Part II. Distributed Data

2019年7月1日　23:27

1. 扩展来支持更高负载Scaling to Higher Load

a. 无共享架构Shared-Nothing Architectures

每个运行数据库软件的虚拟主机都称为节点，且节点之间不共享任何资源，只通过软件层面的网络进行通信

b. 复制对比分区Replication Versus Partitioning

分区亦称分片（sharding）

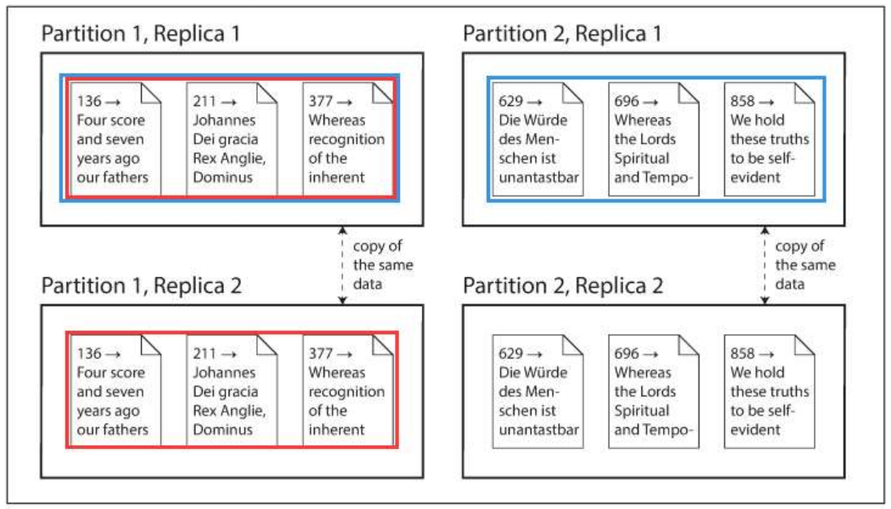


Figure II-1. A database split into two partitions, with two replicas per partition.

S05 Replication

2019年7月1日 23:27

1. 领导者与追随者Leaders and Followers

a. 基于领导者的复制leader-based replication：

i. 某一个复制点作为领导者，所有用户的读写请求都通过领导者执行

ii. 领导者处理请求后将数据转发给所有追随者，追随者也同样应用这些改变

iii. 追随者可以应对只读请求

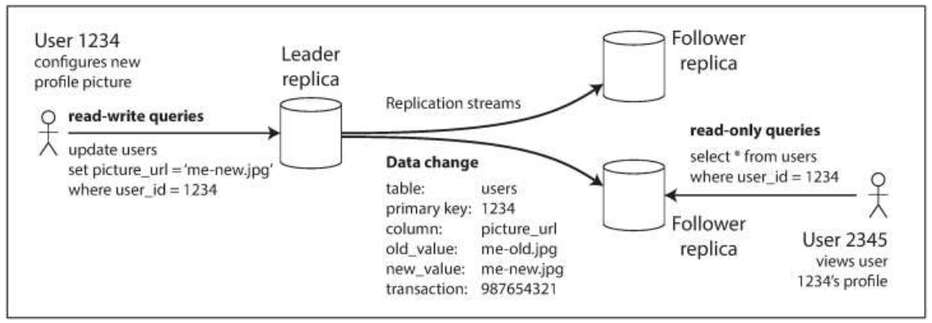


Figure 5-1. Leader-based (master–slave) replication.

b. 同步复制对比异步复制Synchronous Versus Asynchronous Replication

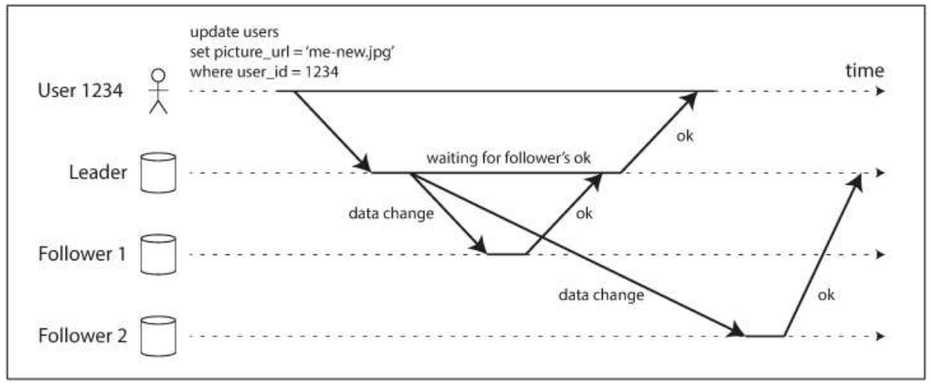


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

同步的优点在于确保追随者拥有的数据一定与领导者一样是最新的，而延迟会高；异步的优点在于响应速度快，而可能丢失数据

实践来说，所有追随者都同步就会导致很有可能任一追随者故障都会将整个系统阻塞，因此往往少数几个追随者同步复制，而大多数追随者异步复制

c. 设置新的追随者Setting Up New Followers

i. 领导者快照snapshot

ii. 新追随者启用快照，并连接上领导者

iii. 请求快照之后所有新的修改，待和领导者同步，称为赶上caught up

d. 处理节点宕机

i. 追随者失效：追赶恢复

ii. 领导者失效：故障转移

1) 确认领导者已经失效
2) 选择新的领导者，通常拥有最新数据的追随者是新领导者的候选
3) 系统确认新的领导者，所有请求重定向到新的领导者

领导者切换存在一些需要仔细处理的细节

1) 如果采用了异步复制，则如何处理丢失的数据
2) 宕机的旧领导者重启之后重新加入集群，如何处理新旧领导者的冲突
3) 脑裂问题，当两个节点都认为自己是领导者，如何处理冲突
4) 如何确认故障的超时时间，即领导者失去响应多久才可以算是宕机

e. 实现复制日志Implementation of Replication Logs

i. 基于语句的复制Statement-based replication

每一条执行语句（在关系数据库中即每一条INSERT/UPDATE/DELETE语句）都被推送给追随者执行，从而达成复制，存在的问题诸如语句中包含NOW()等函数，导致追随者执行时与领导者不一致，或是语句存在副作用，虽然简单但没有广泛使用

ii. 传输预写式日志的复制WAL shipping

对于SSTables和B树，都存在顺序添加的log，直接传输这个log来达成复制，缺点在于这些log是直接应用到SSTable/B-trees，存在和底层存储引擎的耦合

iii. 逻辑日志复制Logical (row-based) log replication

使用逻辑日志可以使得与底层存储引擎（SSTables/B-trees）解耦，通常是一系列行级别描述数据修改的日志

iv. 基于触发器的复制Trigger-based replication

前述都是基于数据库的复制，当需要更灵活时，复制的逻辑就可以提升到应用层来完成，但是不当设置容易导致问题

2. 复制延迟问题Problems with Replication Lag

对于从追随者上读，但是在领导者上写的场景，复制延迟就会带来问题

a. 读己之写Reading Your Own Writes

写发生在领导者上，而读发生在追随者上，由于复制延迟，有可能出现无法看到已经成功提交的请求，己无法读己之写，即不保证读写一致性（read-after-write consistency），注意读写一致性只保证读己之写，不保证读他之写，部分方法如下：

i. 读可能修改的数据时，从领导者读，其他情况从追随者读，这需要一种策略来判断读是否读了可能修改的数据
ii. 用户可以记录最近写的时间戳，读时带上时间戳，由系统保证读的结果一定不比时间戳旧，否则的话由领导者或其他节点来服务，时间戳可以是逻辑时间用来标注读的顺序

b. 单调读Monotonic Reads

即不会时光倒流，已经读到新的数据，再次读确保一样或更新，不会读到旧的数据，这可能会发生在不同节点上读，先读的节点数据更新，而后读的节点延迟大，出现了非单调读，可以通过始终读同一个副本来确保单调性

c. 一致前缀读Consistent Prefix Reads

即多个不同的写按某个顺序发生时，任何读也要读到这个顺序，例如先写A再依赖A写B，但是读时从两个不同延迟的节点上分别读A和B，结果先读到B再读到A，但是B依赖A，此时出现因果不一致

3. 多主复制Multi-Leader Replication

a. 多主复制的应用场景Use Cases for Multi-Leader Replication

i. 多个数据中心Multi-datacenter operation

1) 性能：若是单主节点，则在多个数据中心中，则只有一个数据中心能够执行写任务
2) 数据中心宕机：若是单主节点，则主节点所在中心宕机时，需要选举出新的主节点，而多主节点时系统正常运行
3) 网络故障：单主节点对网络问题更加敏感，多主节点则更稳定
4) 冲突：多主节点必须解决写冲突

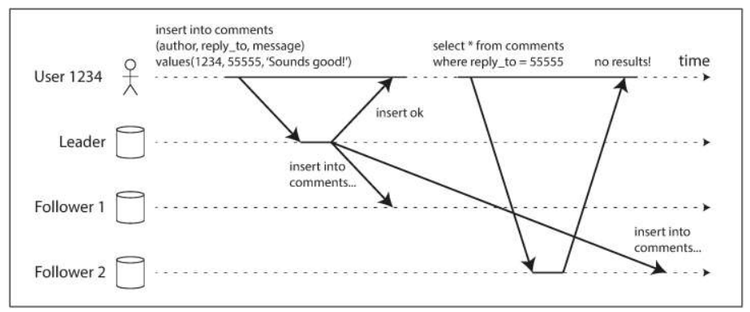


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

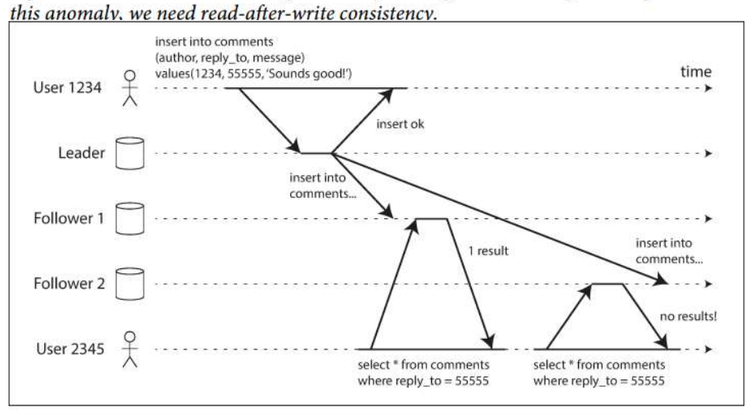
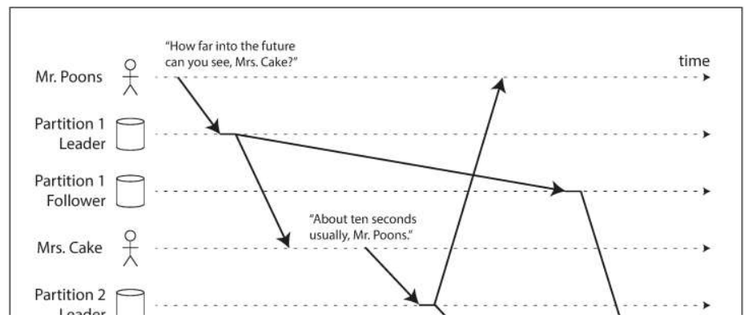


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.



