第 3 章

数据库基本原理

3.1 数据库的基本原理

3.1.1 关系数据库的基本原理

1. 存储介质

在关系数据库中，表示信息的数据存储在物理磁盘上，由操作系统统一管理。为了克服基于磁盘存储的高延迟，同时仍然提供持久性的存储保证，关系数据库使用了回滚日志(Undo Log)和重做日志(Redo Log)两种数据结构，如图 3-1 所示。其中，回滚日志主要用于保证原子性(允许事务回滚)，而重做日志则是保证磁盘页的不可变性。

1) 数据页

磁盘访问速度较慢，而内存中数据的访问速度远快于固态硬盘中的数据访问速度，甚至要快几个数量级。基于这个考量，基本上所有的数据库引擎都尽可能地避免访问磁盘数据，并且无论是数据库表还是数据库索引都被划分成固定大小的数据页(Data Page)。

当需要读取表或者索引中的数据时，关系数据库会将磁盘中的数据页映射入存储缓冲区。当需要修改数据时，关系数据库首先会修改内存页中的数据，然后利用 fsync()这样的同步工具将改变同步回磁盘中。

存储基于磁盘的数据页的缓冲池大小有限，因此通常需要存储数据工作集。只有当整个数据都可以放入内存时，缓冲池才能存储整个数据集。如果需要缓存新数据页，且磁盘上的总体数据大于缓冲池大小，则缓冲池将不得不逐出旧的数据页，为新的数据页腾出空间。

2) 回滚日志

由于可能同时有多个事务并发地对内存中的数据进行修改，关系数据库往往需要依赖

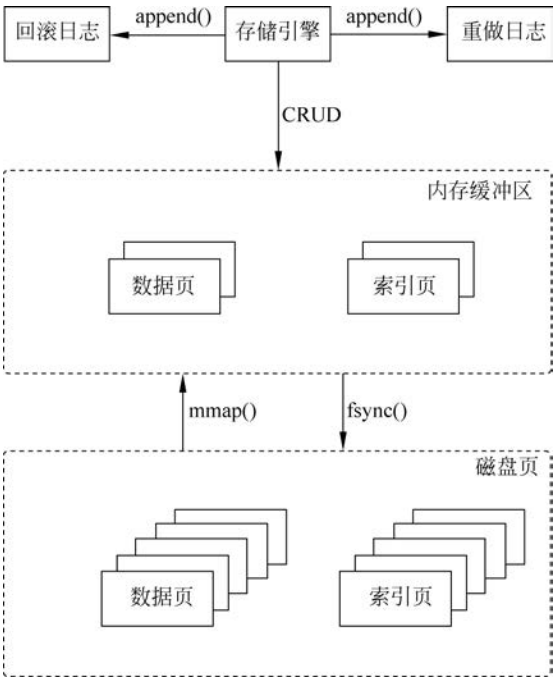

图 3-1　关系数据库的工作原理

于某个并发控制机制(例如,两段锁协议),来保证数据一致性。因此,当某个事务需要更改数据表中某一行时,未提交的更改会被写入内存数据中,而之前的数据会被追加写入回滚日志文件中临时保存。

Oracle 或者 MySQL 中使用回滚日志数据结构,而 SQL Server 中则是使用事务日志(Transaction Log)完成此项工作。PostgreSQL 并没有回滚日志,不过其内建支持多版本的表数据,即同一行的数据可能同时存在多个版本。总而言之,关系数据库都采用类似的数据结构以支持事务回滚,这是原子性的强制性要求。

如果当前运行的事务发生了回滚,回滚日志会被用于重建事务起始阶段时的内存页。

3) 重做日志

一旦某个事务提交,内存中的改变就需要同步到磁盘中。不过,并不是所有的事务提交都会立刻触发同步。事实上,过高频次的同步反而会对应用性能造成损伤。根据 ACID 原则,提交之后的事务必须要保证持久性,这意味着即使此时数据库引擎宕机了,提交之后的更改也应该被持久化存储下来。

