：

### 1. 时间戳：

去中心化

### 2. 提交协议：

1. 通过优化布局的方式减少分布式事务的数量——额外的计算开销，数据迁移开销
2. 以批为单位，进行提交——提高了整体吞吐量，同时也增加了单个事务的延迟
3. RDMA

### 3. 确定性事务

需要提前知道事务的读写集合，绝大多数场景不适用

### 4. 共识协议的优化：(类似于加快单机事务中的写盘速度)

1. 自身的复杂性
2. 协议的正确性和安全性难以证明