大数据计算基础

BIG DATA COMPUTING

刘显敏 海量数据计算研究中心
liuxianmin@hit.edu.cn

大数据计算系统 2025年秋

大数据算法需要好系统支撑 2

- ✓ 适合大数据计算软硬件平台
- ✓ 有效的资源管理方法
- ✓ 高效的数据存取方法
  - 数据存储结构
  - 数据分布策略
  - 数据索引方法
- ✓ 编写适用于大数据的“好程序”
  - 避免使用系统垃圾回收机制
  - 减少内存拷贝
  - 减少数据重分布次数
  - 减小重分布数据量

大数据计算系统 3

大数据计算系统

- 用于处理大数据的完整的、正在工作的计算机，包括计算机以及完成某项大数据计算任务所需要的软件和周边设备
- 大数据计算系统最重要特点是所有部件都潜在和其他部件交互
- 大数据计算系统面向数据密集型应用，即数据是其主要挑战(数据量、数据复杂度或数据变化速度快)，与之相对的是计算密集型应用，即处理器速度是其瓶颈。

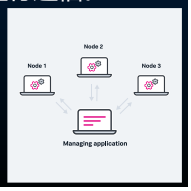
- **数据密集型应用**
  - 使用数据并行方法来处理大数据（通常为TB或PB大小）。
  - 使用数据并行开发应用程序的关键问题：算法的选择、数据分解策略、处理节点的负载平衡、消息传递节点之间的通信以及结果的整体准确性

大数据计算系统 4

分布式大数据计算系统

- 物理上分离但通过配备分布式系统软件的集中式计算机网络连接的自治计算机系统的集合。
- 自治计算机将通过共享资源和文件并执行分配给它们的任务在每个系统之间进行通信。

优点
- 更好的性价比和灵活性
- 更短的响应时间和更高的吞吐量
- 更高的可靠性和可用性

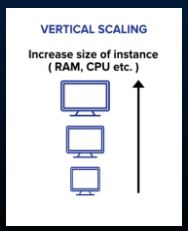


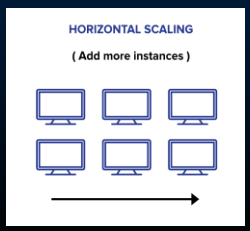
大数据计算系统的挑战 5

- 可靠性/容错性: 能够处理系统故障而不损失数据以及降低性能
- 可扩展性:
  - 可以适用于数千台机器
  - 可以轻易地添加机器
- 可维护性：系统容易维护
- 一致性: 保持数据一致性即使某个结点失效或信息丢失
- 不均匀性(例如p2p系统):
  - 延迟: 1ms 到 1000ms
  - 带宽: 32Kb/s 到100Mb/s

大数据计算系统的挑战 6

- **可扩展性**：可以适应数据增长的快速变化，无论是流量还是数量。
- **有两种常用的数据扩展类型：纵向扩展和横向扩展**





大数据计算系统的挑战

7

- **一致性**：每个节点/副本在给定时间点都具有相同的数据视图。

根据操作并发性与操作顺序之间的权衡，可以采用不同的一致性模型

✓ **其中最弱的一致性保证就是最终一致性**

在分布式系统中，副本最终会收敛到相同的状态。如果给定数据项没有正在进行写入操作，最终一致性可保证所有副本开始使用最后更新的值来服务读取请求。

8

数据复制

Replication of Data

9

目 录

Contents

数据的复制

10

复制: 在网络连接的多台机器上保留相同数据的副本

数据复制的动机

- 减少延迟
  - 使得数据与用户在地理上接近
- 提升可用性
  - 即使系统的一部分出现故障，系统也能继续工作
- 提高读取的吞吐量
  - 扩展可以接受读请求的机器器数量

数据无更新⇒简单　　数据有更新?

主从复制　**多主复制**　**无主复制**

11

目 录

Contents

数据的副本

12

副本（replica）假设每个机器可以存储完整数据集

存储完整数据库的每个节点称作一个副本

- 当存在多个副本时，如何保证**一致性**?
  - 确保所有数据都落在了所有的副本上
  - 数据库的写入操作需要传播到所有副本上

主从复制：基于领导者的复制（leader-based replication）

也称主动/被动（active/passive)或

主/从（master/slave）复制

主从复制

13

工作原理

- 副本之一被指定为领导者，也称为主副本（主节点）。当客户端要向数据库写入时，它必须将请求发送给领导者，领导者会将新数据写入其本地存储。

主从复制

14

工作原理

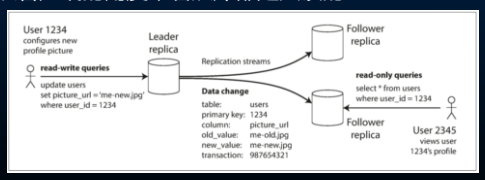
- 其它副本被称为追随者，亦称为从副本（从节点）。当领导者将新数据写入本地存储时，它会将数据变更（日志）发送给所有追随者，称为复制日志记录。每个追随者获取日志后，相应更新本地存储，要求按照相同顺序应用所有写入操作。