这里,关系数据库依靠重做日志来达成这一目的,提供持久性的同时却不在每次事务提交时都触发同步。重做日志是一个仅允许追加写入的基于磁盘的数据结构,它会记录所有尚未执行同步的事务操作。相较于一次性写入固定数目的数据页到磁盘中,顺序地写入重做日志会比随机访问快很多。因此,关于事务 ACID 特性的保证与应用性能之间也就达成了较好的平衡。

该数据结构在 Oracle 与 MySQL 中被称为重做日志,而在 SQL Server 中仍是由事务日志(Transaction Log)执行,在 PostgreSQL 中则将其称为预写日志(Write-Ahead Log,WAL)。

回到上面的问题,应该在何时将内存中的数据写入磁盘中。关系数据库系统往往使用

检查点来同步内存的“脏”数据页与磁盘中的对应部分。为了避免I/O阻塞,同步过程往往需要在较长的时间段内分块完成。因此,关系数据库需要保证,即使在所有内存脏数据页同步到磁盘之前引擎就崩溃,也不会发生数据丢失。同样地,在每次数据库重启时,数据库引擎会基于重做日志重构自上次成功的检查点以来所有的内存数据页。

2. 数据库三级封锁协议

封锁是实现并发控制的一个非常重要的技术。所谓封锁就是事务T在对某个数据对象,如表、记录等操作之前,先向系统发出请求,对其加锁。加锁后事务T就对该数据对象有了一定的控制,在事务T释放它的锁之前,其他事务不能更新此数据对象。基本的封锁类型有两种,分为排他锁(Exclusive Lock,X锁)和共享锁(Share Lock,S锁)。

排他锁又称为写锁。若事务T对数据对象A加上排他锁,则只允许T读取和修改A,其他任何事务都不能再对A加任何类型的锁,直到T释放A上的锁。这就保证了其他事务在T释放A上的锁之前不能再读取和修改A。

共享锁又称为读锁。若事务T对数据对象A加上共享锁,则其他事务只能再对A加共享锁,而不能加排他锁,直到T释放A上的共享锁。这就保证了其他事务可以读A,但在T释放A上的锁之前不能对A做任何修改。

在运用排他锁和共享锁这两种基本封锁对数据对象加锁时,还需要约定一些规则,例如,应何时申请排他锁或共享锁、持续时间、何时释放等,这些规则被称为封锁协议(Locking Protocol)。对封锁方式规定不同的规则,就形成了各种不同的封锁协议。下面介绍三级封锁协议。三级封锁协议分别在不同程度上解决了更改丢失、不可重复读和读“脏”数据3种不一致问题,为并发操作的正确调度提供一定的保证。

更改丢失指的是当有多个事务对数据进行操作时,有些事务的操作被其他事务所替代。

不可重复读指的是在某一个事务执行期间其他的事务修改、插入或删除了数据,则该事务在验证数据时会出现与开始结果不一致的情况。

读“脏”数据指的是某一个事务读取了另一个事务执行期间的数据内容,当另一个事务回滚时,该事务读取到的内容就是无效数据。

一级封锁协议:事务T在修改数据R之前必须先对其加排他锁,直到事务结束才释放。事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK)。一级封锁协议可防止更改丢失,并保证事务T是可恢复的。在一级封锁协议中,如果仅仅是读数据而不对其进行修改,是不需要加锁的,所以它不能保证可重复读和不读“脏”数据。

二级封锁协议:在一级封锁协议的基础之上,进一步要求事务T在读取数据R之前必须先对其加共享锁,读完后即可释放。二级封锁协议除了防止更改丢失,还可进一步防止读“脏”数据。

三级封锁协议:在一级封锁协议的基础之上,进一步要求事务T在读取数据R之前必须先对其加共享锁,直到事务结束才释放。三级封锁协议除了防止更改丢失和不读“脏”数据外,还进一步防止了不可重复读。