3) 网络故障：单主节点对网络问题更加敏感，多主节点则更稳定
4) 冲突：多主节点必须解决写冲突

ii. 存在离线操作Clients with offline operation

存在离线操作的云端备份本质上也属于多主复制，当离线时本地就是主节点，修改被直接接收，重新联网后解决和云端主节点的冲突，并备份到云端

iii. 合作编辑Collaborative editing

实时多人合作编辑也属于多主复制，每个人的修改都需要被接收，多人的修改之间解决冲突

b. 处理写冲突Handling Write Conflicts

i. 同步对比异步冲突检测

同步冲突检测要求写入成功时再通知所有冲突侧，则不如使用单主节点，而异步冲突检测要求冲突的写入立即被接收，直到检测到冲突时再提醒用户，可能会有延迟

ii. 冲突避免

最佳方式就是避免冲突，即涉及到相同数据的修改通过同一个主节点，此时天然不会有冲突

iii. 收敛至一致的状态

冲突解决要求所有冲突的数据最终在所有副本上都达成一致

1) 每一个写赋予一个UID（逻辑时间等），冲突时拥有最大UID的为最终值（LWW, last write wins），可能有数据丢失
2) 每个副本赋予一个UID，冲突时来自更大UID的副本的修改可以覆盖来自较小UID的副本的修改，可能有数据丢失
3) 不丢弃任何修改，合并冲突值
4) 记录冲突的数据和相关的元信息，由应用层来负责解决冲突

iv. 自定义冲突解决逻辑

大多数多主复制都通过调用应用层逻辑来解决冲突，应用层逻辑会在如下时刻调用

1) 写时：当数据库修改数据时检测到冲突，即调用冲突解决逻辑
2) 读时：当数据库修改数据时检测到冲突，记录所有冲突的情况，等下次读到对应的数据时，多个版本的冲突数据都传给应用层冲突解决逻辑，由应用层解决并保留某一个版本并写回数据库

c. 多主复制拓扑关系Multi-Leader Replication Topologies

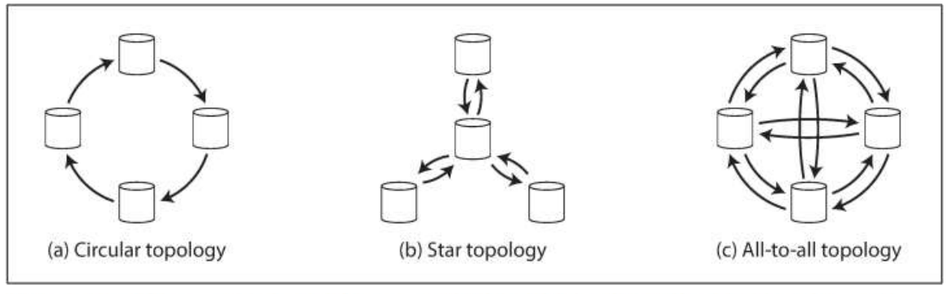


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

对于非全连接的结构可能存在单点故障，但是对于全连接来说也可能会出现部分连接速度不一致，导致部分节点的修改出现不一致，例如右图出现了非一致前缀读，使用版本向量version vectors可以解决

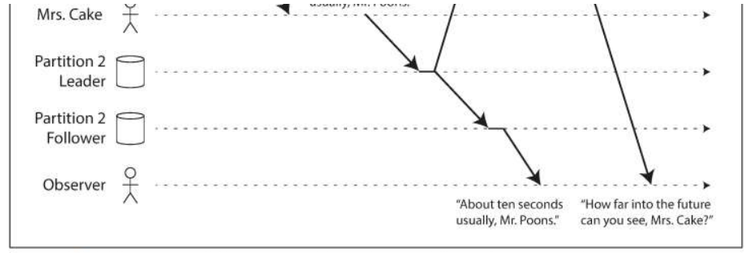


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

4. 无主复制Leaderless Replication

a. 当副本节点宕机时写入数据库

写入时同时在所有副本上写入，无视失败的节点，半数以上成功（quorum）即视为成功，读取时在所有副本上读取，半数以上返回即视为成功，使用版本号（version number）来决定最新值，并将最新值再写入到返回过时值得副本中

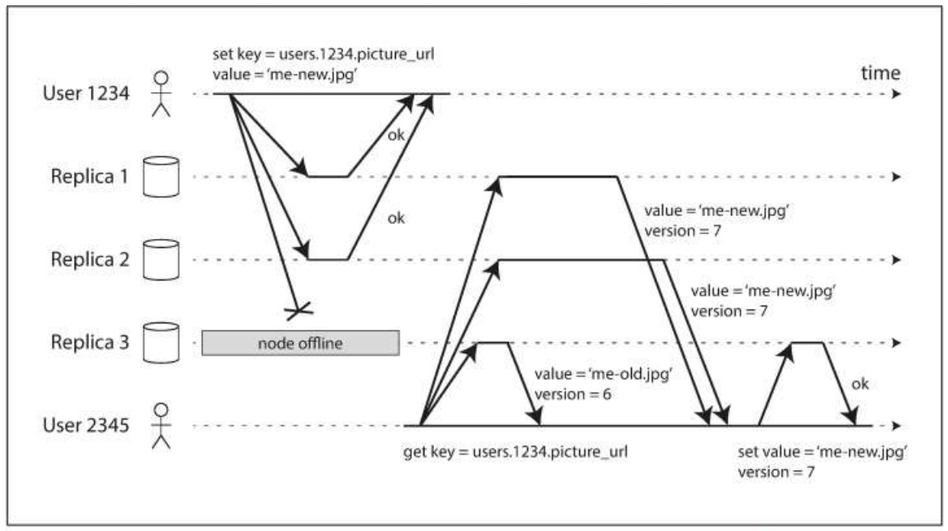


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

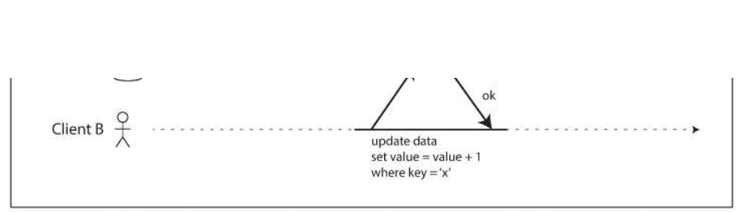


Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

i. 读修复和反熵Read repair and anti-entropy

1) 读修复：当并行从多个副本读时，会自动进行读修复，过时的数据会被重写为最新的数据，但是一直未被读到的数据就可能难以保持多副本和持久性
2) 反熵过程：后台运行检查进程，不断扫描副本的不一致情况，并通过其他副本对不一致副本进行修复

ii. 读写的法定人数Quorums for reading and writing

n-总副本数，r-读成功副本数，w-写成功副本数

法定人数即为w+r>n，通常可以选择n=2k+1，w=k+1，r=k+1由此来确保，通过根据系统负载的情况，可以灵活选择w/r来是的系统的综合性能最高，例如读多写少，就可以较大的w配合较小的r

b. 法定人数一致性的限制Limitation of Quorum Consistency

本质上法定人数要求w+r>n是为了确保读写成功的节点总有至少一个重合，就确保了一致性，因此只要能够确保读写成功节点有重合，不必要w+r>n也可以确保一致性

i. 松散的法定人数会导致读写的节点没有重合，从而读到过时数据
ii. 并发写入依然需要解决冲突问题
iii. 并发读写，写可能只成功了部分节点，读未必能够成功读到写入成功的节点，即使随后法定人数也满足，也出现了读到过时数据
iv. 如果写入在部分节点上失败了，最终不足法定人数，整体失败，但是由于成功的节点不会回滚，此后的读也是不确定的
v. 如果携带最新数据的节点宕机数据丢失，当从其他副本恢复时可能被旧数据覆盖，从而破坏了法定人数前提
vi. 并且前述的读写一致性、单调读、前缀一致性读都不能保证，更强的一致性需要事务transaction或是共识consensus来解决

c. 松散法定人数与带提示的接力Sloppy Quorums and Hinted Handoff

d. 检测并发写Detecting Concurrent Writes

i. 最后写者获胜，丢弃并发写入Last write wins, discarding concurrent writes

给每个写入附带一个时间戳，大者获胜；但是LWW会导致并发写入成功的但非最后写入的数据被丢弃，甚至会导致非并发写入的数据也被丢弃，当数据不允许丢弃时，LWW算法并不适用；通常如果数据只会写入一次，随后视为immutable，那么LWW算法就是合适的

ii. happens-before关系和并发

若A依赖B发生，则B happens-before A，若B依赖A发生，则A happens-before B，若相互无依赖，则AB并发（时间上是否同时并不重要，只要AB互相不依赖就认为AB是并发的）

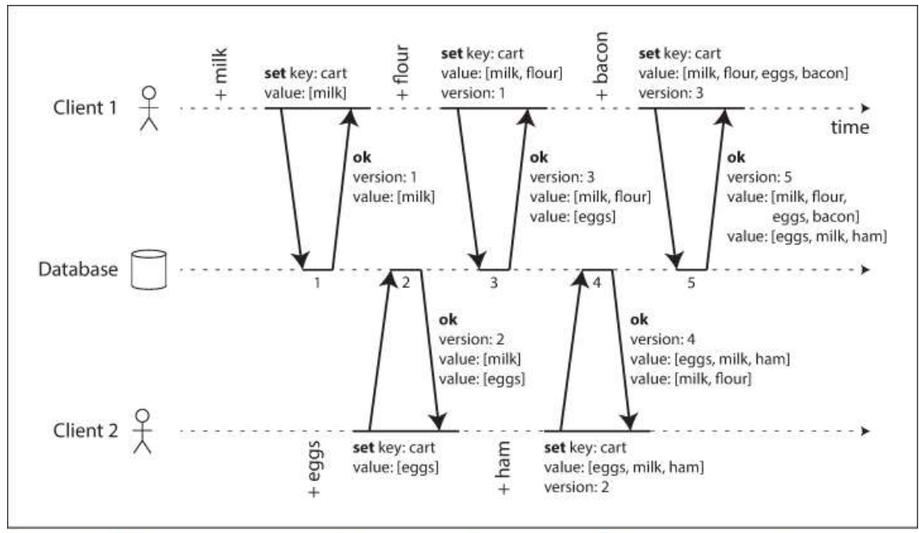
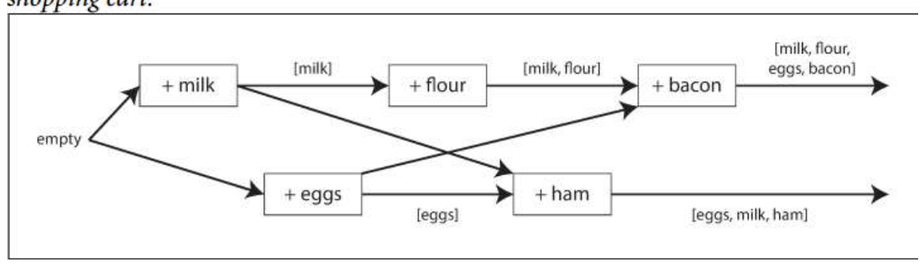


Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.



iii. 捕获happens-before关系Capturing the happens-before relationship

通过版本号来控制并发：

1) 每个key都维护一个单调递增的版本号，存储数据和相应的版本号
2) 读时，返回所有尚未被覆盖的数据和最新的版本号（写之前必须先读）
3) 写时携带之前获得的版本号，并合并之前获得的所有数据，随后在此基础上写并获得新的版本号和所有数据，覆盖之前读的版本号对应的数据
4) 当服务端收到带有版本号的写时，覆盖所有版本号<=此版本号的数据，但是要保存所有>此版本号的数据
5) 当不携带版本号进行写时则与所有其他写并发，不会覆盖任何内容（类似于上图出现第三条分支）

iv. 合并并发写入Merging concurrently written values

版本号算法确保没有数据会被无声丢失，但是客户端需要做额外的工作，执行数据的合并，当删除数据时需要使用墓碑标记

v. 版本向量Version vectors

对于一个副本上的数据用一个版本号，则在多个备份的数据下就是用一系列版本号，即版本向量，每个副本一个版本号

S06 Partitioning

2019年7月3日　15:01

1. 分区与副本Partitioning and Replication

分区与副本往往结合使用，副本用来支持备份和高可用，而分区用来负载均衡、性能以及容纳更多数据，所有对数据库的备份策略对分区的备份依然有效

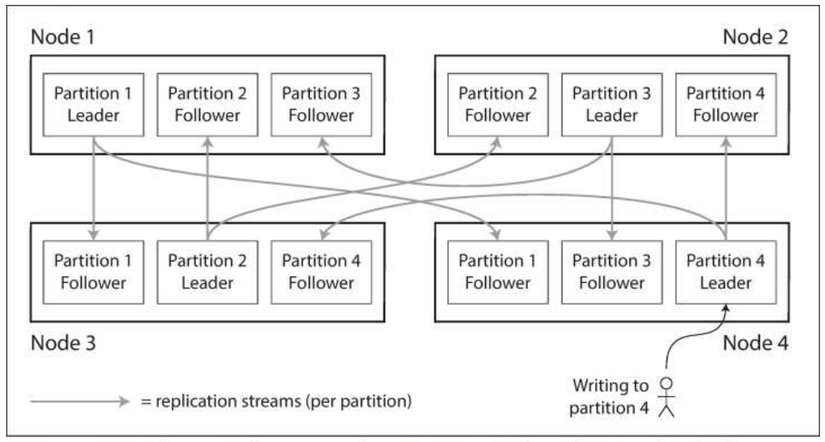


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

2. 键值数据的分区Partitioning of Key-Value Data

分区需要考虑均匀分布数据，不均匀程度称为倾斜skewed，尽可能将负载也均匀分配，若某个分区有异常高的负载就称为热点hot spot

a. 基于key的范围分区Partitioning by Key Range

   基于范围需要仔细选择边界来使得所有数据均匀分布，在每个分区内依然可以使所有的key有序（使用SSTables/LSM-trees）使得范围扫描时有优势，但是范围分区可能导致热点