主从复制

15

工作原理

- 当客户想要从数据库中读取数据时，它可以向领导者或追随者查询，但只有领导者才能接受写操作。
- 从客户端的角度来看从库都是只读的



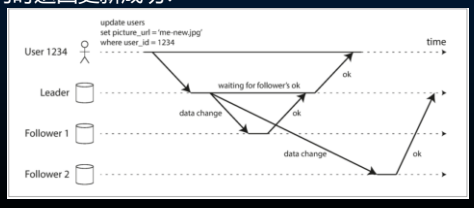
基于领导者(主-从)的复制

同步复制与异步复制

16

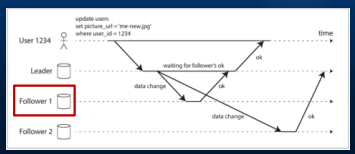
一个重要细节 **复制是同步发生还是异步发生**

- 情景：网站用户更新个人头像。客户向主库发送更新请求；主库收到请求并更新存储；主库会将数据变更转发给从库。最后，主库通知客户更新成功。
- 何时返回更新成功？



同步复制与异步复制

17

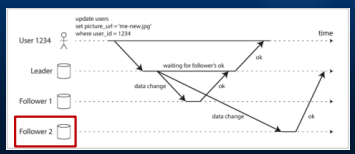


- 上图中从库1的复制是同步的：

  在向用户报告写入成功，并使结果对其他用户可见之前，主库需要等待从库1的确认，确保从库1已经收到写入操作并完成更新。

同步复制与异步复制

18



- 上图中从库2的复制是异步的：

  主库发送消息，但不等待从库的响应。

  有些情况下，从库可能落后主库几分钟或更久；例如：从库正在从故障中恢复，系统在最大容量附近运行，或者如果节点间存在网络问题

同步复制 19

优点

- 从库保证有与主库一致的最新数据副本。如果主库突然失效，我们可以确信这些数据仍然能在从库上找到

缺点

- 如果同步从库没有响应（比如它已经崩溃，或者出现网络故障，或其它任何原因），主库就无法处理写入操作。
- 主库必须阻止所有写入，并等待同步副本再次可用

因此，将所有从库都设置为同步的是不切实际的：任何一个节点的中断都会导致整个系统停滞不前。

同步复制 20

半同步

- 实际上，如果在数据库上启用同步复制，通常意味着其中一个跟随者是同步的，而其他的则是异步的；
- 如果同步从库不可用或缓慢，则使一个异步从库同步；
- 这保证至少在两个节点上拥有最新数据副本：主库和同步从库。

设置新从库 21

设置一个新的从库

- 可能需求：增加副本的数量，替换失败的节点
- 可能需求：提高读取性能

如何确保新的从库拥有主库数据的精确副本？

简单地将数据文件从一个节点复制到另一个节点 ✗

- 客户端不断向数据库写入数据，数据总是在不断变化，标准的数据副本会在不同的时间点总是不一样。复制的结果可能没有任何意义

设置新从库 22

通过锁定数据库(使其不可用于写入)来使磁盘上的文件保持一致

- 违背高可用的目标

按照如下方式，不停机、不中断服务，完成更新

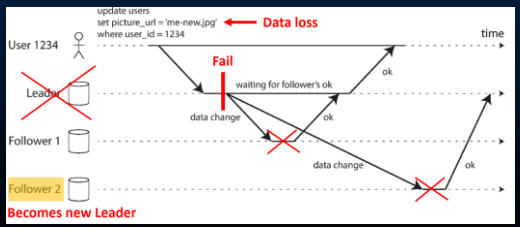
基本过程

1. 获取主库的**一致性快照**（如果可能），而不必锁定整个数据库，大多数数据库都具有这个功能；
2. 将快照复制到新的从库节点；
3. 从库连接主库，**拉取快照之后发生的所有数据变更**。这要求快照与主库复制日志中的位置精确关联；
4. 当从库处理完快照之后积压的数据变更，我们说它**赶上**了主库，形成新的从库。

处理节点宕机 23

系统中的任何节点都可能宕机

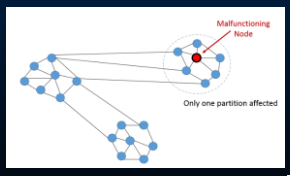
可能因为意外的故障，也可能由于计划内的维护



处理节点宕机 24

目标

- 即使个别节点失效，也能保持整个系统运行，并尽可能控制节点停机带来的影响



- 如何通过基于主库的复制实现高可用？

处理节点宕机 25

从库失效：追赶恢复

如果从库崩溃并重新启动，或者，如果主库和从库之间的网络暂时中断，则比较**容易恢复**：

- 从库可从**日志**知道发生故障之前处理的最后一个事务
- 从库可以连接到主库，并请求在从库断开连接时发生的所有数据变更
- 当应用完所有这些变化后，它就赶上了主库，并可以像以前一样继续接收数据变更流

处理节点宕机 26

主库失效：故障切换

主库失效处理起来相当棘手：

- 其中一个从库需要被提升为新的主库，需要重新配置客户端，以将它们的写操作发送给新的主库，其他从库需要开始拉取来自新主库的数据变更。这个过程被称为**故障切换**
- 故障切换可以手动进行（通知管理员主库挂了，并采取必要的步骤来创建新的主库）或自动进行

