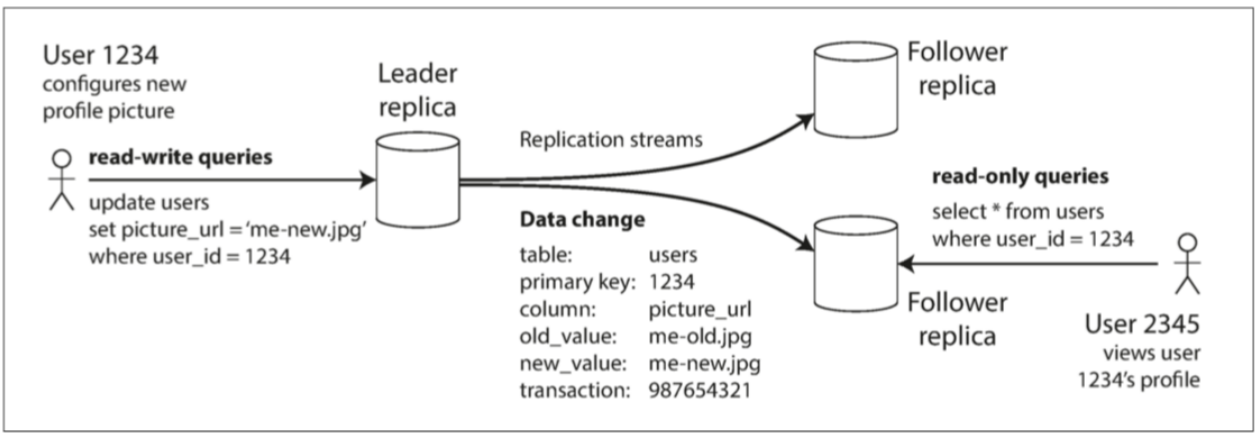
分布式系统中的复制

主从复制

- 存储数据库副本的每个节点称为 **副本（replica）。**
- 多副本的问题：如何确保数据都落在了所有的副本上。
  - 每次对数据库的写入都要传播到所有副本上，否则副本就会有不一样的数据。
  - 常见的解决方案：基于领导者的复制（主从复制）。

主从复制工作原理

- 副本之一被指定为领导者（leader，也被称作主库）
  i. 客户端写数据时，要把请求发送给领导者；
  ii. 领导者把新输入写入本地存储。
- 其他副本被称为追随者（followers，也被称作只读副本、从库、热备）
  i. 每当领导者将新数据写入本地存储时，他会把数据变更发送给所有的追随者，称之为复制日志或变更。
  ii. 每个追随者从领导者拉取日志，并相应更新其本地数据库副本，方法是按照领导者处理的相同顺序应用所有写入。
- 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。



新增从库

有时会增加一个新的从库，通常会经历如下步骤：

1. 在某个时刻获取主库的一致性快照，大多数数据库都具有这个功能，因为它是备份必需的。

2. 将快照复制到新的从库节点。
3. 从库连接到主库，拉取从快照开始到快照复制完成时所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。该位置有不同的名称：例如，PostgreSQL将其称为 **日志序列号（log sequence number, LSN）**，MySQL将其称为 **二进制日志坐标（binlog coordinates）**。
4. 当从库处理完快照之后积压的数据变更，我们说它 **赶上（caught up）** 了主库。现在它可以继续处理主库产生的数据变化了。

处理节点宕机

我们的目标：即使个别节点失效，也要能保持整个系统运行，并尽可能控制节点停机带来的影响，即保障分区容错性，即 CAP 理论中的 “P”。

从库失效：追赶恢复

- 从库可以从日志知道，在发生故障前处理的最后一个事务。
- 所以从库可以连接到主库，并拉取断开连接后的所有数据变更。
- 应用完成所有变更之后，它就赶上了主库，继续接收数据变更流。

主库失效：故障切换

- 故障切换：需要把一个从库提升为新的主库，重新配置客户端，其他从库需要开始拉取来自新主库的变更。
- 故障切换可以手动或者自动进行。

故障切换是个经典的问题，例如，可能会有如下场景发生：

- 主库接收到了新的数据，但主库只更新了少量的从库，然后主库宕机，现在只有少量节点是最新的数据，而绝大多数节点都是陈旧的数据，现在如何选取新的主库？是相信民主，选择多数节点的一方，还是相信科学，选择较新数据的一方？
- 可能主库短暂的断开连接，而此时从库认为主库宕机，选举出新的主库，这是旧的主库重新恢复连接，此时可能会出现两个主库的情况，这种情况被称为 **“脑裂”**。

这是经典的一致性与共识问题，现在较为成熟并且广被接受的解决办法是 **Raft** 算法，我们会好好学习学习它的。

复制日志的实现

基于主从库的复制，底层工作有几种不同的复制方式。

基于语句的复制

在最简单的情况下，主库记录下它执行的每个写入请求（**语句（statement）**）并将该语句日志发送给其从库，从库执行语句而“复制”主库。

问题：

- 任何调用 **非确定性函数（nondeterministic）** 的语句，可能会在每个副本上生成不同的值。比如 NOW(), RAND()。
- 如果语句使用了**自增列（auto increment）**，或者依赖于数据库中的现有数据（例如，UPDATE … WHERE <某些条件>），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。影响并发。
- 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。

传输预写式日志（WAL）

写操作通常追加到日志中：

- 对于**日志结构存储引擎**（SSTables 和 LSM 树），日志是主要存储位置。日志段在后台压缩，并进行垃圾回收。
- 覆盖单个磁盘块的 B 树，每次修改会先写入**预写式日志（Write Ahead Log, WAL）**，以便崩溃后索引可以恢复到一个一致的状态。

所以，日志都是包含所有数据库写入的仅追加字节序列。可以使用完全相同的日志在另一个节点上构建副本：主库把日志发送给从库。

PostgreSQL和Oracle等使用这种复制方法。

缺点：

- 复制与存储引擎紧密耦合。
- 不可能使主库和从库上运行不同版本的数据库软件。
- 运维时如果升级软件版本，有可能会要求停机。

逻辑日志复制（基于行）

采用逻辑日志，可以把复制与存储逻辑分离，例如 Mysql 中的 bing-log。

关系型数据库通常以行作为粒度描述数据库写入的记录序列：

- 对于插入的行，日志包含所有列的新值；
- 对于删除的行，日志包含足够的信息来唯一标识已删除的行。通常是主键，或者所有列的旧值。

- 对于更新的行，日志包含足够的信息来唯一标识更新的行，以及所有列（至少是更新列）的新值。

优点：

- 逻辑日志与存储引擎分离，方便向后兼容。可以让领导者和跟随者运行不同版本的数据库软件。
- 对于外部应用，逻辑日志也更容易解析。比如复制到数据仓库，或者自定义索引和缓存。被称为数据变更捕获。