   例如根据日期分区写入传感器数据，则今天所有数据都写入同一个分区，产生了热点，可以通过每个数据的日期再前缀传感器下标，由此使得数据既根据日期分了区，又使不同传感器分布在不同分区上，避免了热点（类似二重分区）

b. 基于key的散列分区Partitioning by Hash of Key

   使用key的散列进行分区，就可以通过设计良好的散列函数，使得数据均匀且避免热点（但要注意在任何节点任何时间下散列函数对同一个key应产生一致的散列值），散列分区高效支持key的点查询，但是散列分区丢失了数据的局部性，无法支持高效的范围查询（所有范围查询需要所有节点同时运行并且合并结果）；可以通过复合key（compound primary key），例如多个列组成一个key，根据第一个列进行散列分区，根据余下的部分进行排序，来间接支持范围查询

   例如根据id和timestamp复合，由id来确定分区，由timestamp来排序，则可以高效获取某个id在一段时间内的所有事件

c. 负载倾斜与消除热点Skewed Workloads and Relieving Hot Spots

   极端情况下，即使是散列分区，都会由于例如重度访问同一个key而导致负载倾斜与热点，大部分现有架构都不会自动处理这种负载倾斜，可以通过给预先知道会出现重度负载的key添加不同的随机数，进一步分区，查询时就在多个分区上把同样key不同随机数的结果合并，从而进一步分散负载消除热点

3. 分区与二级索引Partitioning and Secondary Indexes

a. 基于文档的二级索引Partitioning Secondary Indexes by Document

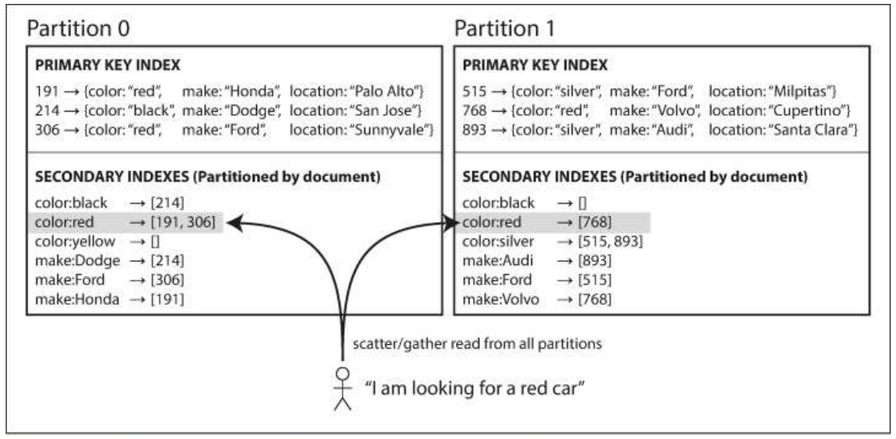


Figure 6-4. Partitioning secondary indexes by document.

基于文档的二级索引又称为本地索引（local index，相对应的是全局索引|global index），因为每个分区对应的索引只关心本地数据，数据修改时也只需要修改本地索引，但是依赖二级索引的读取就需要向所有分区的索引发送查询scatter，并合并结果gather

b. 基于关键词的二级索引|Partitioning Secondary Indexes by Term

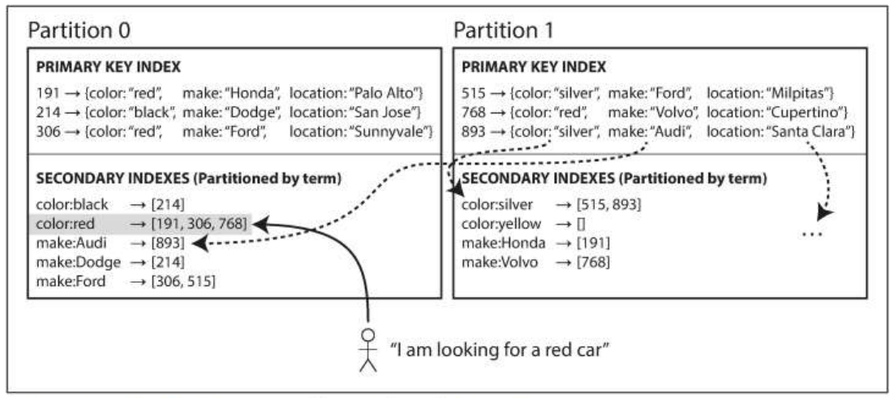


Figure 6-5. Partitioning secondary indexes by term.

基于关键词的二级索引又称为全局索引global index，全局索引覆盖了所有数据，因此本身也要被分区，而分区方式可以有别于key的分区（可以根据关键词本身进行分区），优点在于比本地索引更加高效，不存在scatter/gather过程，缺点在于某个分区的数据更新，其二级索引可能位于其他分区，二级索引的更新存在延迟并且实现更加复制

4. 分区再平衡Rebalancing Partitions

随着数据增多或是负载增大，需要新增节点并对已有数据进行分区重新平衡，这要求重平衡能够依然保持数据和负载均匀分布，过程中依然持续提供读写服务，并且尽可能少的移动现有数据来提高性能

a. 再平衡策略Strategies for Rebalancing

   i. 反面教材：对节点数取模How not to do it: hash mod N

      直接对当前节点数取模，则当加入新节点时新的取模运算会导致海量数据需要重新迁移，这是不可接受的

   ii. 固定分区数Fixed number of partitions（每个分区大小与总数据量成正比）

      创造远多于节点数的分区数，每个节点管理一系列分区（也可称为chunk），当新增节点时从原节点处获取一部分分区，实现再平衡，并且再获取过程中原来的分区依然可以对外提供服务

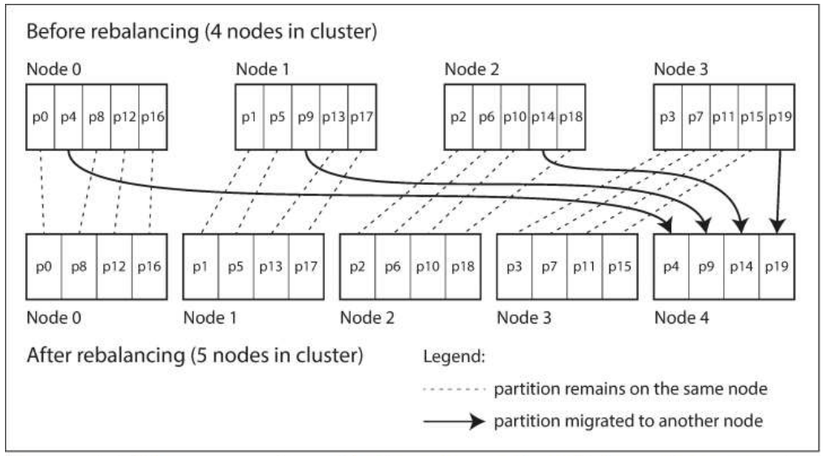


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

进一步，这种策略还允许异构节点的存在，性能更高的节点存储更多分区，但是缺点在于对于未知总量的数据，固定分区数可能导致一开始数据分区过多（起始数据少）而后期每个分区过大（数据不断增多但总分区数固定）

iii. 动态分区Dynamic partitioning（分区数量与总数据量成正比）

使用固定分区数的范围分区是很不方便的，一旦初始分区边界设定不良，可能会随着数据进入，部分分区包含数据过多，而另一部分分区始终很空

动态分区允许当某个分区数据量超过一定限制时就自动分裂成两个分区，少于一定限制时就自动和相邻分区合并，类似B树的分裂和合并过程；同时也约定每个节点管理多个分区，当分区分裂时根据当前节点负载决定是否要移动到其他节点上

iv. 按节点比例分区Partitioning proportionally to nodes（分区数量与节点数量成正比）

等同于固定每个节点的分区数，当有新节点加入时，现存其他节点的部分分区分裂成两份，其中一份转移到新节点上

对于随机挑选分区边界要求使用基于散列的分区，例如原来[0,10,20,40]，新增一个节点[0,10,20,30,40]，将[20,40]分割成两个分区，其他区域数据不移动，这种做法更接近一致性散列consistent hashing

b. 运维：自动还是手动再平衡Operations: Automatic or Manual Rebalancing

最好是手动再平衡，自动可能会出现一些意想不到的情况

5. 请求路由Request Routing

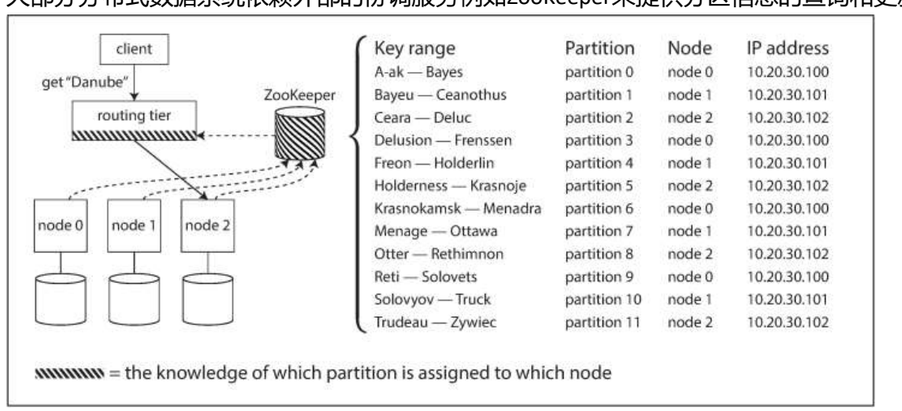
分区可能变化，而查询请求发起方并不一定实时知晓最新的位置，因此需要请求路由，也类似于服务发现service discovery，常见做法：

a. 允许请求发送给任意节点，若该节点不拥有相关数据，则推送请求给拥有数据的节点，若有数据则直接服务

b. 所有请求发送给负载均衡器，由负载均衡器根据分区与节点的信息推送请求给相应节点

c. 要求客户端对所有分区信息感知，主动发送给拥有数据的节点

大部分分布式数据系统依赖外部的协调服务例如ZooKeeper来提供分区信息的查询和更新（即服务发现）



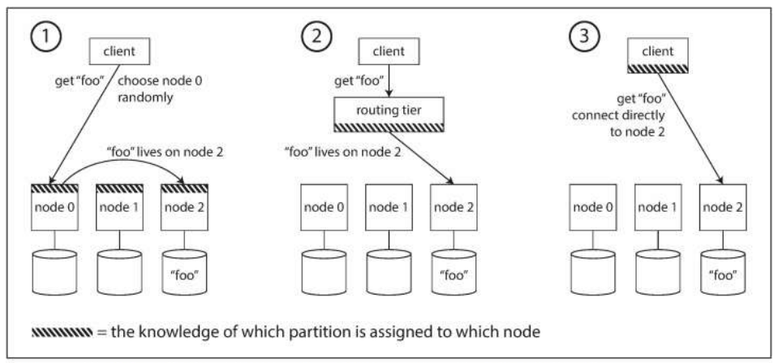
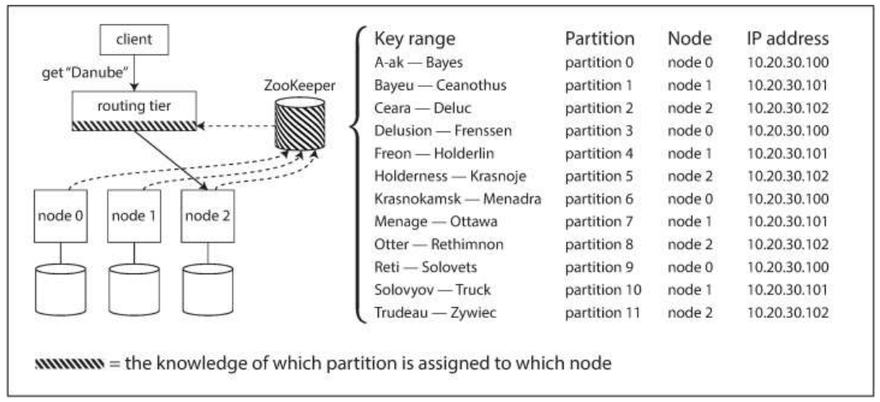


Figure 6-7. Three different ways of routing a request to the right node.