处理节点宕机 27

自动故障切换过程

1. 确认主库失效(崩溃，停电，网络问题等)：大多数系统只是简单使用超时：节点频繁地相互来回传递消息，并且如果一个节点在一段时间内（例如30秒）没有响应，就认为它挂了（因为计划内维护而故意关闭主库不算）；
2. 选择一个新的主库：可以通过选举过程（主库由剩余副本以多数选举产生）来完成，或者可以由之前选定的控制器节点来指定新的主库。主库的最佳人选通常是拥有旧主库最新数据副本的从库（最小化数据损失）；
3. 重新配置系统以启用新的主库：如果老领导回来，系统需要确保老领导认可新领导，成为一个从库

处理节点宕机 28

故障切换可能出现的问题

- 如果使用异步复制，则新主库可能没有收到老主库宕机前最后的写入操作。
- 选出新主库后，如果老主库重新加入集群，新主库在此期间可能会收到冲突写入，那这些写入该如何处理？
- 最常见的解决方案是简单丢弃老主库未复制的写入，这很可能打破客户对于数据持久性的期望
- 如果数据库需要和其他外部存储相协调，那么丢弃写入内容是极其危险的操作

处理节点宕机 29

故障切换可能出现的问题

- 发生故障时可能会出现两个节点都以为自己是主库的情况。
- 这种情况非常危险：如果两个主库都可以接受写操作，却没有冲突解决机制，那么数据就可能丢失或损坏。
- 一些系统采取了安全防范措施：当检测到两个主库节点同时存在时会关闭其中一个节点，但设计粗糙的机制可能最后会导致两个节点都被关闭

复制日志的实现 30

基于主库的复制底层是如何工作的?

❶基于语句的复制

- 主库记录下它执行的每个写入请求（语句）并将该语句日志发送给其从库。
- 对于关系数据库来说，这意味着每个INSERT，UPDATE或DELETE语句都被转发给每个从库，每个从库解析并执行该SQL语句，就像从客户端收到一样

这样的方式虽然简单，但是会出现什么问题?

复制日志的实现 31

基于语句的复制的缺点

- 任何调用非确定性函数的语句，可能会在每个副本上生成不同的值
- 如果语句使用了自增列，或者依赖于数据库中的现有数据，则必须在每个副本上按照完全相同的顺序执行它们，否则可能会产生不同的效果。当有多个并发执行的事务时，这可能成为一个限制
- 有副作用的语句（例如，触发器，存储过程，用户定义的函数）可能会在每个副本上产生不同的副作用，除非副作用是绝对确定的

复制日志的实现 32

❷传输预写式日志（WAL）

- NoSQL类型平台
  - 日志结构存储引擎，日志是主要的存储方式
  - 日志段在后台压缩，并进行垃圾回收
- 关系数据平台
  - 基于磁盘块存储（如B树），每次修改都会先写入预写式日志，以便崩溃后索引能恢复到一个一致的状态
- 日志都是数据库写入的追加字节序列
  - 可用完全相同的日志在另一个节点上构建副本
  - 除了将日志写入磁盘，主库还通过网络将其发送给从库
  - 从库应用日志，建立和主库一模一样的副本

复制日志的实现 33

传输预写式日志会出现什么问题?

传输预写式日志的缺点

- 日志记录的数据非常底层
- WAL包含哪些磁盘块中的哪些字节发生了更改
- 这使得复制与存储引擎紧密耦合，存储格式稍有变化（如版本升级），日志失效

复制日志的实现 34

❸基于数据条目(行)的**逻辑**日志复制

- 复制和存储使用不同的日志格式，这样可以使复制日志从存储引擎内部分离出来，被称为**逻辑日志**

逻辑日志：以行的粒度描述对数据库表的写入请求

- 行插入：日志包含所有列的新值；
- 行删除：日志包含“能唯一标识已删除的行”的信息，如主键，或者所有列的值；
- 行更新：日志包含足够的信息来唯一标识更新的行，以及所有列的新值（或至少所有已更改列的新值）

复制日志的实现 35

逻辑日志复制有什么优点?

逻辑日志复制的优点

- 可以更容易向后兼容，从而使领导者和跟随者能够运行不同版本的数据库软件甚至不同的存储引擎
- 对于外部应用程序来说，逻辑日志格式也更容易解析

复制日志的实现 36

❹基于触发器的复制

- 将复制移动到应用程序层
- 更灵活，可以复制部分数据

工作原理

- 触发器注册在数据库系统中，当发生数据更改（写入事务）时，自动执行自定义应用程序代码
- 可以实现将更改记录到一个单独的表中
- 后续可以使用外部程序读取这个表，再加上业务逻辑处理，将数据变更复制到另一个系统去

复制日志的实现

37

基于触发器的复制会出现什么问题?