基于触发器的复制

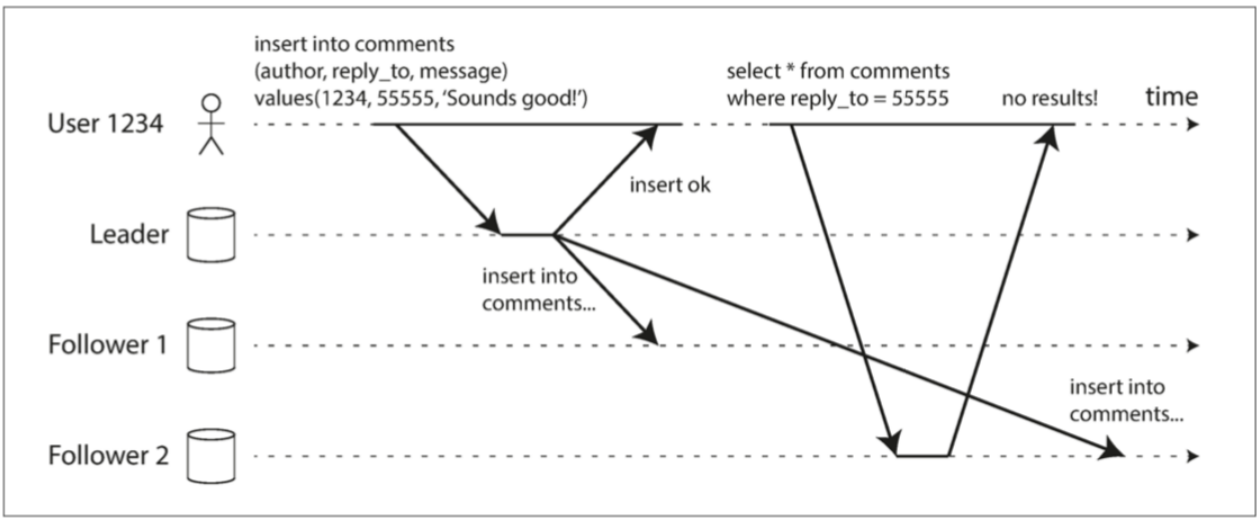
- 上述复制都是数据库自己实现的。也可以自定义复制方法：数据库提供了触发器和存储过程。
- 允许数据库变更时，自动执行应用的程序代码。
- 开销更大，更容易出错。但更灵活。

复制延迟问题

- 主从异步同步会有延迟：导致同时对主库和从库的查询，结果可能不同。
- 因为从库会赶上主库，所以上述效应被称为「最终一致性」。
- 复制延迟可能超过几秒或者几分钟，下文是 3 个经典的例子。

读己之写

如果用户把数据提交到了主库，但是主从有延迟，用户马上看数据的时候请求的从库，会感觉到数据丢失。



此时需要「读写一致性」，也成为读己之写一致性。

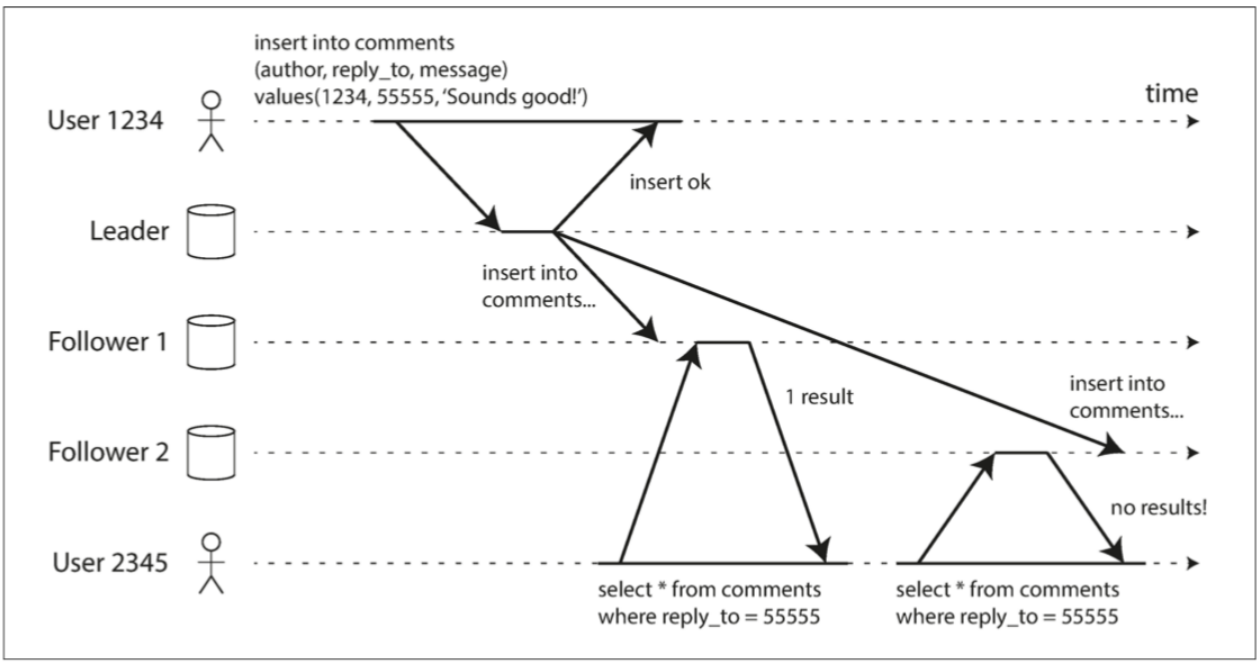
解决技术：

- 读用户可能已经修改过的内容时，都从主库读；比如读个人资料都从主库读，读别人的资料可以读从库。
- 客户端记住最近一次写入的版本号（或时间戳），客户端查询时，连同版本号（或时间戳）一起发送，从库提供查询时，必须保证对应数据的版本必须大于等于客户提供的版本；否则从另外的从库读，或者等待从库追赶上来。
- 如果副本在多个数据中心，则比较复杂。任何需要从领导者提供服务的请求，都必须路由到包含主库的数据中心。