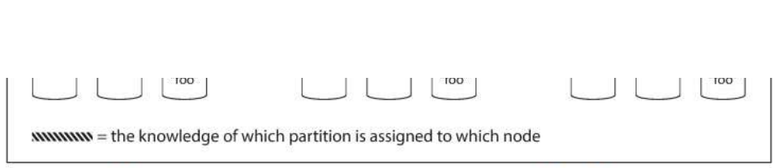


Figure 6-7. Three different ways of routing a request to the right node.

目前的讨论都针对最简单的KV数据库，也就是大多是分布式NoSQL支持的级别，而例如大规模并发处理（MPP）架构的分析型关系数据库则提供更加复杂的SQL查询支持，例如join/filtering/grouping/aggregation，MPP查询优化器将查询语句分解成针对每个分区的具体查询语句，最后再整合最终结果

S07 Transactions

2019年7月3日 22:05

1. 事务的简介The Slippery Concept of a Transaction

a. ACID的含义The Meaning of ACID

事实上ACID的意义是非常模糊的，很多宣称支持ACID的数据库实际上提供的保证各不相同

i. 原子性Atomicity

要么都做，要么都不做，即事务中途失败需要把已经做的修改回滚撤销掉

ii. 一致性Consistency

要求数据库保持一致性，即例如一个事务开始时数据库某些约束是满足的，事务结束时这些约束依然满足，但事实上除了外键约束等数据库自身的约束，例如A+B=C这种约束是由应用程序来保证的，数据库只负责存取数据，因此一致性并不应该是数据库的属性

iii. 隔离性Isolation

多个并发事务之间相互隔离，类似于并发场景下的竞争问题，一个事务做了各种修改之后，另外的事务要么看到提交后的所有修改，要么什么也看不到

iv. 持久性Durability

数据能够持久存储而不会丢失，一般来说就是事务一旦提交，数据就落盘存储不会丢失，事实上极端场景下数据无法满足持久性，随着分布式系统的兴起，持久化除了可以落盘实现，也可以通过副本实现，只要始终有一个完整的副本，也可以视为持久性

b. 单对象和多对象操作Single-Object and Multi-Object Operations

i. 单对象写Single-object writes

Atomicity可以通过写日志来实现，Isolation可以通过对象加锁实现，但是事务往往操作多个对象，执行多个操作

ii. 多对象事务的需求The need for multi-object transactions

iii. 错误处理与终止Handling errors and aborts

事务的核心特点有一旦失败就会回滚，因此应用程序可以安全重试（无主复制leaderless replication系统只提供尽我所能保证，应用程序需要自己处理半完成的情况），但是重试一个终止的事务虽然简单高效，也存在缺陷：

1) 如果事务实际上成功了，二是在告知应用层时出现了网络等原因，导致了应用重试了一个成功的事务，则需要内部去重机制，也即幂等性idempotence的保证
2) 如果事务由于负载过大失败，则重试会加重服务器过载，导致雪崩，可以考虑指数回退exponential backoff算法重试
3) 只有暂时性错误重试才有意义，例如暂时网络中断、正在故障转移等，对于永久性错误重试无意义
4) 如果事务对于数据库以外的部分都有副作用，而数据库回滚只是数据库内的，则真正的事务原子性并没有得到保证，对于跨系统的事务可以考虑二阶段提交two-phase commit协议来解决
5) 如果客户端本身在重试时崩溃，则数据也全都丢失了

2. 弱隔离级别Weak Isolation Levels

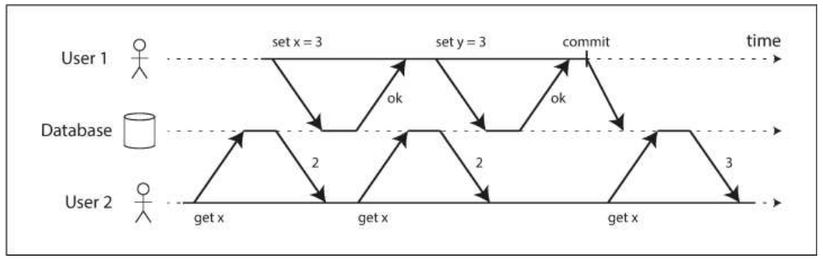
可序列化的隔离级别（一系列并发的事务其处理结果可以序列化成顺序执行的事务的处理结果）代价过高，大部分系统一般都是提供比可序列弱的隔离级别，例如：

a. 读已提交Read Committed

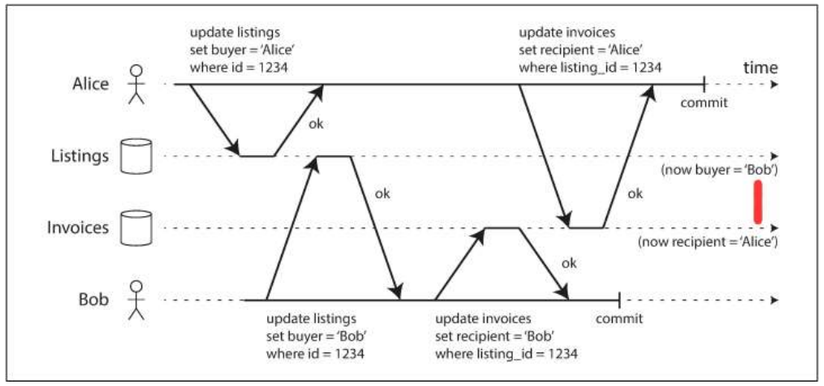
最基本的隔离级别就是读已提交，保证读数据时只有提交的事务数据才会被看到，没有脏读dirty reads；保证写数据时只有提交的事务数据才有可能被覆盖，不会覆盖还未提交的数据，没有脏写dirty writes

注意，读已提交并不保证不丢失数据，例如并发递增（依赖旧值），则读已提交依然会导致丢失值

i. 没有脏读No dirty reads



ii. 没有脏写No dirty writes



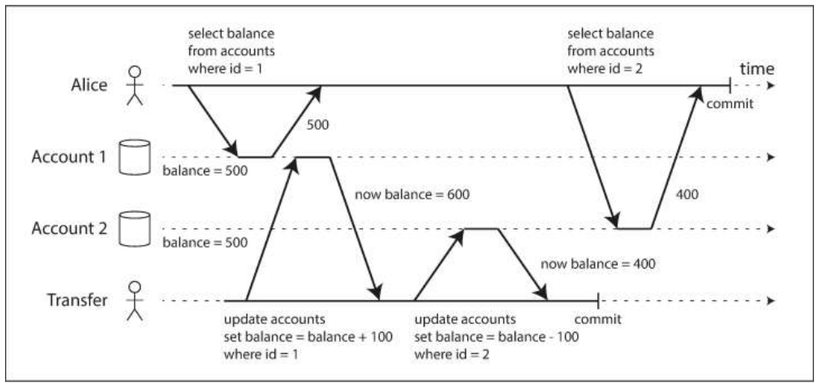
iii. 实现读已提交Implementing read committed

使用行级锁row-level locks来避免脏写，一个已经事务持有行级锁再进行修改，则提交前锁没有被释放，其他事务只能等待；

但是不用锁来避免脏读（性能损失太大），而是通过保存旧值和每个事务自己正在使用的新值，则并发事务都首先获得当前旧值

b. 快照隔离和可重复读Snapshot Isolation and Repeatable Read

读已提交还存在其他问题例如Alice两次读到到的数额加起来发现少了100，出现了数据不一致，如果此时再读一次数据就一致了，这种现象称为不可重复读nonrepeatable read或读取偏差read skew

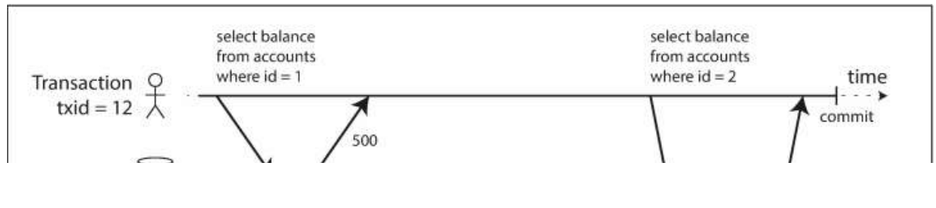


在部分场景下这是可以接收的，但是涉及到例如备份backups（备份需要如此读所有数据，则出现这种情况时偏差值被备份了，并在恢复时会导致数据库数据不一致）或分析型查询（检查数据一致性，出现偏差时就会失败）时就不可接受了

i. 快照隔离要求每个事务始终只会看到事务开始时的数据，即使后续其他事务修改了数据也不可见，即类似给数据库打了快照，在快照上执行事务

ii. 实现快照隔离

1) 使用锁来避免脏写，由此多个写事务需要等待，但是读事务可以并发且不需要锁，因此写并不阻塞读，读也并不阻塞写；数据库需要保存一个数据的多个提交版本，这种方式也叫多版本并发控制MVCC, multiversion concurrency control
2) 在每个事务开始时获得一个递增的TID，写入任何数据时数据都带上对应事务的TID，每行都有created_by=TID，deleted_by=TID，访问时每个事务的可见性规则见iii



2) 在每个事务开始时获得一个递增的TID，写入任何数据时数据都带上对应事务的TID，每行都有created_by=TID，deleted_by=TID，访问时每个事务的可见性规则见iii
3) 定期删除不会再被任何事务访问到（所有进行中事务TID均大于deleted_by=TID参考iii）的且被标记为deleted的数据
4) update被翻译成delete+insert两步走

iii. 快照隔离的可见性规则Visibility rules for ovserving a consistent snapshot

1) 每个事务开始时，数据库列出所有活跃的进行中事务，任意这些事务的修改无论是否commit，都被忽略
2) 任意在此事务之后开始的新事务的任意修改无论是否commit，都被忽略
3) 所有终止的事务的任何修改，都被忽略
4) 其他所有修改都可见