基于触发器的复制的缺点

- 比其他复制方法具有更高的开销
- 比数据库的内置复制更容易出错

38

目 录

Contents

复制延迟问题

39

回想一下数据复制的原因

- 故障容忍、可扩展性（处理比单个机器更多的请求）和减少延迟（让副本在地理位置上更接近用户）

主从复制

- 所有写入由单个主库处理，读查询可由任意副本处理
- 对于读多写少场景（互联网），可以创建很多从库，将读请求分散到所有从库上去，减小主库的负载，并允许向最近的副本发送读请求

局限 **增加主库⇒增加可扩展性?**

- 只适于异步复制
- 如果尝试同步复制到所有从库，则单个节点故障或网络中断将使整个系统无法写入

复制延迟问题

40

实际应用中仅保障最终一致性（无法完全同步）

- 应用程序从异步从库读取时，如果从库落后，它可能会看到过时的信息
- 这会导致数据库中出现明显的不一致：同时对主库和从库执行相同的查询，可能得到不同的结果，因为并非所有的写入都反映在从库中。
- 这种不一致只是暂时的——如果停止写入数据库并等待一段时间，从库最终会赶上并与主库保持一致

复制延迟

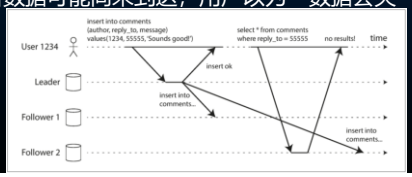
- 写入主库到反映至从库之间的延迟
- 延迟太大? **读己之写** **单调读** **前缀一致读**

复制延迟问题-读己之写

41

问题情景

- 应用让用户提交一些数据，然后查看提交的内容
  - 提交新数据时，必须将其发送给主库
  - 但是当用户查看数据时，可以读取从库
- 对于异步复制，如果用户在写入后马上就查看数据，则新数据可能尚未到达，用户以为“数据丢失”



复制延迟问题-读己之写

42

如何解决？读写一致性（写后读一致性）

- 用户重新加载页面，总看到自己提交的更新；其它用户无保障

实现方法

❶读用户可能已经修改过的内容时，都从主库读

- 简单规则：从主库读取用户自己的档案，在从库读取其他用户的档案

❷如果应用中大部分内容都可能编辑，可以用其它规则

- 如，更新一分钟内从主库读，或监控从库的复制延迟

❸客户端记住最近一次写入的时间戳，随读请求发送

- 从库响应查询时，若该时间戳前的变更都已经复制，则回答查询
- 否则，可以从另一个从库读，或者等待从库追赶上来

❹如果副本分布在多个数据中心，需将请求路由到包含主库的数据中心

复制延迟问题-读己之写 43

更复杂的情况

- 同一个用户从多个设备请求服务，需要提供跨设备的写后读一致性

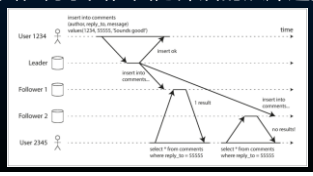
需要考虑的问题

- 记住用户上次更新时间戳变得更加困难，因为一台设备上运行的程序不知道另一台设备上发生了什么。
- 如果副本分布在不同的数据中心，很难保证来自不同设备的连接会路由到同一数据中心

复制延迟问题-单调读 44

时光倒流问题

- 用户从不同从库进行多次读取
- 下图中用户2345两次进行相同的查询，首先查询了一个延迟很小的从库，然后是一个延迟较大的从库
- 第一个查询返回最近由用户1234添加的评论，但是第二个查询不返回任何东西，因为滞后的从库还没有拉取写入内容



复制延迟问题-单调读 45

单调读

- 如果一个用户顺序地进行多次读取，则他们不会看到时间后退，即，如果先前读取到较新的数据，后续读取不会得到更旧的数据

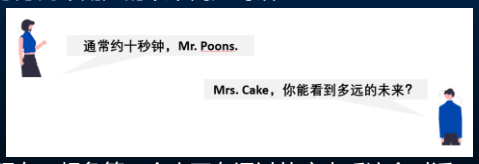
实现方式

- 实现单调读取的一种方式是确保每个用户总是从同一个副本进行读取（不同的用户可以从不同的副本读取）
- 例如，可以基于用户ID的散列来选择副本，而不是随机选择副本
- 但是，如果该副本失败，用户的查询将需要重新路由到另一个副本（仍然可能有问题）

复制延迟问题-前缀一致读 46

问题情景

- 考虑两个用户的以下简短对话：

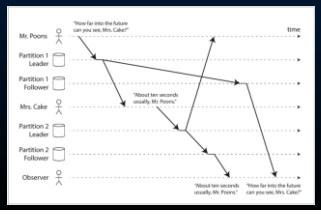


- 现在，想象第三个人正在通过从库来听这个对话
- Cake夫人说的内容是从一个延迟很低的从库读取的，但Poons先生所说的内容，从库的延迟要大的多
- 于是，这个观察者观察到的就是这样的对话

复制延迟问题-前缀一致读 47

前缀一致读

- 如果一系列写入按某个顺序发生，那么任何人读取这些写入时，也会看见它们以同样的顺序出现
- 一种解决方案是，确保任何因果相关的写入都写入相同位置



复制延迟问题 48

- 在实际中如何解决?
- 避免：设计时以为是同步，实际上异步
- 思考延迟带来的后果
  - 几分钟?
  - 几小时?
- 采取更强的一致性要求
- 应用层打补丁解决
- 分布式事务