执行了封锁协议后,就可以克服数据库操作中的数据不一致引起的问题。

3. 数据库架构

随着业务规模增大,数据库存储的数据量和承载的业务压力也不断增加,数据库的架

构需要随之变化，为上层服务。

如图 3-2 所示，数据库架构按照主机数量分为单机架构和多机架构，这里的主机指的是数据库服务器。单机架构就是指单个数据库主机。在单机架构中，又分为单主机和独立主机。所谓单主机指的是应用和数据库都放在一个主机上，这对主机的性能负担过重。因此有了独立主机，将应用和数据库分开，数据库放在专门的数据库服务器上，应用则放在专门的应用服务器上。

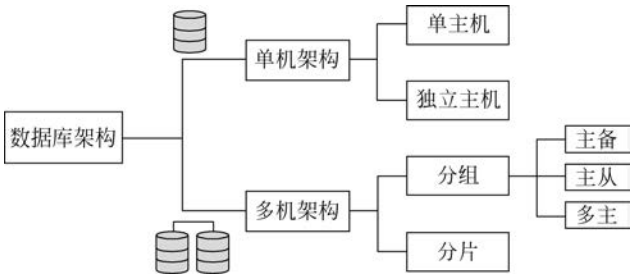

图 3-2　关系数据库架构

多机架构通过增加服务器数量来提升整个数据库服务的可用性和服务能力，包括分组和分片。其中，分组指的是对每个服务器区分角色，这里分为主备、主从和多主结构。不管是哪种结构，本质上都是在多个数据库结构相同、存储数据相同的服务器之间进行数据同步。而分片结构则是将数据库按照算法分摊在各节点上，每节点维护自己的数据库数据。

1）单机架构

关系数据库单机架构如图 3-3 所示。为了避免应用服务和数据库服务对资源的竞争，单机架构也从早期的单主机模式发展到数据库独立主机模式，将应用和数据服务分开。应用服务可以增加服务器数量，进行负载均衡，增大系统并发能力。早期互联网 LAMP（Linux、Apache、MySQL、PHP）架构的服务器就是最典型的单机架构的使用。

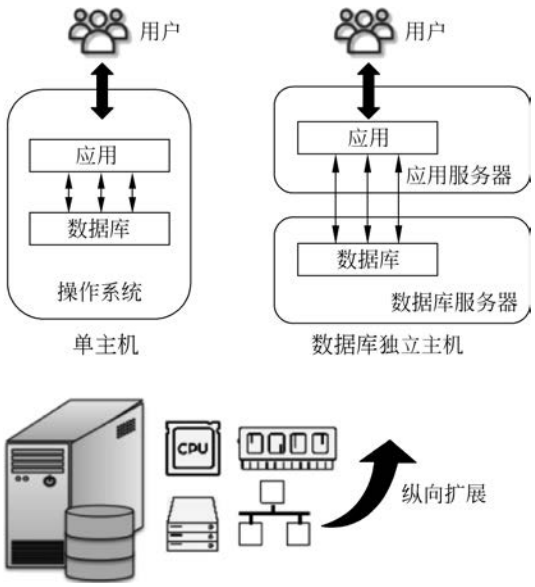

图 3-3　关系数据库单机架构

(1) 优点。

部署集中,运维方便。一个数据库即一个主机,对单机架构服务器进行维护很方便。

(2) 缺点。

① 可扩展性差。数据库单机架构只有纵向扩展(Scale-up),通过增加硬件配置来提升性能。但单台主机的硬件可配置的资源会遇到上限。

② 存在单点故障。扩容时往往需要停机扩容,服务停止。硬件故障导致整个服务不可用,甚至数据丢失。

③ 性能瓶颈。性能压力集中在单机上。

2) 分组架构——主备

关系数据库"主备"分组架构如图 3-4 所示。数据库部署在两台服务器上,其中承担数据读写服务的服务器称为"主机",另外一台服务器利用数据同步机制把主机的数据复制过来,称为"备机"。备机在正常情况下不会被使用,只会进行主备之间数据的同步。当主机出现故障后,备机将代替主机的作用进行数据读写。同一时刻,只有一台服务器对外提供数据服务。