换言之只要满足以下情况就可见

1) 每个事务开始时，created_by=TID事务已经commit
2) 对象没有deleted_by，或者deleted_by=TID事务还未commit

iv. 索引和快照隔离Indexes and snapshot isolation

索引指向所有快照，快照有更新时就更新索引，或者对B-trees进行copy-on-write

v. 可重复度和命名混淆Repeatable read and naming confusion

快照隔离被广泛使用，但是在不同的系统中给其定了不同的名字，Oracle称之为可序列化，PostgreSQL/MySQL称之为可重复读

c. 避免更新丢失Preventing Lost Updates

在涉及read-modify-write时可能会出现更新丢失，当多个事务并发对同一个数据read-modify-write时，就可能会出现更新丢失，事实上先读，基于读做增量修改，再写回非常常见，因此也有一系列方法来解决：

i. 原子写操作Atomic write operations

原子写确保对对象的修改是并发安全的，底层往往通过对修改对象加锁来实现，由此对于多副本的情况，也有分布式锁的概念

ii. 显式加锁Explicit locking

当原子写操作不足以支持安全时，可以通过显式加互斥锁，例如使用for update语句

```
SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; ❶

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

❶ The FOR UPDATE clause indicates that the database should take a lock on all rows returned by this query.

iii. 自动检测更新丢失Automatically detecting lost updates

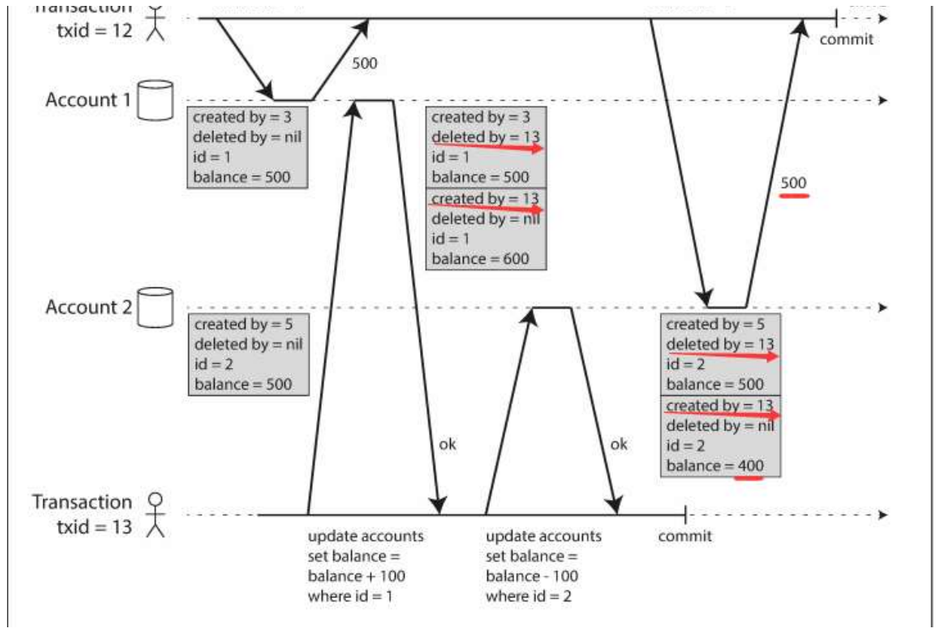
允许操作并发执行，但是由事务管理器来检测是否有丢失的更新，如果有则主动中止一些事务并发起重试，检测丢失的更新可以和快照隔离搭配实现

iv. CAS修改, Compare-and-set

v. 冲突解决和副本Conflict resolution and erplication

由于多副本的数据库，每个副本可能被单独修改，因此检测同一数据在多个副本上的并发安全要求额外的工作，同时由于显式加锁和CAS都假定一个对象，因此对于无主leaderless和多主multileader复制不适用，参考检测并发写入，允许并发修改，但是通过维护不同版本的值（版本号），由应用程序来解决冲突

d. 写入偏差和幻读Write Skew and Phantoms



i. 写入偏差

例如要确保至少有1人，但是2人并发提出请假，在快照隔离的作用下，2人分别请假的事务都提交成功了，结果反而最后1人都没有了；可以通过对currently_on_call的查询显示加互斥锁for update，由此将两个单独的事务序列化，不同时发生，这种并发写入也叫写入偏差

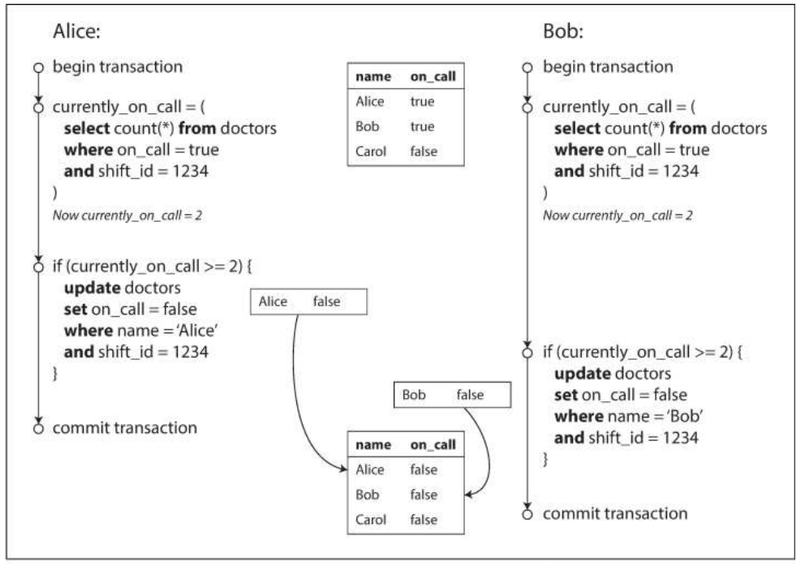


Figure 7-8. Example of write skew causing an application bug.