49

目录

Contents

多主复制

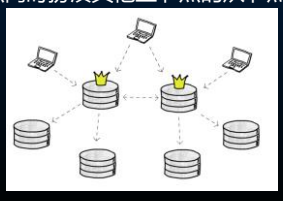
50

单主复制

- 只有一个主库，而所有的写入都必须通过它

多主复制

- 允许多个主节点接受写入
- 每个主节点同时扮演其他主节点的从节点



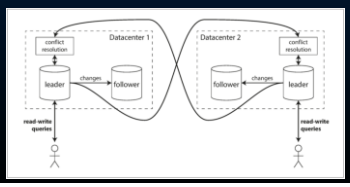
应用场景

51

单个数据中心：需求不大、复杂程度远超好处

多个数据中心

- 假如有数据副本分散在好几个不同的数据中心；
- 每个数据中心都有主库，内部使用常规主从复制；
- 数据中心之间，主节点之间协作、复制

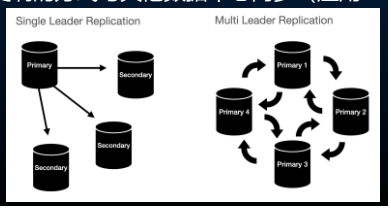


单主复制 v.s. 多主复制

52

性能

- 单主：每个写入都必须穿过互联网，进入主库所在的数据中心，偏离了设计初衷（就近访问）
- 多主：每个写操作都可以在本地数据中心进行处理，然后采用异步复制的方式与其他数据中心同步（应用层透明）

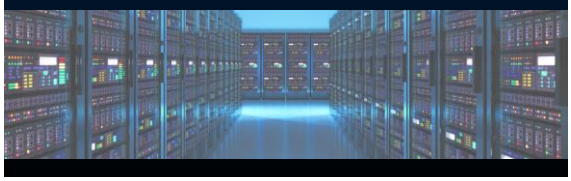


单主复制 v.s. 多主复制

53

容忍数据中心级别的失效

- 单主：如果主库所在的数据中心发生故障，故障切换可以使另一个数据中心里的某个从节点成为主节点
- 多主：每个数据中心独立运行，数据中心发生故障后，恢复、更新即可



单主复制 v.s. 多主复制

54

容忍网络问题（数据中心之间的广域网）

- 单主：对数据中心间的连接问题非常敏感，因为通过这个连接进行的写操作是同步的。
- 多主：数据中心间是异步复制的，通常能更好承受网络问题：临时的网络中断并不会妨碍正在处理的写入

应用场景

55

需要离线操作的客户端

- 应用程序在断网之后仍然需要继续工作
- 每个设备有充当主节点的本地数据库（处理写请求）
- 不同设备同步时，异步的多主复制过程

协同编辑

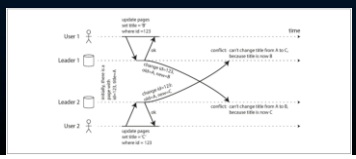
- 实时协作编辑应用允许多个人同时编辑文档
- 用户编辑时，更改立即应用到本地副本（Web浏览器等），并异步复制到server和编辑同文档的其他用户
- 多主复制方法允许多个用户同时进行编辑
- 解决编辑冲突是其中的关键挑战

处理写冲突

56

问题情景 **多主复制最大的问题是写冲突**

- 两个用户同时编辑维基页面
- 用户1将页面标题从A更改为B
- 用户2同时将标题从A更改为C
- 每个用户的更改已成功应用到其本地主库
- 当异步复制时，会发现冲突

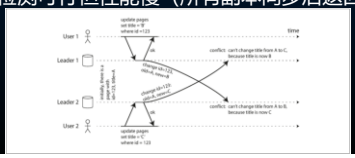


处理写冲突--冲突检测

57

同步 v.s. 异步冲突检测

- 单主复制
  - 第二个写入将被阻塞，并等待第一个写入完成
  - 或中止第二个写入事务，强制用户重试
- 多主复制
  - 两个写入都是成功的，稍后异步复制中检测冲突
  - 同步检测可行但性能慢（所有副本同步后返回成功）



处理写冲突--避免冲突

59

避免冲突

- 确保特定记录的所有写入都通过同一个数据中心的主节点，那么冲突就容易处理
- 例如，应用程序中，用户更新自己数据时（如头像），确保编辑请求始终路由到一个特定的数据中心
- 不同用户可能有不同的指定数据中心（根据用户地理位置），但单个用户角度看，其实是单主复制
- 需要考虑和处理数据中心失效或者迁移的问题

处理写冲突--收敛至一致状态

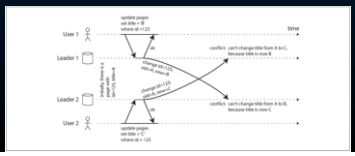
60

很多应用中，冲突不可怕，最终状态一致即可

单主复制：最后一个更新即是一致状态

多主复制

- 写入顺序没有定义，所以最终值是什么并不清楚
- 如下图：主库1标题首先更新为B而后更新为C；在主库2中，首先更新为C，然后更新为B



处理写冲突--收敛至一致状态

61

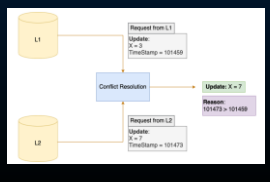
问题矛盾

- 如果每个副本只是按照它看到写入的顺序写入，那么数据库最终将处于不一致的状态
- 因此，数据库必须以一种收敛的方式解决冲突，这意味着所有副本必须在所有变更复制完成时收敛至一个相同的最终值