单调读

用户可能会遇到 **时光倒流**。

第一次请求到从库看到了评论，第二次请求到另外一个从库发现评论消失。



单调读保证了这种异常不会发生。

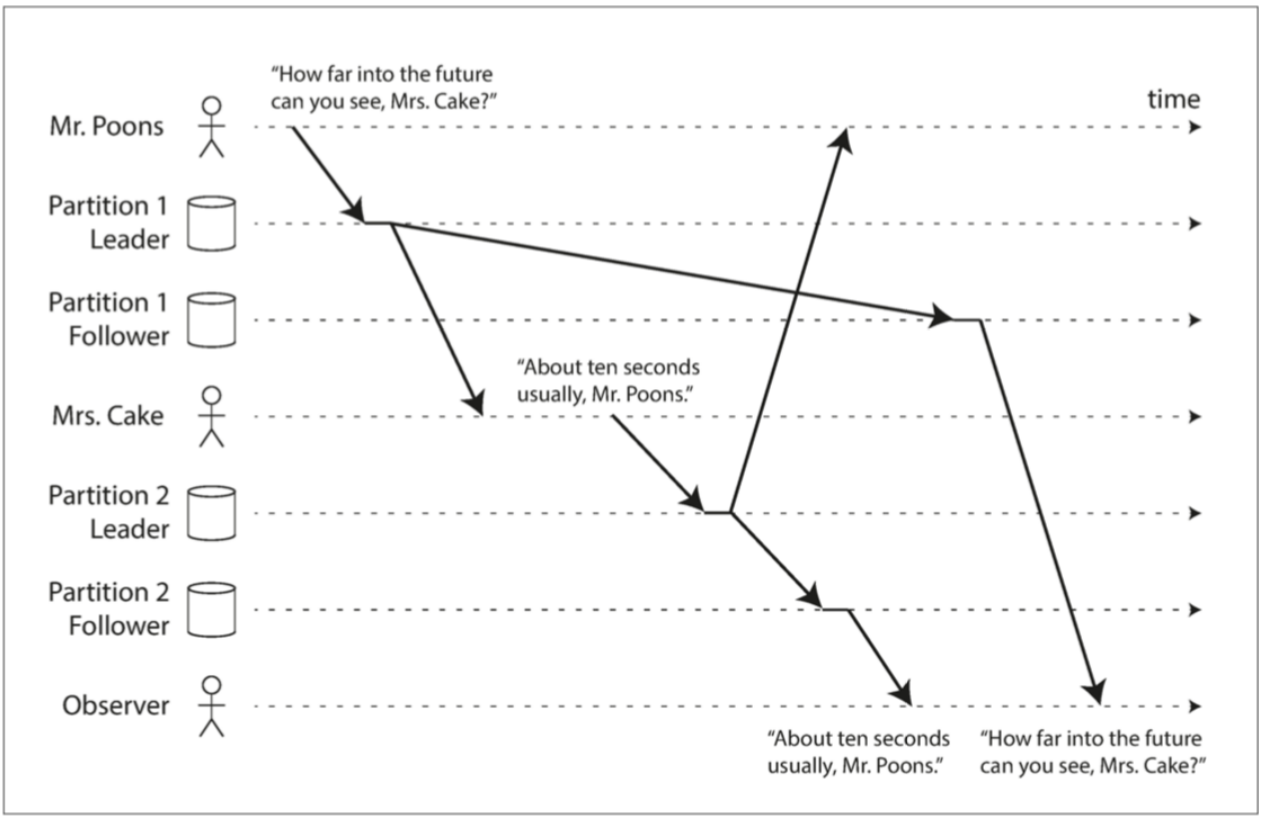
方法：

- 确保每个用户总是从同一副本来读取。比如基于用户 ID 的散列来选择副本，而不是随机选。但是如果该副本失败，则可能需要路由到另一个副本。

一致前缀读

一系列事件可能出现前后顺序不一致问题。比如回答可能在提问之前发生。

这是分区（分片）数据库或多主数据库中的一个特殊问题：不同分区之间独立，不存在全局写入顺序。



需要「一致前缀读」。

解决方法：

- 任何因果相关的写入都写入相同的分区。

复制延迟的解决方案

- 可以使用事务。**事务（transaction）** 存在的原因：**数据库通过事务提供强大的保证**，所以应用程序可以更加简单。通过使用事务，只有当主库写入了所有从库，请求才算完成，但这代价太高了，因此复制延迟本质几乎没办法解决，只能通过不同的解决方案去解决不同的场景，例如上诉三种经典问题。