```
BEGIN TRANSACTION;
SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; ❶
```

ii. 幻读

幻读的定义是事务在执行连续的读取时，由于并发事务的修改导致了连续的读取返回了不一样的值

快照隔离确保了只读查询不会出现幻读，但是依然无法避免自身对数据修改，最后多个并发事务提交的结果合并时出现约束被打破，而读已提交则无法避免幻读

iii. 物化冲突Materializing conflicts

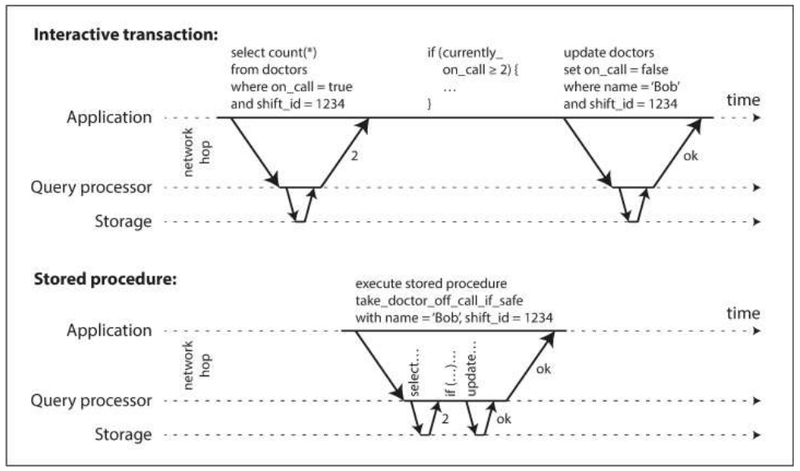
上述的例子要求select count(*)....>1，由于这个限制并不是一个对象，不能显式加锁，最后通过对相关的行加锁for update来解决；由此可以通过把这个count(*)作为一个对象，此时即引入了物化视图Materialized view，但物化冲突难以使用，应避免

3. 可序列化Serializability

a. 真的串行执行Actual Serial Execution

i. 在存储过程中封装事务Encapsulating transactions in stored procedures

由于用户的输入输出缓慢，并且数据库与应用程序的沟通也未必高效，为了高性能执行事务，可以将事务复杂的过程组合成一个完整的存储过程，由存储过程来执行



但存储过程缺点过多，实际上是被生产环境禁止使用的

ii. 分区Partitioning

通过分区将数据拆分后，若事务涉及的数据能够控制在一个分区内，则每个分区单独串行执行事务，效率也非常高，但是一旦事务跨分区，就需要事务协调者进行整体控制，且代价很大

iii. 序列执行的总结Summary of serial execution

1) 每个事务都足够短小
2) 所涉及的数据最好能全部放在内存中，否则涉及到磁盘IO，事务吞吐量显著下降
3) 读写延迟尽可能低，跨分区事务尽可能少

b. 两阶段锁2PL, Two-Phase Locking

相比于快照隔离（读不阻塞写，写不阻塞读），两阶段锁是互斥锁，读写相互阻塞，但是两阶段锁提供可序列化，避免了竞争、写偏差、更新丢失等问题

i. 实现两阶段锁Implementation of two-phase locking

每个对象都有相应的锁，并且分为共享模式shared mode和互斥模式exclusive mode，如下使用：

1) 读时，获取共享模式的锁，如果其他事务已经持有互斥模式的锁，则获取失败进入等待
2) 写时，获取互斥模式的锁，如果其他事务已经持有共享或互斥模式的锁时，则获取失败进入等待
3) 一旦获得锁，所必须被一直持有直到提交commit或终止abort

所谓两阶段，第一阶段获得锁并持有，第二阶段提交或终止时释放锁，数据库可以自动检测是否出现了死锁，并强制终止事务

ii. 两阶段锁的性能Performance of two-phase locking

由于事务是没有有效期的，一旦事务缓慢，持有的锁无法释放，就会阻塞所有相关的事务，并且更容易出现死锁，死锁导致的事务终止和重试也会带来额外开销

iii. 谓词锁Predicate locks

谓词锁相当于是查询的条件的锁，查询某一个条件如果已经加了锁就要等待，如果查询的条件没有被加锁，就加锁并执行

iv. 索引范围锁Index-range locks

由于条件过多，通过谓词判断来加锁，性能低下，实际上大部分数据库支持了索引范围锁，索引范围锁的封锁对象往往是谓词锁封锁对象的超集，类似于concurrent hash map，每个bucket加锁，如果key落入这个bucket，则对整个bucket加锁

c. 序列化快照隔离Serializable Snapshot Isolation

i. 悲观锁和乐观锁Pessimistic versus optimistic concurrency control

悲观锁认为数据可能已经被持有了，所以首先尝试加锁随后再修改数据；乐观锁认为数据没有被持有，直接修改数据，直到提交时再检查是否有人加锁，若有终止事务并回滚操作，序列化快照隔离是一种乐观操作，即基于快照的方式直接读写数据，当要将快照的修改应用到数据上时检查真正的数据是否和修改前的一致，一致就应用修改，否则回滚事务重试，在竞争激烈时，乐观锁

反而会加重系统负担，但是竞争不激烈时，乐观锁显著提升性能

ii. 基于过时前提的决策Decisions based on an outdated premise

由于事务中对数据的查询和修改可能存在依赖，例如查到什么值就执行什么写入，因此需要检测查到的值在写入时是否已经过时，考虑：

1) 检查是否读了过时的MVCC对象（读之前存在未提交的写）
2) 检查写是否影响之前的读（读之后再进行写）

iii. 检查旧MVCC读取Detecting stale MVCC reads

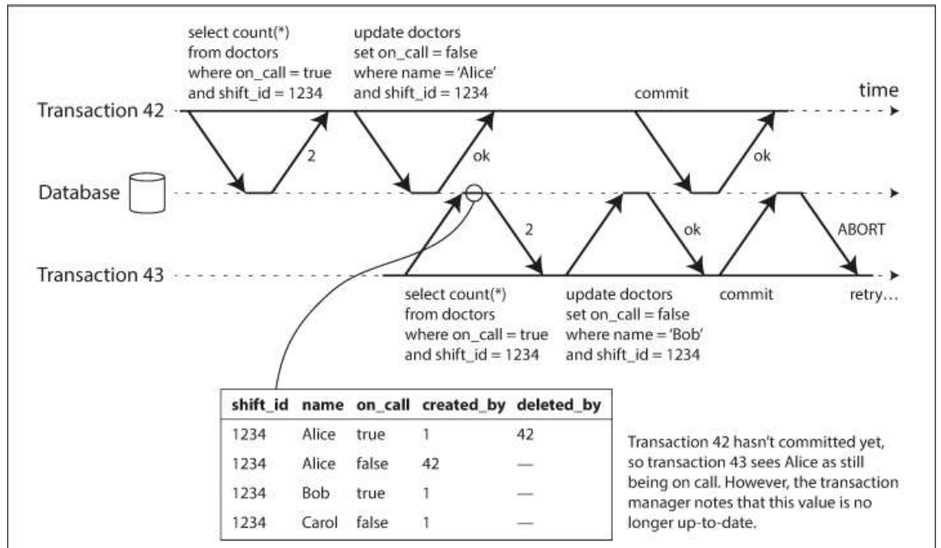


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

数据库需要追踪一个事务是否忽略了另一个事务的未提交写入（基于快照隔离可见性规则），当原事务提交时，就要检测另一个事务的写入是否已经提交了，此时说明另一个事务的写入可能影响到了原事务，原事务需要终止

注意：直到原事务提交时才检测快照是否还有效，是因为原事务如果是只读的（只有事务执行完才能直到这一点），则无需终止；如果追踪的另一个事务并没有提交即原事务的快照依然有效，则也无需终止

iv. 检查写是否影响之前的读Detecting writes that affect prior reads

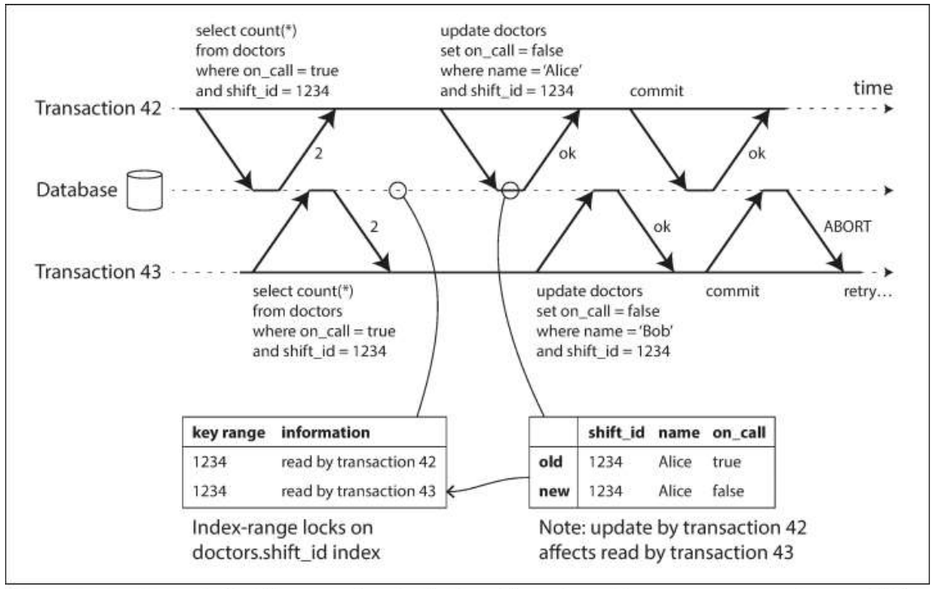


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

读数据时记录数据被那些事务读取，而写数据时根据记录的事务通知其读取的数据已经过时，当某个事务要提交时检查所有影响自己读的数据的事务是否已经提交，如果它们还未提交则自己提交，如果它们已经提交，说明自己读的数据失效，则终止并回滚

序列化快照隔离的性能Performance of serializable snapshot isolation

1) 过细的追踪事务读写，使得可以精细控制事务的终止回滚，但簿记开销就很大，反之则事务回滚重试开销就很大
2) 对于主要负载都是只读的系统来说，与2PL相比，序列化快照隔离既可以提供序列化，又可以使得只读事务完全不被阻塞，性能非常好
3) 由于序列化快照隔离的基本原则就是悲观模式，因此事务的终止比例（竞争激烈度）显著影响了性能

S08 The Trouble with Distributed Systems

2019年7月5日 16:50

1. 故障与部分失效Faults and Partial Failures

a. 云计算和超级计算机Cloud Computing and Supercomputing

云计算往往基于互联网，有以下特点：

i. 云计算往往提供在线服务，一旦不可用是不可接受的，而超级计算机出现故障，整体停机影响不大
ii. 超级计算机往往使用高度可靠的硬件，而云计算使用廉价商用的节点但故障率相对更高
iii. 云计算基于IP和以太网，而超级计算机基于专用网络
iv. 当一个系统大到一定规模时，可以假定任意时刻总有部分不能正常工作（概率）
v. 系统如果对故障有容忍度，并且一定程度内依然可以提供服务是非常重要的
vi. 云计算可以部署在跨地域的数据中心，相互沟通通过不可靠的因特网，而超级计算机往往集中放置

2. 不可靠的网络Unreliable Networks

a. 真实世界的网络故障Network Faults in Practice

i. 请求在网络中丢失
ii. 请求进入队列等待，延迟较高
iii. 远端节点失效，可能已经宕机
iv. 远端节点暂时无法响应，诸如正在进行垃圾回收等，但随后就会重新响应
v. 响应在网络中丢失
vi. 响应进入队列等待，延迟较高

发送者无法判断是哪一种情况，只能通过延迟timeout来判断是否出现了问题，但延迟依然无法判断具体情况

b. 检测故障Detecting Faults

i. 能够到达目标机器，但是目标端口无程序在监听，证明进程已经退出，但是假如正在处理请求时进程崩溃，无法判断原请求处理到什么程度
ii. 如果路由器回送IP不可达，则说明目标机器已经宕机，但是路由器本身也不一定可靠

为了确保请求真的被处理了，应用程序需要有自己的ACK机制，因为TCP的ACK只保证请求送达，而不保证此后程序正确处理了请求

c. 超时和无穷延迟Timeouts and Unbounded Delays

由于延迟没有上界，往往用超时来判断远端是否可用，但是超时的时间难以确定，过长的超时可能需要等待额外的时间，而过短的超时可能导致更高的风险出现误报，在高负载工作时可能延迟更大，此时一旦因为延迟而误报节点死亡，转移功能和负载到其他节点上会导致进一步负载增加，最后出现级联失效cascading failure

注意：TCP执行拥塞控制算法，使得数据包在收发端都可能进行排队，并且TCP本身也有超时重传

d. 同步网络和异步网络Synchronous Versus Asynchronous Networks

3. 不可靠的时钟Unreliable Clocks

a. 单调钟与时钟Monotonic Versus Time-of-Day Clocks

i. 时钟Time-of-day clocks

也称为挂钟wall-clock，通常使用NTP与授时服务器同步，返回的是时间点，因此当时钟比授时服务器慢或快时，会被重置与服务器同步，这意味着时间并不一定单调增加，可能会突然增加或减少（被重置了）

ii. 单调钟Monotonic clocks

单调钟保证一定单调增加，适用于测量时间间隔（经过时间elapsed time），但是单调钟返回的并不是绝对时间时间点，因此都通过单调钟返回的多次值之差来看时间间隔，其绝对值没有意义

b. 时钟同步与准确性Clock Synchronization and Accuracy

单调钟不需要同步，时钟需要同步，但是同步方法本身并不是很可靠，且精度也不如所想之高

c. 依赖同步时钟Relying on Synchronized Clocks

由于同步时钟受NTP影响，可能会出现时间跳跃，同时错误配置的NTP可能导致同步时钟逐渐偏离轨迹

i. 有序事件的时间戳Timestamps for ordering events

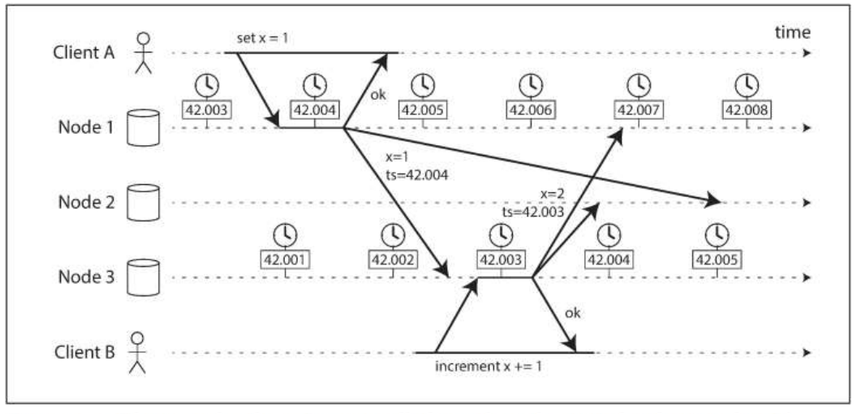


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

由于时钟同步问题，导致事件发生的顺序受到了影响，常见通过最后写者获胜LWW来解决冲突，参考检测并发写入

ii. 时钟读数存在置信区间Clock readings have a confidence interval

iii. 全局快照的同步时钟Synchronized clociks for global snapshots

例如快照隔离要求有全局单调递增的事务TID，在单机上一个原子计数器即可，但是在分布式系统中不能直接使用时间戳作为TID，因为不同系统的同步时钟存在精确度问题，可以通过某一个节点专门生成TID，或者通过采用考虑了精确度问题的时钟，例如时钟同步且精确度在5ms内，则相邻事务的时间戳±5ms这个区间不重叠，就可以确保先后顺序

d. 进程暂停Process Pauses

假定一个单主节点的系统，为了确保主节点确认自己的有效性，可以通过获得并持有一个租约lease来实现，在到期前更新租约的有效时间，当节点宕机后其他节点就能通过获得租约成为新的主节点。但是问题在于租约的超时检查本身可能不及时，例如遇到了垃圾回收的STW, stop the world、虚拟化环境中虚拟机被暂停等情况，导致可能已经超时了，但是"主节点"并没有意识到此时已经有新的主节点出现，这种情况称为进程暂停，分布式系统中需要假设这种情况会发生

i. 响应时间保证Response time guarantees

实时系统要求极高，在一般服务器端数据处理系统来说，实时保证是不经济无必要的，因此这种分布式系统需要考虑到程序暂停或时钟的不稳定性

ii. 限制垃圾回收的影响Limiting the impact of garbage collection

现代语言的GC时间已经非常短暂，并且还可以通过有计划的GC来充分减小影响

4. 知识、真相和谎言Knowledge, Truth, and Lies

a. 真相由多数决定The Truth Is Defined by the Majority