处理写冲突--收敛至一致状态 62

解决方式A

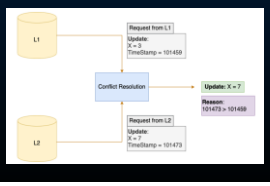
- 给每个写入一个唯一的ID（例如，一个时间戳，一个长的随机数，一个UUID或者一个键和值的哈希）
- 挑选最高ID的写入作为胜利者，并丢弃其他写入
- 如果使用时间戳，该技术被称为最后写入胜利（LWW）
- 会丢失数据



处理写冲突--收敛至一致状态 63

解决方式B

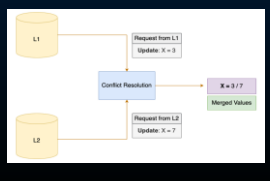
- 为每个副本分配一个唯一的ID，ID编号更高的写入具有更高的优先级
- 与LWW类似，会有数据丢失



处理写冲突--收敛至一致状态 64

解决方式C

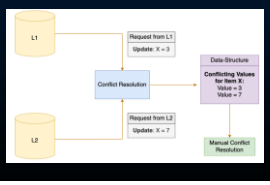
- 以某种方式将这些值合并在一起 - 例如，按字母顺序排序，然后连接它们



处理写冲突--收敛至一致状态 65

解决方式D

- 在保留所有信息的显式数据结构中记录冲突，并编写解决冲突的应用程序代码（也许通过提示用户的方式）



处理写冲突—应用层 66

自定义冲突解决逻辑

大多数多主复制工具允许使用应用程序代码编写冲突解决逻辑。该代码可以在写入或读取时执行

- 写时执行：只要数据库系统检测到复制更改日志中存在冲突，就会调用冲突处理程序
- 读时执行：当检测到冲突时，所有冲突写入被存储。下一次读取数据时，会将这些多个版本的数据返回给应用程序。应用程序可能会提示用户或自动解决冲突，并将结果写回数据库

多主复制的拓扑结构 67

多主复制的拓扑

描述写入从一个主节点传播到另一个主节点的通信路径

- 环形拓扑：每个节点接收来自一个节点的写入，并将这些写入（加上自己的任何写入）转发给另一个节点
- 星形拓扑：指定的根节点将写入转发给所有其他节点
- 全部到全部：每个领导者将其写入每个其他领导

68

目录

Contents

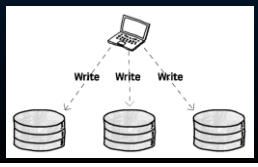
无主复制

69

无主复制

一些数据存储系统采用不同的方法，放弃主节点概念，允许任何副本直接接受客户端写入，即无领导

- 一种可能实现，客户端直接写入几个副本中
- 另一种实现，一个协调者节点代表客户端进行写入
  - 与主节点不同，协调者不执行特定的写入顺序



节点失效时写入数据库

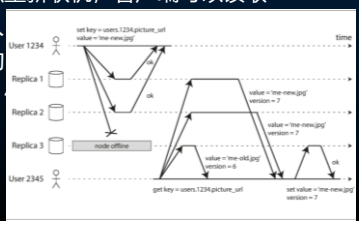
70

问题情景

- 假设有三个副本，而其中一个副本目前不可用
- 在无主复制中，不需要故障切换，写请求发送到所有副本，当收到两个确认回复后，即认为写入成功

假如，不可用的节点重新联机，客户端可以读取

- 节点关闭时的写入都丢失，该节点的数据陈旧（过时）
- 解决：根据版本号确定哪个值新



节点失效时写入数据库

71

节点失效引起数据陈旧该如何处理?

读修复(频繁读取适用)

- 当客户端并行读取多个节点时，可以检测到陈旧数据
- 将新值写入

反熵(主动修复)

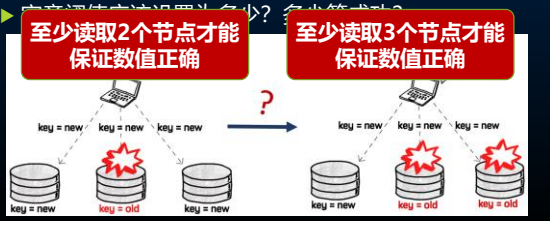
- 数据存储的后台进程不断查找副本之间的数据差异，并将丢失的数据更新从一个副本复制到另一个副本

读写的阈值

72

问题情景

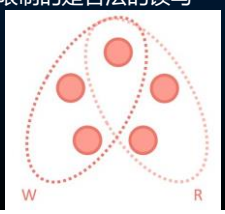
- 在之前的例子中，我们认为即使仅在三个副本中的两个上进行处理，写入仍然是成功的
- 如果三个副本中只有一个接受了写入，会怎样?
- [illegible]



读写的阈值

73

- 更一般地说，如果有n个副本，写入须由w个节点确认才能被认为是成功的，每次读取至少r个副本
- 那么只要**w + r > n**，就能期望在读取时获得最新的值，因为r个读取中至少有一个节点是最新的
- 遵循上述阈值限制的是合法的读写