> 单节点事务存在了很长时间，但是分布式数据库中，许多系统放弃了事务，因为事务的代价太高。

多主复制

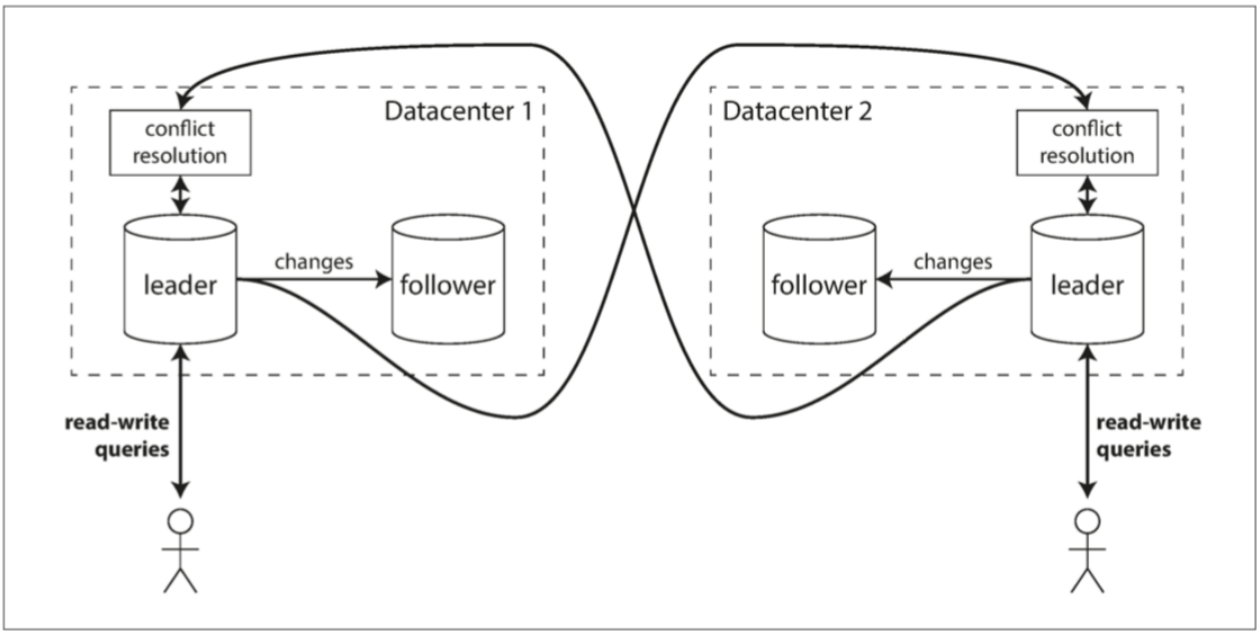
- 单个领导者的复制架构是个常见的方法，但也有其他架构。
- 基于领导者复制的主要缺点：只有一个主库，所有的写入都要通过它。
- 多个领导者的复制：允许多个节点接受写入，复制仍然是转发给所有其他节点。每个领导者也是其他领导者的追随者。

多主复制的应用场景

- 单个数据中心内部使用多个主库没有太大意义，通常多主复制用于多个数据中心中。

多个数据中心

- 多领导配置允许每个数据中心都有自己的主库。
- 每个数据中心内部使用常规的主从复制；
- 数据中心之间，每个数据中心的主库都会将其更改复制到其他数据中心的主库中。



运维多个数据中心时，单主和多主的适应情况比较：

1. **性能**
    - 单主配置中，每个写入都得穿过互联网，进入主库所在的数据中心。会增大写入时间。
    - 多主配置中，每个写操作都可以在本地数据中心进行处理，与其他数据中心异步复制。感觉到性能更好。
2. **容忍数据中心停机**
    - 单主配置中，如果主库所在的数据中心发生故障，必须让另一个数据中心的追随者成为主领导者。
    - 多主配置中，每个数据中心都可以独立于其他数据中心继续运行。若发生故障的数据中心归队，复制会自动赶上。
3. **容忍网络问题**
    - 数据中心之间的网络需要通过公共互联网，不如数据中心之内的本地网络可靠。
    - 单主配置对网络连接问题非常敏感，因为写是同步的。

    - 异步复制的多主配置更好地承受网络问题。

多主复制的缺点

- 两个数据中心可能会修改相同的内容，写冲突必须解决。
- 多主复制比较危险，应尽可能避免。

需要离线操作的客户端

多主复制的另一适用场景：应用程序在断网后仍然需要继续工作。

在这种情况下，每个设备都有一个充当领导者的本地数据库（它接受写请求），并且在所有设备上的日历副本之间同步时，存在异步的多主复制过程。复制延迟可能是几小时甚至几天，具体取决于何时可以访问互联网。

每个设备相当于一个“数据中心”

协同编辑

协作式编辑不能视为数据库复制问题，但是与离线编辑有许多相似。

一个用户编辑文档时，所做的更改将立即应用到其本地副本（web 或者客户端），并异步复制到服务器和编辑同一文档的任何其他用户。

如果想要不发生编辑冲突，则应用程序需要先将文档锁定，然后用户才能进行编辑；如果另一用户想编辑，必须等待第一个用户提交修改并释放锁定。这种协作模式相当于主从复制模型下在主节点上执行事务操作。

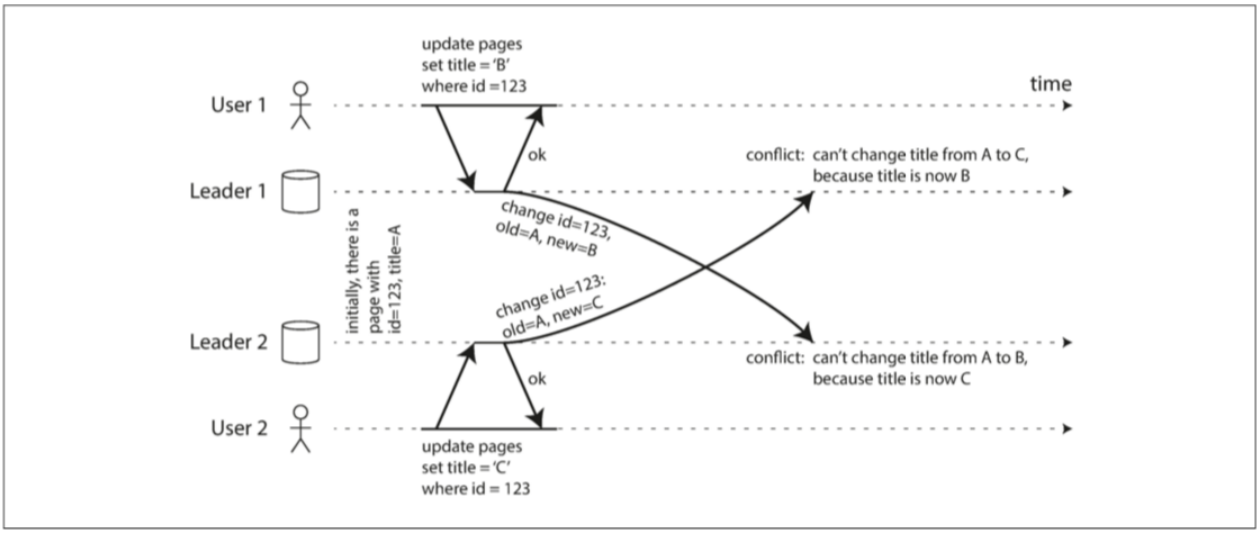
但是，为了加速写作，可编辑的粒度需要非常小（例如单个按键，甚至全程无锁）。

这会面临所有多主复制都存在的挑战，即如何解决写入冲突。

处理写入冲突

多领导者复制的最大问题是可能发生写冲突，因此需要解决冲突。

假如两个用户同时修改标题：



在这种情况中，两个标题被分别写入不同数据中心，在这一刻数据写入是成功的，但当领导者之间的复制开始时，它们就会检测到冲突，**这种冲突并不是立即被检测出来的，而是具有一定的延迟**，这是多主复制中的特殊的写入冲突。