(1) 优点。

① 应用不需要针对数据库故障来增加开发量。

② 相对单机架构解决了单点故障的情况,提升了数据容错性。

(2) 缺点。

① 资源浪费。备机和主机同等配置,但长期范围内基本上资源闲置,无法利用。

② 性能瓶颈。性能压力依旧集中在单机上,无法解决性能瓶颈问题。

③ 当出现故障时,主、备机切换需要一定的人工干预或者监控。

3) 分组架构——主从

关系数据库"主从"分组架构如图 3-5 所示。主从结构的部署模式和主备机模式相似,"备机"上升为"从机",对外提供一定的数据服务。在主从结构中,主机依旧只能有一台,但是从机可以有多台,根据业务需求进行横向扩展。主从结构通过读写分离的方式分散压力,应用在主机(写库)上进行写入、修改和删除操作,并把查询请求分配到从机(读库)上。

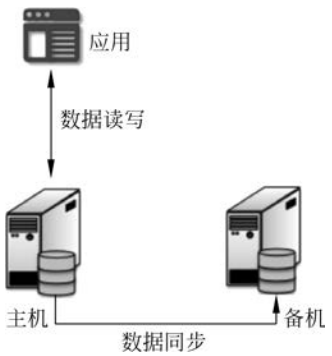

图 3-4 关系数据库"主备"分组架构

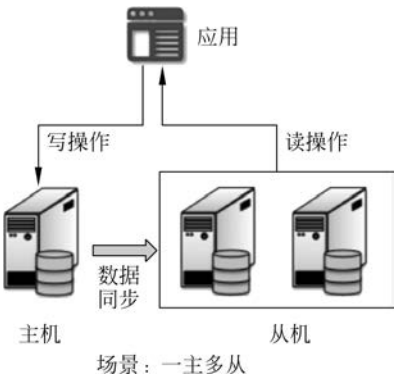

图 3-5 关系数据库"主从"分组架构

(1) 优点。

① 提升资源利用率,适合读多写少的应用场景。

② 在高并发读的使用场景,可以使用负载均衡在多个从机间进行平衡。

③ 从机的扩展性比较灵活，扩容操作不会影响到业务进行。

(2) 缺点。

① 延迟问题。数据同步到从机数据库时会有延迟，所以应用必须能够容忍短暂的不一致性，对于一致性要求非常高的场景是不适合的。

② 写操作的性能压力还是集中在主机上。

③ 主机出现故障时，需要实现主从切换，人工干预需要响应时间，自动切换复杂度较高。

4) 分组架构——多主

关系数据库“多主”分组架构如图 3-6 所示。多主架构也称为双活、多活架构。在这种架构下，数据库服务器不再区分主从设备，都互为主从，同时对外提供完整的数据服务。

(1) 优点。

资源利用率较高的同时降低了单点故障的风险。

(2) 缺点。

① 双主机都接受写数据，要实现数据双向同步。双向复制同样会带来延迟问题，极端情况下有可能导致数据丢失。所以对于一致性要求非常高的场景，不适合使用这种架构。

② 数据库数量增加会导致数据同步问题变得极为复杂，实际应用中多见双机模式。

在多主架构中还有一种特殊的结构，称为共享存储多活架构(Shared-Disk)，如图 3-7 所示。它解决了主从设备之间数据同步带来的数据一致性问题，数据库服务器共享数据存储，而多个服务器实现均衡负载。但其实现技术难度大，当存储器接口带宽达到饱和时，增加节点并不能获得更高的性能，存储 I/O 容易成为整个系统的性能瓶颈。共享存储多活架构较为成功的案例有 Oracle 的 RAC(Real Application Cluster)和微软的 SQL Server Failover Cluster。