分布式系统中无法由单个节点来确定状况，往往依赖于法定人数quorum

i. 领导者与锁定The leader and the lock

通常一个系统真一些操作只能由一个领导者来确定

1) 只允许一个领导者进行数据库分区，避免脑裂split brain
2) 只允许一个事务或用户持有特定资源或对象的锁，避免并发写入导致数据损坏
3) 每个特定的名字只能由一个用户或服务来注册，避免命名冲突

ii. 防护令牌Fencing tokens

例如一个分布式锁，C1获取租约之后进入了GC，而租约到期后自动解锁但是C1没有感知到，此后C2获得租约，待C1从GC恢复后租约已经过期，但是自以为依然持有锁，出现C1和C2并发修改一个数据

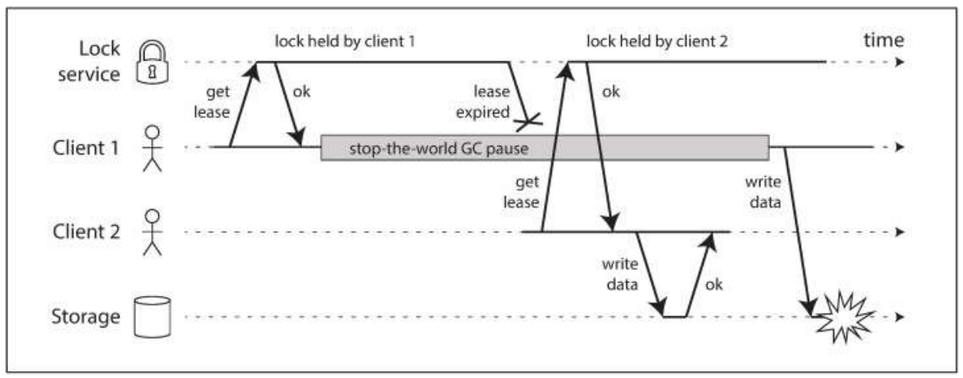


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

通过使用单调递增的锁序列号，将所有获得锁的顺序序列化记录称为防护令牌fencing tokens，一旦接收了更大令牌的控制，就会拒绝所有旧的令牌

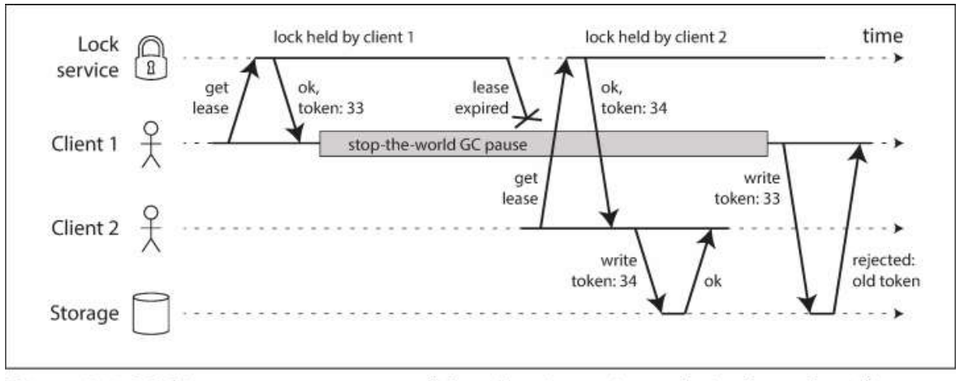


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

b. 拜占庭故障Byzantine Faults

假设所有节点本身不可靠（会做出欺骗行为）的情况称为拜占庭故障，对于如何在有不可信节点的情况下达成所有节点的共识，称为拜占庭将军问题Byzantine Generals Problem

注意：即使大部分情况下可以假设所有节点是可信的，但是依然需要手段来防止意外的数据错误，例如错误配置、比特跳转等，例如

i. 网络传输可能由于硬件错误或bug出现数据损坏，由此TCP/UDP引入了校验和checksum，在引用层也可以引入校验和
ii. 不信任任何用户输入，对用户输入进行合法性校验
iii. 使用多个NTP服务器，每次同步时间交叉验证多个NTP的结果，来确保单个错误NTP服务器不会影响整个系统

c. 系统模型和现实

i. 根据时间的假设，三种常用的模型

1) 同步模型Synchronous model

假定网络延迟有界，程序暂停有界，时钟错误有界，绝大部分情况下并不符合现实

2) 部分同步模型Partially synchronous model

大部分情况下与同步模型一致，但是一些时候网络延迟、程序暂停、时钟漂移会突破界限，比较符合现实情况

3) 异步模型Asynchronous model

没有任何时间上的保证，甚至没有时钟（不能使用超时），少数算法能适用这个模型，非常局限

ii. 节点故障，三种常用的模型

1) 崩溃停止故障Crash-stop faults

节点只存在崩溃故障，一旦崩溃就失去响应且不会再恢复

2) 崩溃恢复故障Crash-recovery faults

节点只存在崩溃故障，但是可以在未知时间内恢复，且假定内存数据被丢失，硬盘数据能够跨越崩溃时间

3) 拜占庭故障Byzantine (arbitrary) faults

节点存在任意故障，包括欺骗其他节点这种形式的故障

绝大多数情况下，系统模型都是部分同步模型+崩溃恢复故障

i. 算法的正确性Correctness of an algorithm

对于算法来说，其所有属性在某个确定的系统模型下，都能够得到满足，则认为其是正确的，例如对于防护令牌fencing tokens

1) 唯一性Uniqueness：任意两个请求返回的值都不相同

2) 单调序列Monotonic sequence：后请求的值一定大于前请求的值

3) 可用性Availability：节点要求获得令牌且自身没有崩溃，则最终一定能获得令牌

ii. 安全性和活性Safety and liveness

安全性Safety可以认为是没有坏事发生nothing bad happens；活性Liveness可以认为是最终好事发生something good eventually happens

iii. 将系统模型映射到真实世界Mapping system models to the real world

S09 Consistency and Consensus

2019年7月7日 19:20

1. 一致性保证Consistency Guarantees

2. 线性一致性Linearizability

线性一致性（又称原子一致性atomic，强一致性strong，立即一致性immediate，外部一致性external consistency）的核心就在于在外部看来多个副本的数据行为表现就和只有一个副本一样，是最强的一致性保证

a. 什么使得系统保证线性一致性What Makes a System Linearizable

所有读取都能读到最近一个成功的写入，所有读取一旦读到了一个值，此后都只会读到更新的值

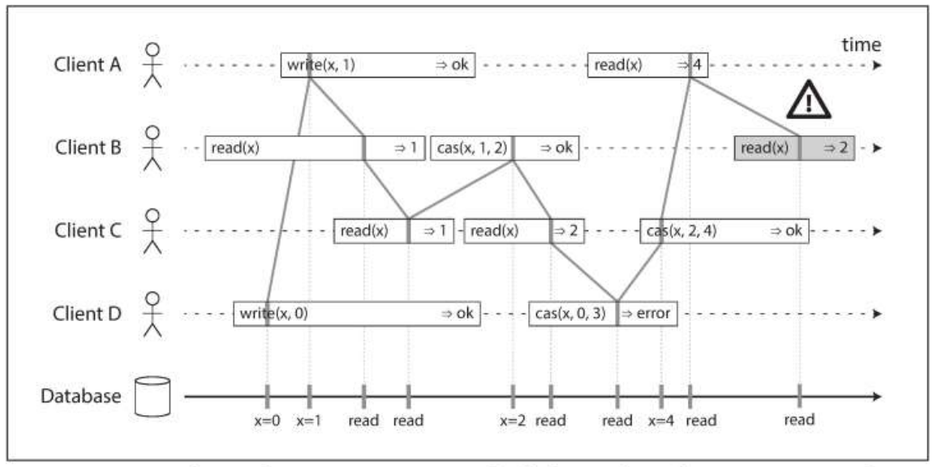


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

i. 途中DB按顺序处理了D的写、A的写、B的读，和接收请求的顺序不一致，这是可以接受的，因为这些请求是并发的

ii. B先获得了X=1的返回值，随后A才获得了X<-1的写入成功，这是可以接收的，因为A获得返回值与DB成功写入1存在网络延迟等

iii. 这个梦想是为了阐述线性一致性，而没有假设任何事务的隔离，因此ABCD都能看到相互的读写且相互影响

iv. 最后由B提出的读返回了X=2是不符合线性一致性的，因为在此之前A已经成功获得X=4

注意区分线性一致性Linearizability和可序列化Serializability：

i. 可序列化是事务隔离等级的一个级别，多个事务可以读写多个对象，可序列化代表这些事务对多个对象的读写结果可以序列化成某一个按顺序执行操作的结果，事务真正执行的操作和序列化后的操作顺序可以不一致，强调的是结果一致

ii. 线性一致性是对一个对象的读写的新鲜度保证，并不将多个操作包装成事务，因此并不保证写偏差等问题

iii. 基于真正串行执行Actual Serial Execution或两阶段锁2PL的可序列化方式，也是线性一致性的

iv. 基于可序列化快照隔离SSI的可序列化方式，并不是线性一致性的，因为快照隔离不会读到快照后发生的写

b. 依赖线性一致性Relying on Linearizability

i. 锁定和领导选举Locking and leader election

例如通过获得一个锁（分布式锁）来证明自己是leader，就需要保证线性一致性，如果多个结点读到的leader不同就会出现脑裂

ii. 约束和唯一性保证Constraints and uniqueness guarantees

iii. 跨信道的时序依赖Cross-channel timing dependencies

c. 实现线性一致性的系统Implementing Linearizable Systems

i. 常见的方法以及线性一致性的可能性
   1) 单主复制（可能线性一致）
   2) 共识算法（线性一致）
   3) 多主复制（非线性一致）
   4) 无主复制（可能非线性一致）

ii. 线性一致性和法定人数Linearizability and quorums

严格的法定人数并不能保证线性一致性

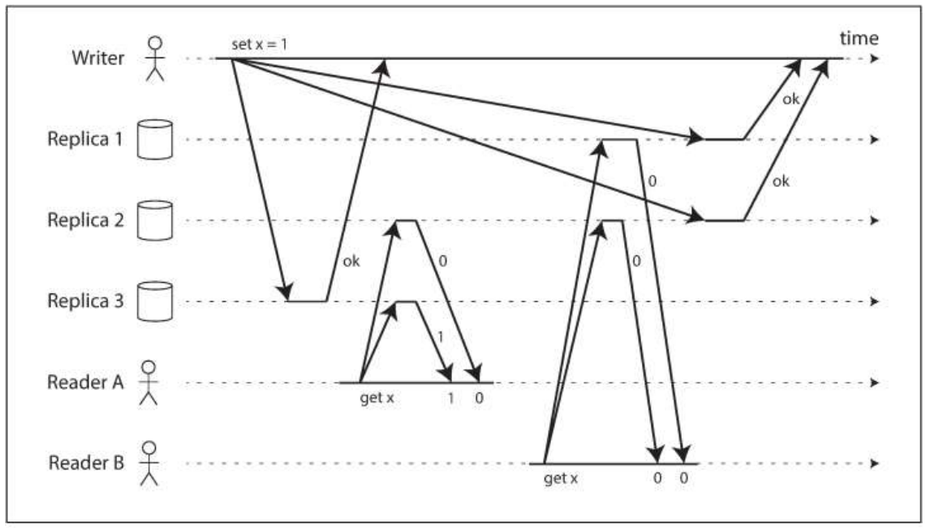


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

d. 线性一致性的代价The Cost of Linearizability

当需要线性一致性时，一些副本由于网络等原因无法连接其他副本，导致失联的副本无法处理读写请求，由此这些副本就不可用了；

当不需要线性一致性时，副本可以独立处理读写请求例如多主复制，由此保证了可用性，但丢失了线性一致性

i. CAP理论（已经过时）

CAP指一致性Consistency（只考虑线性一致性），可用性Availability，分区容错性Partition tolerance（只考虑网络分区），CAP理论指出三者只能选其二，但是CAP理论是无助的，因为网络隔离等带来的网络分区是一种错误，总是会发生而不是一种选择，因此更好的描述是当P发生时CA二选一，但是由于A的定义也非常模糊，事实上在实际中用更准确地描述，而回避CAP的说法

ii. 线性一致性和网络延迟Linearizability and network delays

放弃线性一致性的主要原因在于性能，由于已经从数学上证明读写响应时间至少是正比于网络延迟，因此不可能即获得高性能又保证线性一致性，由此可以考虑为了性能放弃线性一致性

3. 顺序保证Ordering Guarantees

a. 顺序与因果Ordering and Causality

若一个系统服从因果关系规定的顺序，则称为满足因果一致性causally consistent，例如快照隔离提供了因果一致性

i. 因果顺序不是全序The causal order is not a total order

1) 线性一致性Linearizability

所有操作是全序的，对于任意两个操作都一定有先后顺序

2) 因果性Causality

对于有因果关系操作是有序的，例如B依赖A，则根据因果一致性A一定发生在B前，但是对于无因果关系的操作（并发操作）是可以无序的，因此因果一致性是偏序partial ordered而非全序total ordered的

ii. 线性一致性强于因果一致性Linearizability is stronger than causal consistency

因果一致性是能够保证系统不因网络延迟而劣化性能的最强的保证，还比较前沿，目前业界都是用线性一致性来保证因果一致性的

iii. 捕获因果关系Capturing causal dependencies

参考无主复制的检测并发写入与版本号、参考事务隔离界别的可序列化快照隔离

b. 序列号顺序Sequence Number Ordering

使用逻辑时钟logic clock例如单调递增的序列号来标记每一个操作的发生时间点，对于有因果关系的操作确保先发送的序列号小，对于并发的操作则随意分配序列号

i. 非因果序列号生成器Noncausal sequence number generators

ii. Lamport时间戳

每个节点都有一个独特的标识符node ID，且每个节点都有一个计数器counter记录执行过的操作数，则Lamport时间戳就是(counter, node ID)，每个节点和每个用户都记录见过的计数器的最大值，并在每一个请求上都带有这个最大值，当遇到更大的值时立刻更新自己的值为新的最大值，(counter1, node ID1) > (counter2, node ID2) when counter1>counter2 or counter1==counter2 & node ID1> node ID2

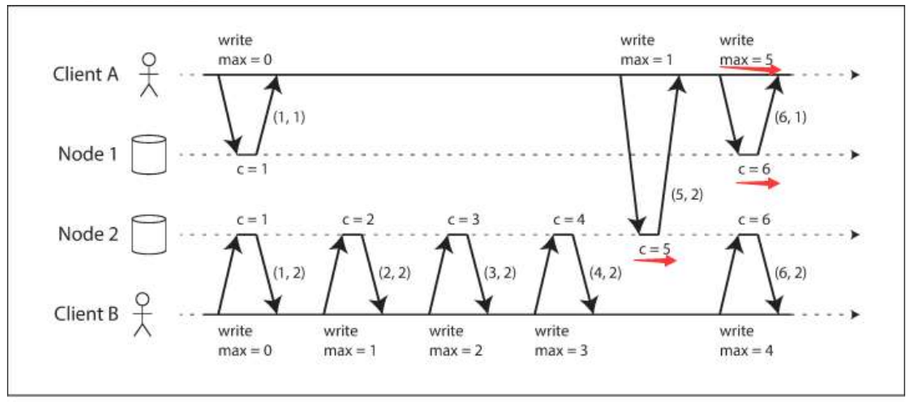


Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

A带着1访问NODE 2时发现计数器是5大于所持有的1，立即更新为5，此后带着5访问NODE 1时，NODE 1立即更新为5，并返回递增后的6给A，由此保证了因果性，因为每一个因果依赖的操作必然带来时间戳的单调递增

注意：Lamport时间戳与版本向量version vectors不同，前者提供一个全序，根据全序无法区分是并发还是依赖，而后者能够区分两个操作是并发的还是因果依赖的

注意：仅仅有Lamport时间戳是不够的，因为基于时间戳获得全序首先需要在所有节点上获得所有操作，假如还没有获得其他节点的操作，则已经发生的操作顺序依然是未知的

c. 全序广播Total Order Broadcast

全序广播的前提条件是满足可靠交付Reliable delivery、全序交付Totally ordered delivery

i. 使用全序广播Using total order broadcast

1) 若数据库的复制时，每条日志也通过全序的方式广播，每个副本按照顺序应用这些日志，则所有副本都能保证相互一致，这也称为复制状态机replicated state machine
2) 若每条广播代表一个事务，则全序后顺序执行的事务，也保证了事务的可序列化，保证了分区和副本的相互一致
3) 使用全序广播也可以实现分布式锁，并且提供防护令牌，全序的序列号可以充当令牌，全序日志就是获得和释放锁的顺序