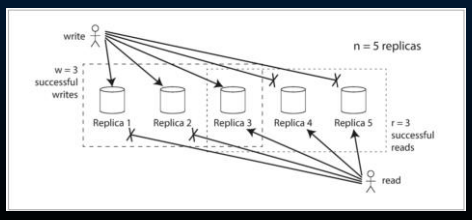


读写的阈值

74

仲裁条件w + r> n允许系统容忍不可用的节点

- 如果w<n，且有节点不可用，仍可能处理写入
- 如果r<n，且有节点不可用，仍可能处理读取
- n=5，w=3，r=3，可以容忍两个不可用的节点



上述做法的局限性

75

仲裁一致性局限性：即使w + r > n，也可能返回陈旧值

- 若使用宽松阈值要求，w个写入和r个读取可能不重叠
- 若两个写入同时发生，则无法判断；根据时间戳（最后写入胜利）挑选，由于时钟偏差，写入可能会丢失
- 若读、写同时发生，写操作仅影响某些副本，则可能读到旧值
- 若写操作部分成功，部分失败，成功副本数目小于w，整体判定写入失败，需回滚，可能读到“未被回滚”的新值
- 若携带新值的节点失效，后续从带有旧值的副本中恢复，则存储新值的副本数可能会低于w

宽松阈值要求

76

上述做法可以容忍故障，但对网络失效仍然容忍度不够

问题情景

- 大型集群中，节点数远大于n，客户使用特定节点群
- 若网络中断，客户可能与大量节点失连
- 客户端仍能连接到某些节点，但达不到阈值要求

在这种情况下，怎么办？

- 达不到阈值要求，则返回错误提示；
- 或者，对于写请求接受，写入一些可达的节点？但，这些节点可能不包含在特定节点群中

宽松阈值要求

77

宽松阈值要求

- 写和读仍然需要w和r成功的响应，但是允许响应的节点不在指定的n个节点范围内

直观的例子

- 你钥匙没带，进不去自己的房间，先在邻居屋休息一下，等钥匙找到再说
- 你和朋友去就餐，朋友出去打电话，他的餐来了，你帮忙收一下，一会儿再给他

数据回传

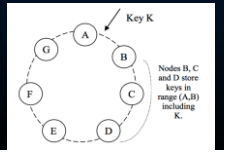
- 一旦网络中断解决，代表另一个节点临时接受的写入都需要被发送“原本的”节点

宽松阈值要求

78

一个例子(DynamoDB)

- 读、写操作在偏好列表中前n个健康节点上进行的，如果节点无法访问，列表可能变化
- 如果节点A暂时无法访问，则将其写操作发送到“代替”节点，代替节点信息预先存储在元数据中
- 代为存储的副本定期扫描以检测是否A已恢复
- 恢复后，将副本发送到A并可能从本地存储中删除



宽松阈值要求

79

宽松阈值要求的优点：**提高写入可用性**

- 只要有任何w节点可用，数据库就可以接受写入
- 然而，这意味着即使当w + r > n时，也不能确定读取某个键的最新值，因为最新的值可能已经临时写入了n之外的某些节点
- 代价是：不能保证r节点的读取正确性

无主复制 v.s. 多个数据中心 80

多数据中心

- 无主复制还适用于多数据中心操作，因为它旨在容忍冲突的并发写入，网络中断和延迟尖峰

Cassandra和Voldemort

- 在正常的无主模型中实现了多数据中心支持
- 副本的数量n包括所有数据中心的节点，在配置中，可以指定每个数据中心中想拥有的副本的数量
- 每个来自客户端的写入都会发送到所有副本
- 客户端只等待来自其本地数据中心内的节点确认
- 不会受到跨数据中心链路延迟和中断的影响

并发写处理 81

问题情景

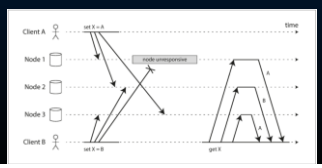
- 当多个客户端同时对某一数据(key值相同的记录)写入时，即使采用严格的阈值要求，也会发生写冲突
- 在读修复或者数据回传中也可能会产生冲突

核心问题在于：由于可变的网络延迟和部分故障，事件可能在不同的节点以不同的顺序到达

并发写处理 82

问题情景

- 下图显示了A和B同时写入三节点数据存储区中的键X
- 节点1接收A的写入，失效后，不接收 B 的写入
- 节点2首先接收 A 的写入，然后接收来自 B 的写入
- 节点 3 首先接收 B 的写入，然后接收来自 A 的写入

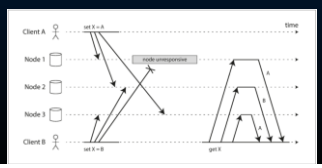


并发写处理 83

问题情景

- 如果每个节点只要接收到来自客户端的写入请求就简单地覆盖了某个键的值，那么节点就会永久不一致
  - 节点2认为 X 的最终值是 B，而其他节点认为值是 A

为了达成一致，副本应趋于相同的值。如何做到?