收敛至一致的状态

在多主配置中，正如我们所讲，冲突检测是具有延迟性的，当检测到冲突时，请求已经完成，冲突无法避免。因此数据库只能以**收敛（convergent）** 的方式解决冲突，即选取一个最终的值，使得所有副本上值都相同。

实现冲突合并解决有多种途径：

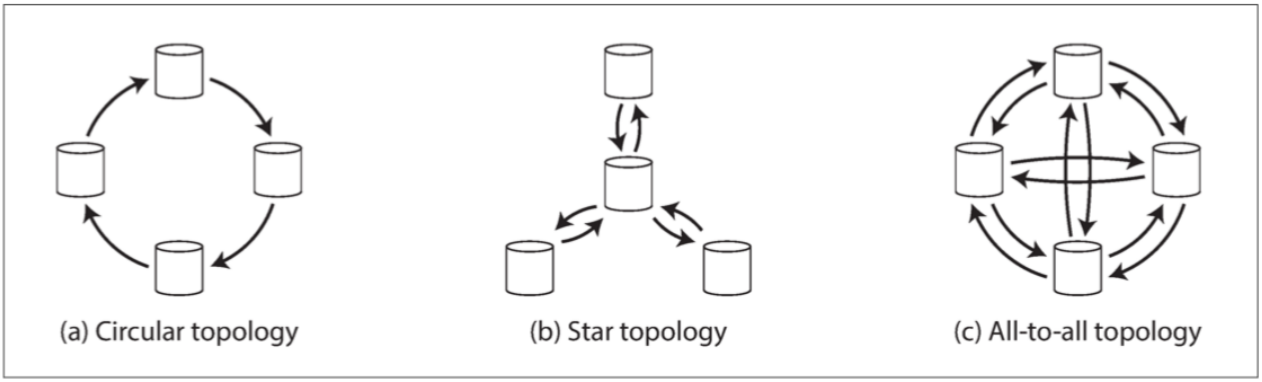
- 给每个写入一个唯一的ID（例如，一个时间戳，一个长的随机数，一个UUID或者一个键和值的哈希），挑选最高ID的写入作为胜利者，并丢弃其他写入。
- 为每个副本（数据中心、或领导者）分配一个唯一的ID，ID 编号更高的写入具有更高的优先级。
- 上述两种方式都意味着数据丢失，我们可以使用某种方式将这些值合并在一起，例如，按字母顺序排序，然后连接它们。
- 用一种可保留所有信息的显式数据结构来记录冲突，并编写解决冲突的应用程序代码（也许通过提示用户的方式）。

多主复制拓扑

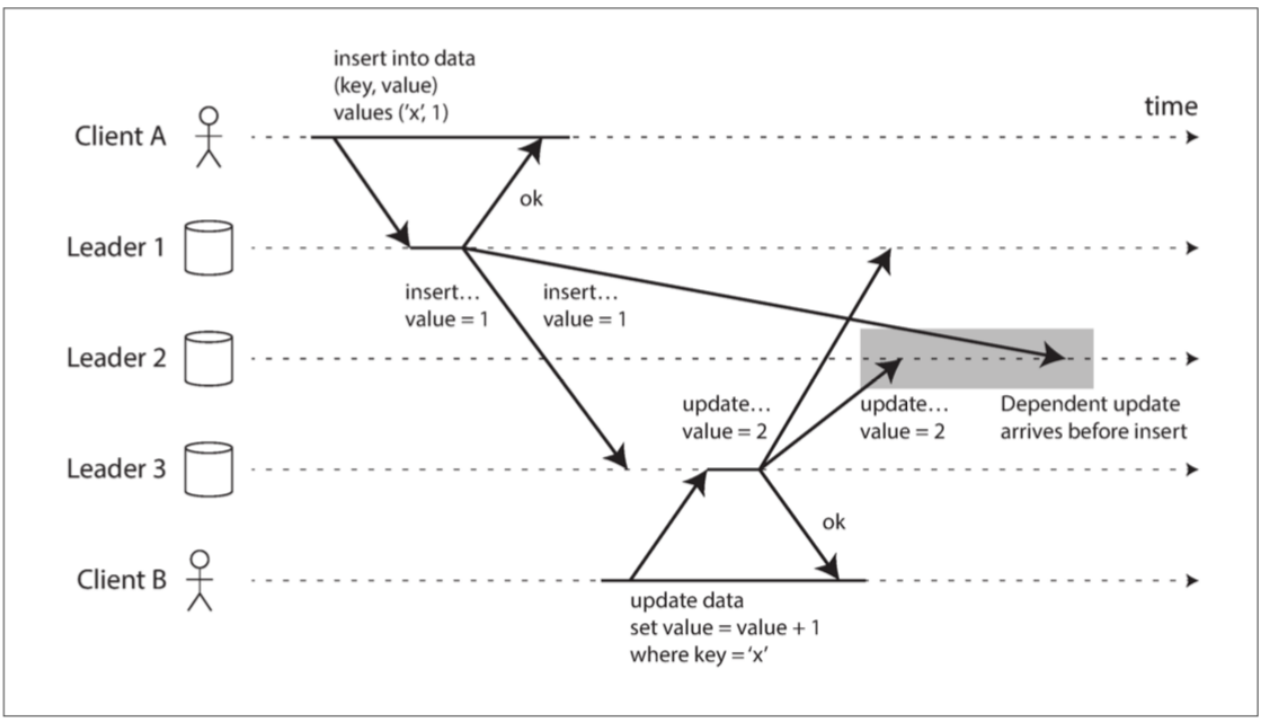
复制拓扑（replication topology）描述写入从一个节点传播到另一个节点的通信路径。

只有两个领导者时，只有一个合理的拓扑：互相写入。

当有两个以上的领导，拓扑很多样：



1. 环形拓扑。实现简单，但问题也多，例如容错性不高，会出现重复复制问题。防止无限复制循环的办法是：每个节点都有唯一的标识符，在复制日志中，每个写入都标记了所有已经过的节点的标识符。
2. 星形拓扑。实现相对简单，不会出现重复复制问题，但容错性也不高。
3. 全部到全部拓扑。实现复杂，容错性高，但消息可能会混乱。例如，网络问题导致消息顺序错乱：