ii. 使用全序广播实现线性一致性存储Implementing linearizable storage using total order broadcast

全序广播与线性一致性不同，全序广播保证了消息的全序传递，可靠传递，但是不一定要同步，而线性一致性是新鲜度的保证，保证了读取一定能读到最新的值（全序广播不保证这一点，因为接收者是可以落后于广播者的，从而不保证线性一致性），但是

基于全序广播可以实现线性一致性：

1) 全序广播一条尝试写入，随后获得所有广播的消息，如果所写入的值就是自己的尝试值，说明成功写入，若不是则放弃本次操作，由此保证了线性一致性写，因为所有写入被全序广播，只有第一条写入会得到所有节点的认可
2) 全序广播一条尝试读取，随后获得所有广播的消息，等到收到自己的尝试读取，就执行真正的读取，由全序广播确保了自己的尝试读取能够真正读取时，一定保证了都在尝试时其他写之后，由此保证了线性一致性读

iii. 使用线性一致性存储来实现全序广播Implementing total order broadcast using linearizable storage

假定有符合线性一致性的递增计数器，则每需要一次广播时就从递增计数器获取一个值，这个值就是全序广播的序列号，按序列号广播，接收者按序列号接收，中间不存在序列号空洞

注意：全序广播、线性一致性、共识是数学上等同的问题，解决其中任意一个，就可以进而实现其他两个

4. 分布式事务和共识Distributed Transactions and Consensus

a. 原子提交和两阶段提交Atomic Commit and Two-Phase Commit (2PC)

i. 从单个节点到分布式的原子提交From single-node to distributed atomic commit

不能通过leader向所有节点发出commit，随后就commit这种单节点的方式，因为涉及到多个节点，可能会出现部分节点commit成功，部分节点commit失败这种不一致情况（节点崩溃、网络丢失、部分节点冲突等）

ii. 两阶段提交简介Introduction to two-phase commit

两阶段提交需要事务协调器coordinator（aka transaction manager）在所有参与事务的节点participants上执行事务

1) 待提交时，首先预备提交prepare到所有参与者，待都返回yes时再执行提交commit，由此事务完成
2) 若prepare阶段任意一个节点返回no或超时timeout，则执行abort，由此事务终止
3) 若commit阶段有节点崩溃，则事务必须在节点恢复后继续提交，commit的事务不可撤销

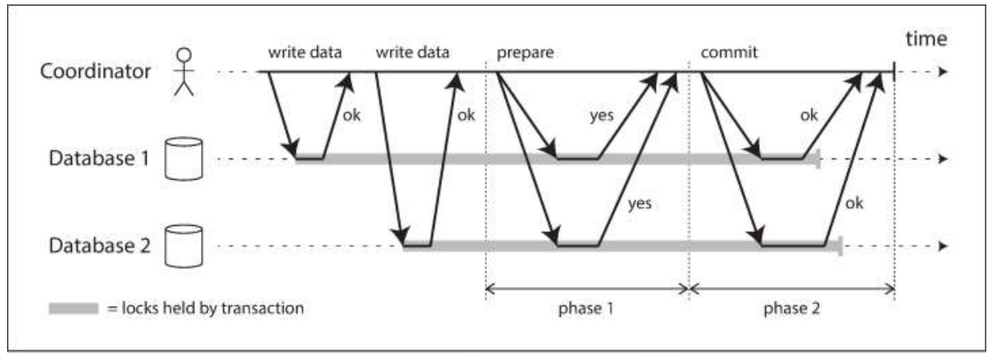


Figure 9-9. A successful execution of two-phase commit (2PC).

iii. 系统承诺A system of promises（两阶段提交细节）

1) 当启动一个分布式事务时，从coordinator处获得一个全局唯一的TID
2) coordinator在每个participant上启动一个单节点事务，并带有这个TID，所有读写都在这些单节点事务上完成，coordinator和任意一个pariticipant都可以在这个阶段abort
3) 准备提交时，coordinator发送prepare请求给所有participants，并带有TID，如果任意participant拒绝或超时，则coordinator发送abort给所有participants终止事务
4) 当participant收到prepare时，进行确定在任意情况下都能commit事务（持久化数据和日志，检查约束等），随后反馈yes给coordinator，自此participant保证一旦收到commit就一定能commit而不会提出abort
5) coordinator收到反馈后来决定commit/abort，并将决定写入持久化日志（用于崩溃恢复，这个节点称为commit point）
6) coordinator持久化决定之后，就将commit/abort发送给所有participants，如果这个决定没有得到yes，就一直重试直到成功，因为决定commit的transaction不允许失败或回退，假设commit时某个participant崩溃，则重启恢复后也要以yes来响应以确保

事务一定能commit

通过两个不可回退点（participant对prepare返回yes时，保证接下来会commit；coordinator决定commit/abort时，保证始终贯彻这个决定直到成功）来实现原子事务提交，而在单节点中两个点被合并到一起（写入commit到持久化日志）

iv. 协调者失效Coordinator failure

如果coordinator在prepare前失效，则participant可以安全abort，但是如果在commit前，prepare之后失效，则participant无从知晓coordinator的决定，从而只能等待，这时pariticipant处于不确定状态uncertain（理论上所有participants可以自行通信尝试达成共识决定commit/abort，但这并非2PC的内容）

v. 三阶段提交Three-phase commit

两阶段提交又称为阻塞原子提交协议，因为2PC需要等待coordinator做出决定，一旦coordinator失效就必须阻塞等待，但是不阻塞的诸如三阶段提交协议要求一个完美的错误检测器perfect failure detector来检测节点是故障与否，这在网络环境中是不可能用超时来做到的，因为超时无法区分是网络延迟还是节点故障，因此2PC依然广泛使用

三阶段分为canCommit, prepareCommit, doCommit三个阶段，且coordinator, participants都引入超时机制，到那时由于超时无法区分延迟还是故障，因此可能出现数据不一致

注意：2PC的coordinator故障是一个单点故障，可以通过使用共识协议来实现coordinator高可用

b. 实践中的分布式事务Distributed Transactions in Practice

i. 两种常见的分布式事务：

1) 数据库内部分布式事务Database-internal distributed transactions：分布式数据库内部使用的分布式事务，意味着所有节点运行相同的程序，可以自由选择协议并针对性的优化，因此数据库内部分布式事务往往运行良好
2) 异构分布式事务Heterogeneous distributed transaction：跨系统的分布式事务，事务的参与者可能运行不同的程序（数据库或非数据库），协调所有参与者进行事务难度更大d

ii. 恰好一次消息处理Exactly-once message processing

采用分布式事务来实现消息队列到数据库的恰好一次消息处理，异构分布式事务

iii. XA事务

XA, eXtended Architecture是实现异构分布式事务2PC的标准，并且被广泛支持，通过在不同的系统上实现XA API就可以支持跨系统的异构分布式事务

iv. 怀疑时持有锁Holding locks while in doubt

在2PC中，当参与者不确定事务状态时，必须持有锁，因为例如2PL中锁的释放只在事务commit/abort时才发生，现在事务状态未知，怀疑时必须持有锁，由此导致了阻塞等待，进而其他涉及相关数据的事务都进入了阻塞等待，整个系统都会变得不可用

v. 从协调者故障中恢复Recovering from coordinator failure

由于有可能出现协调者从崩溃重启后事务日志出错或丢失，此时相关的事务将永远孤立留存，此时只能由手动决定commit/abort，需要仔细检查每个participant是否已经commit/abort

大部分XA事务的实现有一个紧急逃脱方案称为启发式决策heuristic decisions，允许participant在存疑时单方面决定commit/abort，但是这是有可能破坏数据一致性的（参考3PC，例如部分participant决定commit，另一部分决定abort）

vi. 分布式事务的局限性Limitations of distributed transactions

即使XA事务解决了跨系统的异构分布式事务，但是核心在于coordinator本身是数据库，存储事务的结果，因此需要考虑：

1) 如果coordinator本身只是单节点而没有副本时，就是单点故障，因此一旦coordinator崩溃会阻塞其他系统
2) 许多应用程序是无状态的（例如HTTP），由此带来了简单部署和弹性收缩的好处，但是一旦coordinator是这些应用的一部分，系统就发生了本质的变化，coordinator的日志非常重要，进而这些程序也不再无状态
3) 由于XA事务需要跨系统，那么势必是最基础的部分，因此不能支持例如可序列化快照隔离（如何解决不同系统快照的冲突？）、不能解决死锁（如何判断不同系统上的锁和依赖关系？）

4) 由此可见，对于数据库内部分布式事务，局限性小得多
5) 分布式事务有一个放大故障amplifying failures的趋势，即系统一个部分的损坏会逐渐扩大导致整个系统不可用

c. 容错共识Fault-Tolerant Consensus

共识算法要求满足以下属性：一致同意（Uniform aggrement，节点之间不做出不同的决定），完整性（Integrity，节点不决定两次），有效性（Validity，如果节点决定了V，那么V一定是被某个节点提出的），终止（Termination，未崩溃的节点达到法定人数一定会最终做出决定，2PC不符合这一点）

i. 共识算法与全序广播Consensus algorithms and total order broadcast
   1) 一致同意：所有节点按相同顺序交付相同消息，全序
   2) 完整性：消息不会重复，恰好一次，全序
   3) 有效性：消息不会损坏，全序
   4) 终止：消息不会丢失，全序
ii. 单主复制与共识Single-leader replication and consensus
iii. 时代编号与法定人数Epoch numbering and quorums

   共识算法保证在一个时代编号（Raft内的term）内只有一个leader，当当前的leader被认为失效时，就增加时代编号并选举一个新的leader，因此时代编号也是单调递增且全序的，并且高时代编号的leader可以胜过低时代编号的leader，leader通过法定人数quorum选票来确定自己是否能够做出决定，只有过半节点支持，leader才能够前进
iv. 共识的局限性Limitations of consensus

   共识算法提供了容错（只要过半节点正常工作）、全序广播、安全保证，但是代价是性能，共识算法的投票过程本质上是同步复制过程，一旦出现网络分区等，被分隔较小的节点群都被阻塞，并且在高延迟的网络环境中，可能出现频繁的超时影响整体性能例如Raft有个特别的leader跳跃问题：

   当所有节点之间网络都良好，出了特别的AB之间网络中断，且A是leader，由此会出现B收不到A的心跳而发起选举，由于所有其他连接良好，B能够获胜成为leader，重复此过程A也成为leader，随后出现系统抖动，leader反复跳跃（prevote解决）

d. 成员和协调服务Membership and Coordination Services

i. 特性
   1) 线性一致性原子操作Linearizable atomic operations

      例如由共识算法结合CAS操作来实现分布式锁（通常是租约lease），共识算法保证了锁的获取和释放是线性一致的且容错
   2) 全序操作Total ordering of operations

      例如单调递增的TID，由共识算法来给所有事务提供单调递增的TID，且这个提供TID的服务是容错且线性一致性的
   3) 失效检测Failure detection

      用户与服务维持一个长session且定期发心跳，一旦上一次心跳时间超过了session的超时值，则服务就可以判定session失效并释放所有资源
   4) 变更通知Change notifications

      用户的加入与退出，作为一种成员变更事件，可以可靠的通知给其他用户或系统
ii. 将工作分配给节点Allocating work to nodes
iii. 服务发现Service discovery
iv. 成员服务Membership services