并发写处理—解决冲突 84

最后写入胜利(LWW)

- 为每个写入附加时间戳，挑选“最近”的最大时间戳
- 丢弃具有较早时间戳的写入
- LWW实现了最终收敛的目标
- 代价是牺牲持久性：一个Key有多个并发写入，即使它们都被报告为成功（写入 w 个副本），但只有一个写入胜利，其他写入将被丢弃，成功报告失去意义

并发写处理—解决冲突 85

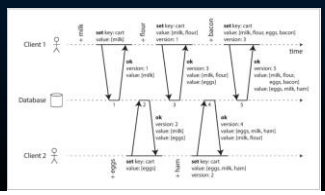
问题情景(并发关系判断)

- 我们如何判断两个操作是否是并发的?
- 事实上，我们可以简单地说：如果两个操作都不在另一个之前发生，那么两个操作是并发的（即两个操作互相之间都不知道另一个）
- 因此，只要有两个操作A和B，就有三种可能性
  - A在B之前发生
  - B在A之前发生
  - A和B并发

并发写处理—解决冲突 86

捕获“此前发生”关系算法

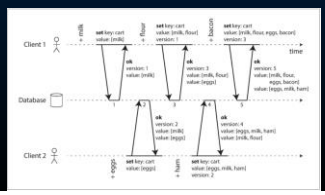
- 利用算法确定两个操作是否为并发的，还是一个在另一个之前
- 如下图所示，两个客户端同时向同一购物车添加项目。最初，购物车是空的。客户端向数据库发出五次写入



并发写处理—解决冲突 87

捕获“此前发生”关系算法

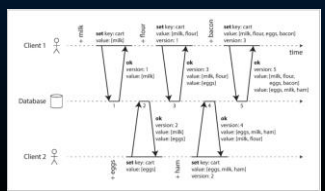
- 1. 客户端1将牛奶加入购物车；这是该键的第一次写入，服务器成功存储了它并为其分配版本号1，最后将值与版本号一起回送给客户端；



并发写处理—解决冲突 88

捕获“此前发生”关系算法

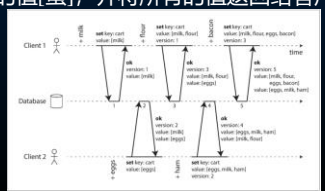
- 2. 客户端2将鸡蛋加入购物车，不知道客户端1添加了牛奶，将鸡蛋发送给服务器；服务器为此写入分配版本号2，并将鸡蛋和牛奶存储为两个单独的值，然后将这两个值都返回给客户端2，并附上版本号；



并发写处理—解决冲突 89

捕获“此前发生”关系算法

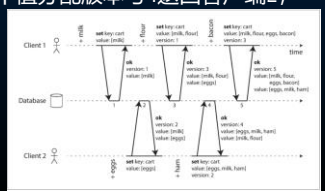
- 3. 客户端1不知道客户端2的写入，想要将面粉加入，因此认为当前购物车应该是[牛奶，面粉]，将其与版本号1发送服务器；服务器看到版本号1，检测到并发，版本3分配给[牛奶，面粉]，覆盖版本1值[牛奶]，但保留版本2的值[蛋]，并将所有的值返回给客户端1；



并发写处理—解决冲突 90

捕获“此前发生”关系算法

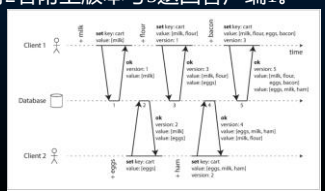
- 4. 客户端2想要加入火腿，不知道客端户1刚刚加了面粉，将收到的两个值[牛奶]和[蛋]合并，并将[鸡蛋，牛奶，火腿]以及版本号2发送服务器；服务器检测到新值会覆盖版本2[鸡蛋]，检测到版本3[牛奶，面粉]并发，将2个值分配版本号4返回客户端2；



并发写处理—解决冲突 91

捕获“此前发生”关系算法

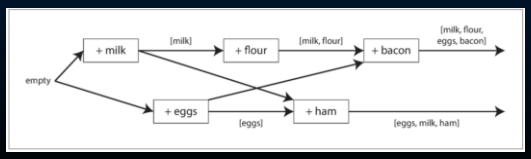
- 5. 最后，客户端1想要加培根，上次从服务器收到 [牛奶，面粉]和[鸡蛋]，它将合并新值[牛奶，面粉，鸡蛋，培根]连同版本号v3发往服务器；服务器检测到新值会覆盖v3[牛奶，面粉]，检测到与v4[鸡蛋，牛奶，火腿]并发，将2者附上版本号5返回客户端1。



并发写处理—解决冲突 92

捕获“此前发生”关系算法

- 先前操作之间的数据流如下图所示
- 在这个例子中，客户端永远不会完全掌握服务器上的数据，因为总是有另一个操作同时进行
- 但旧版本的值最终会被覆盖，并且不会丢失任何写入



并发写处理—解决冲突 93

算法的工作原理

- 服务器为每个键保留一个版本号，每次写入键时都增加版本号，并将新版本号与写入的值一起存储；
- 当客户端读取键时，服务器将返回所有未覆盖的值以及最新的版本号；
- 客户端在写入前必须读取；
- 客户端写入键时，必须包含之前读取的版本号，并且必须将之前读取的所有值合并在一起；
- 当服务器接收到具有特定版本号的写入时，它可以覆盖该版本号或更低版本的所有值，但是它必须保持所有更高版本号的值；

并发写处理—解决冲突 94

- 上一个例子只有一个副本，当有多个副本但没有领导者时，算法如何修改？

解决方案

- 为每个副本和每个主键定义版本号；
- 每个副本处理写入时，增加自己版本号，跟踪其他副本版本号，利用该信息指出覆盖哪些值，保留哪些值

版本向量：所有副本的版本号

- 读取值时，版本向量会从数据库副本发送到客户端
- 写入值时，版本向量需要发送回数据库副本
- 版本向量允许数据库区分覆盖写入和并发写入

95

👉 大数据平台的数据划分