在这里写入时添加时间戳是不够的的。

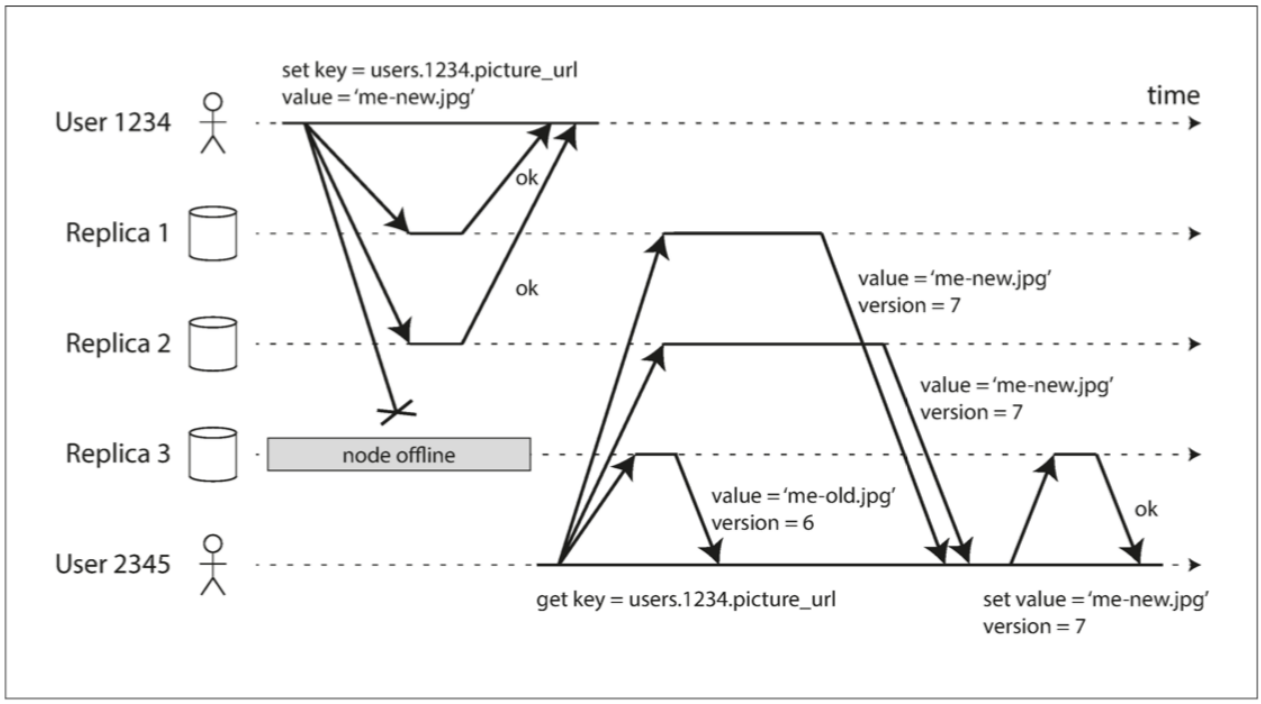
解决办法是下文将提到的**版本向量技术**。

无主复制

- 一些数据库放弃主库的概念，允许任何副本直接接收来自客户端的写入。
- 一些无主配置中，客户端直接写入到几个副本中；
- 另一些情况下，一个协调者节点代表客户端进行写入。

当节点故障时写入数据库

无主复制中，故障切换不存在。



这里存在一个问题：如果一个副本故障或下线，重启后提供的数据是落后的。

解决办法是：客户端同时请求多个副本，根据版本号确定最新值。

读修复和反熵

这里的问题是：故障节点重新上线，怎么追上错过的写入？

读修复（Read repair）

- 客户端检测到陈旧的值，客户端将新值写回到该副本。
- 适合读频繁的值。

反熵过程（Anti-entropy process）

- 数据库的后台进程，不断查找副本之间的数据差异，把缺少的数据进行复制。
- 反熵过程不会以任何特定的顺序复制写入，复制数据之前可能有显著的延迟。

读写的法定人数

在上面讲到，为使用户读取到的数据为最新版本，用户必须读取多个副本以确定最新值，问题是，读取多少个比较合适呢？同样的，客户写入数据时，写入多少个副本合适呢？

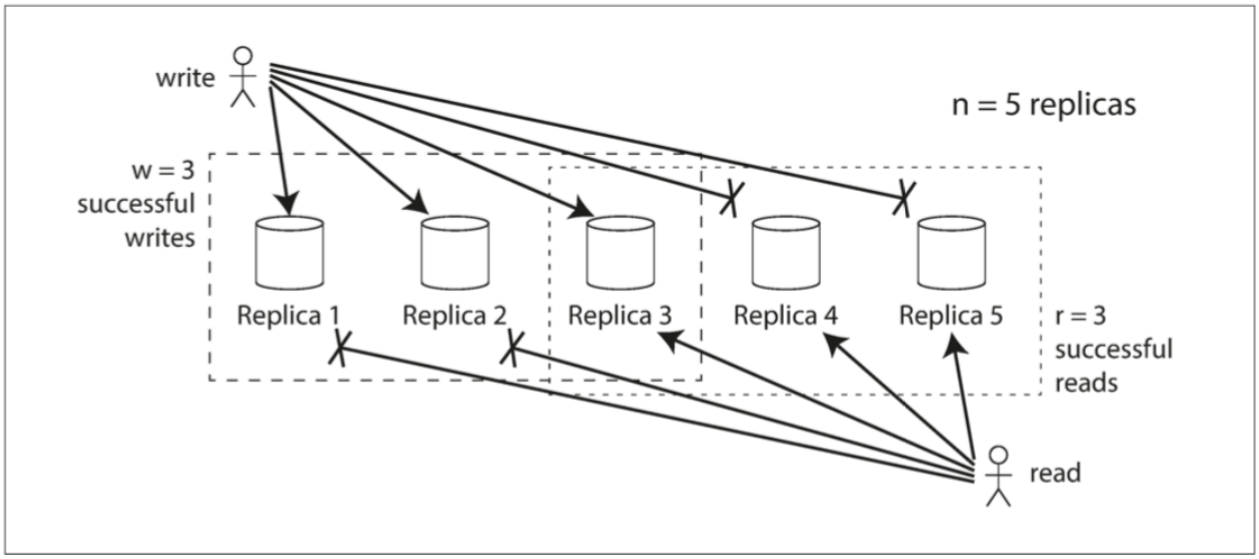
我们将读取的副本数设为 r，写入的副本数设为 w，那么如果有 n 个副本，每个写入必须由 w 节点确认才能被认为是成功的，并且我们必须至少为每个读取查询 r 个节点。

计算公式：

$$w + r > n$$

w 与 r 必须满足此公式，为验证此公式的正确性，假设客户像 w 个副本写入最新值，那么目前有 w 个副本是最新值，n - w 个副本是陈旧值。

现在客户请求数据，客户必须请求至少 r 个副本的数据，根据公式，$ r > n - w $，而所有副本中只有 n - w 个陈旧副本，也就是说用户必定能获取到最新的数据。当用户获取最新数据后，可以采取读修复更新陈旧的副本。



法定人数一致性的局限性

但是，即使在 $w + r> n$ 的情况下，也可能存在返回陈旧值的边缘情况。

- 两个写入同时发生，不清楚哪一个先发生，此时副本可能会误选陈旧值。

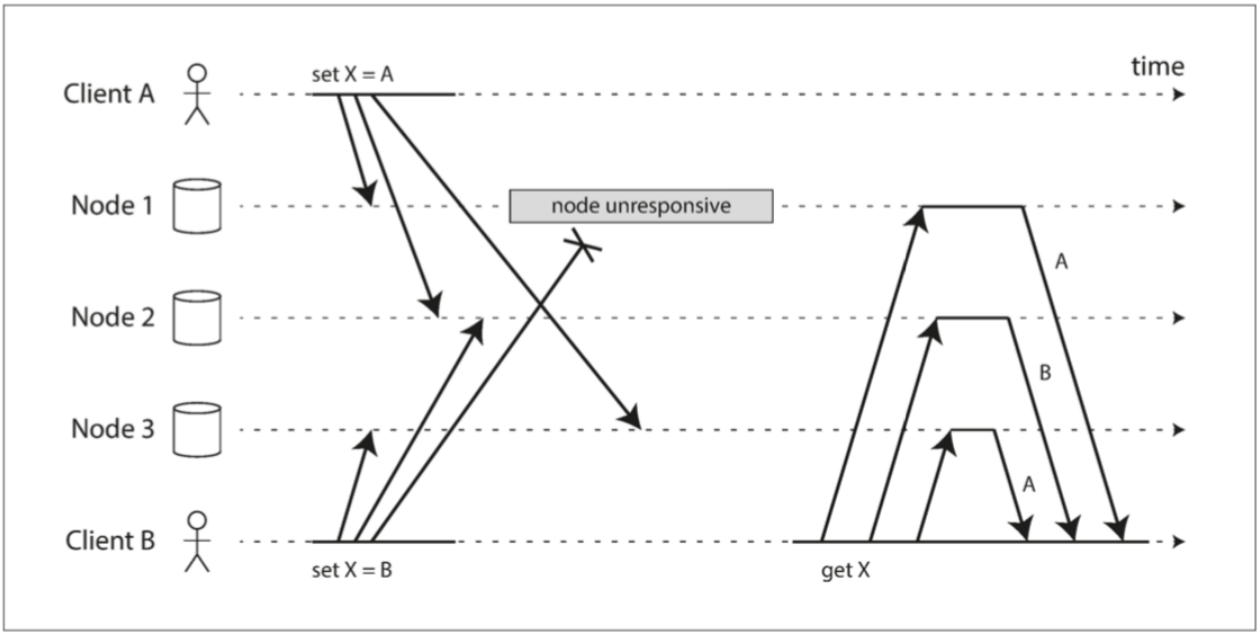
不要把 w 和 r 当做绝对的保证，应该看做是概率，强有力的保证需要事务和共识。

另外，w 与 r 的选取也必须要慎重考虑，如果 w 过大，那么客户的写入必须要等待大量副本的成功响应；r 过大，客户的读取必须同时读取大量的副本，如果此时某个副本发送故障。而如果 $w + r = n + 1$，如果任意一个副本故障都可能无法保证最终一致性。

检测并发写入

无主复制，允许多个客户端同时写入相同的 key，会发生写冲突，这与多主复制中的写冲突不同，多主复制中的写冲突检测是有延迟的，这里说的并发写入通常可以立即检测出冲突，虽然冲突的类型不同，但解决的思想却是类似的。

如果两个客户在某一时刻(不一定非要是同时发生的)对一个 key 写入不同的值，由于网络，写入副本的值是不确定的，如下图所示：



最后写入胜利（丢弃并发写入）

只需要存储最**“最近”**的值，允许**“更旧”**的值被覆盖和抛弃。

需要有一种明确的方式来确定哪个写是“最近的”，并且每个写入最终都被复制到每个副本，那么复制最终会收敛到相同的值。

例如，可以为每个写入附加一个时间戳，挑选最**“最近”**的最大时间戳，并丢弃具有较早时间戳的任何写入。

这种方式其实是我们上文所讲的最终收敛一致性。

其缺点是：以**持久性**为代价：如果同一个Key有多个并发写入，即使它们报告给客户端的都是成功（因为它们被写入 w 个副本），也只有一个写入将存活，而其他写入将被静默丢弃。

如果丢失数据不可接受，那么最后写入胜利是个很烂的选择。

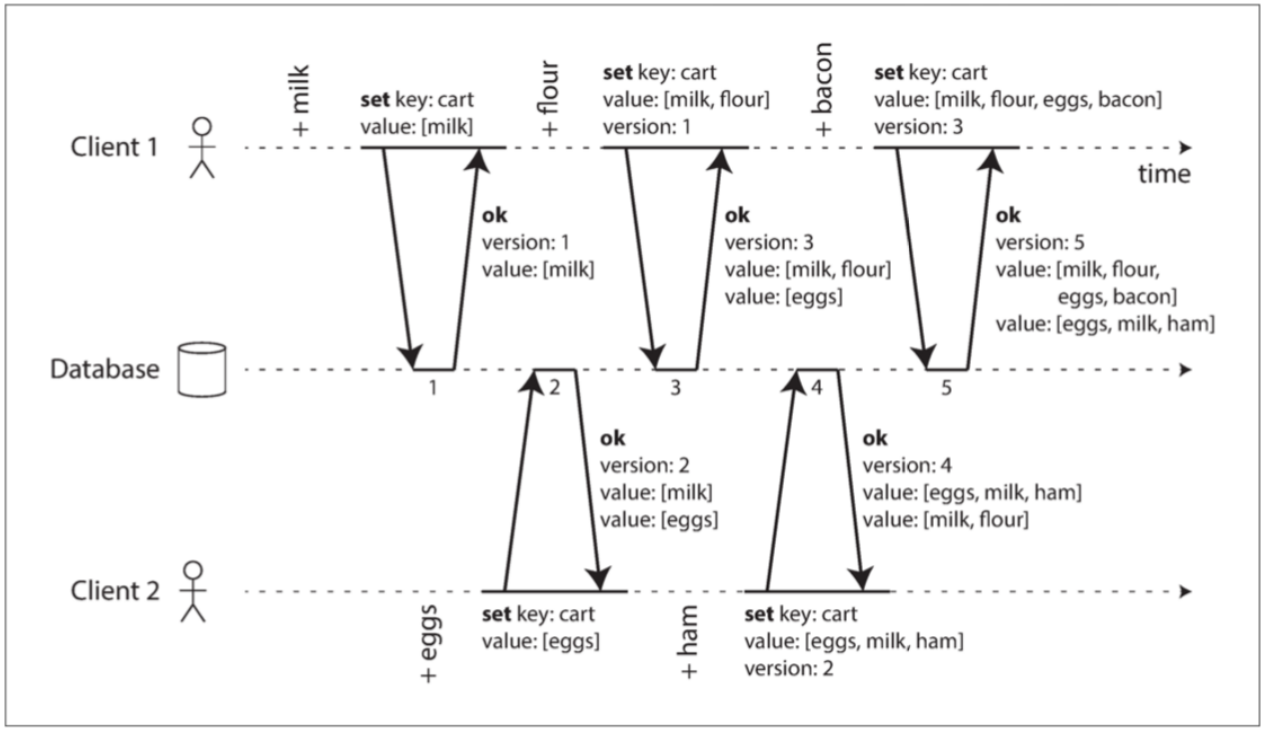
“此前发生”的关系和并发

- 只要有两个操作A和B，就有三种可能性：A在B之前发生，或者B在A之前发生，或者A和B并发。
- 我们需要的是一个算法来告诉我们两个操作是否是并发的。
- 如果一个操作发生在另一个操作之前，则后面的操作应该覆盖较早的操作，但是如果这些操作是并发的，则存在需要解决的冲突。

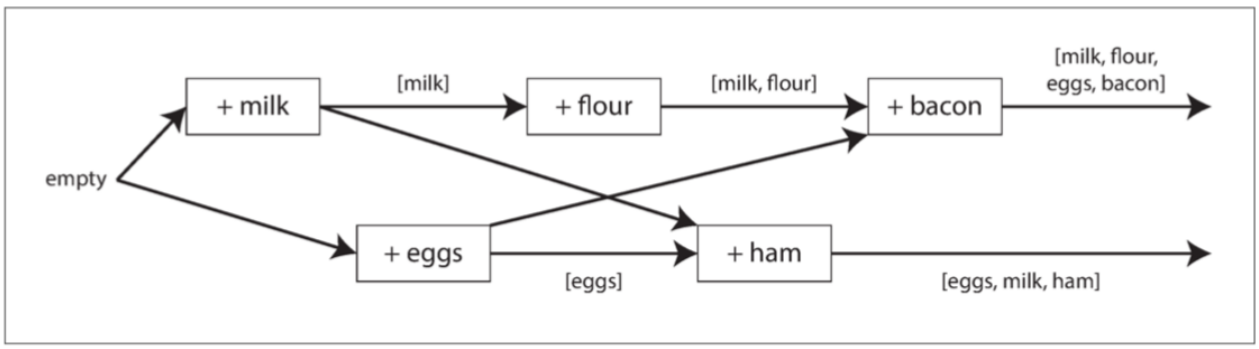
捕获"此前发生"关系

我们需要一个算法，可以确定两个操作是否是并发的，还是先后关系，

例如两个客户端同时向一个购物车添加项目，注意服务端版本号和客户端版本号：



操作依赖关系：



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，但是，旧版本的值最终会被覆盖，并且不会丢失任何写入。

服务器可以通过查看版本号来确定两个操作是否是并发的，算法的原理：

- 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（因为它知道它们已经被合并到新的值中），**但是它必须用更高的版本号来保存所有值**（因为这些值与随后的写入是并发的）。

当一个写入包含前一次读取的版本号时，它会告诉我们的写入是基于之前的哪一种状态。

理论上检测到"此前发生"关系，后发生的请求可以覆盖前面的请求，但本例中记录了同时写入的值，由于最终只有一个结果，这就需要合并同时写入的值。

合并同时写入的值

优点：没有数据被无声地丢弃

缺点：客户端需要额外工作：客户端必须通过合并并发写入的值来擦屁股。

Riak 称这些并发值为兄弟。

合并兄弟值：

- 与多领导者复制的冲突解决相同的问题。
- 最简单的是根据版本号或者时间戳最后写入胜利，但会丢失数据。
- 对于购物车来说，合理的合并方法是集合求并集。
- 但是如果从购物车中删除东西，那么求并集会出错：一个购物车删除，求并集后，会重新出现在并集终值中。
- 所以删除操作不能简单删除，需要留下有合适版本号的标记，被称为墓碑。

版本向量

多个副本，但是没有领导者，该怎么办？

- 每个副本、每个主键都定义一个版本号。
- 每个副本在处理写入时增加自己的版本号，并且跟踪从其他副本中看到的版本号。
- 这个信息指出了要覆盖哪些值，以及保留哪些值作为兄弟。

什么是版本向量

- 所有副本的版本号集合称为**版本向量（version vector）。**
- 版本向量允许数据库区分覆盖写入和并发写入。

使用版本向量可以轻易的检测出冲突，但如何收敛至一致的状态仍然是不确定的。