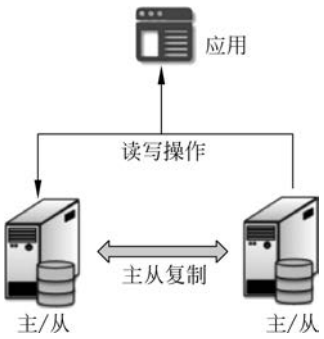

图 3-6　关系数据库“多主”分组架构

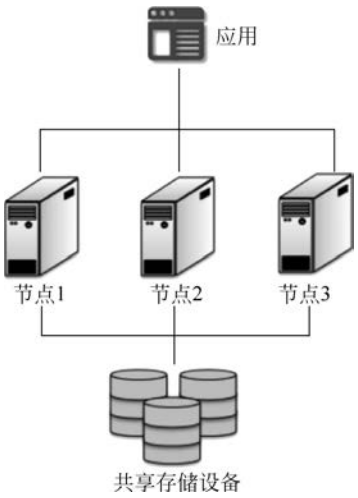

图 3-7　关系数据库共享存储多活架构

5) 分片架构

关系数据库分片(Sharding)架构如图 3-8 所示。分片架构主要表现形式就是水平数据分片架构，把数据分散在多节点上，每个分片包括数据库的一部分，称为一个 shard。多节点都拥有相同的数据库结构，但不同分片的数据之间没有交集，所有分区数据的并集构成数据总体。常见的分片方法有根据列表值、范围取值或哈希值进行数据分片。

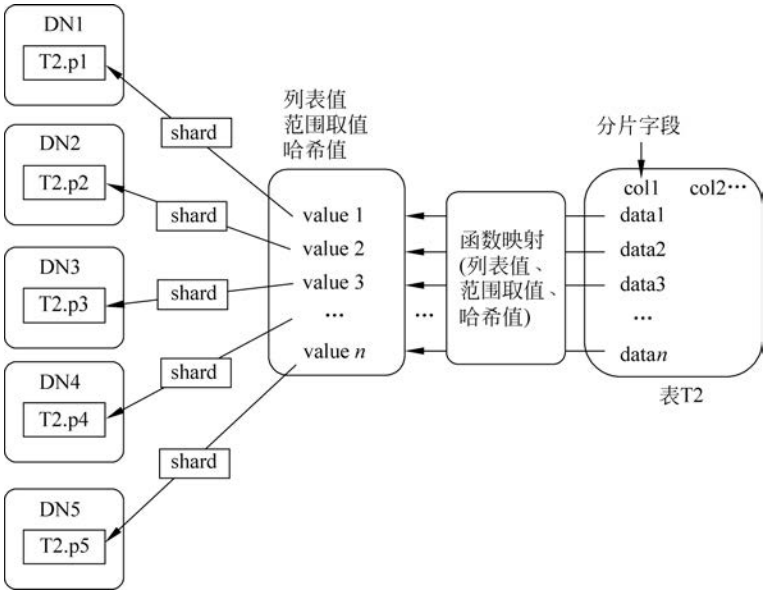

图 3-8　关系数据库分片架构

分片架构的优点是数据分散在集群内的各节点上，所有节点可以独立工作。

在分片架构的基础上，还有两种具有代表性的数据库架构，分别是无共享(Shared-Nothing)架构和 MPP(Massively Parallel Processing，大规模并行处理)架构。无共享架构集群中每个节点(处理单元)都完全拥有自己独立的 CPU/内存/存储，不存在共享资源。各节点(处理单元)处理自己本地的数据，处理结果可以向上层汇总或者通过通信协议在节点间流转。节点是相互独立的，扩展能力强。整个集群拥有强大的并行处理能力。MPP 架构是将任务并行地分散到多个服务器和节点上，在每节点上计算完成后，将各自部分的结果汇总在一起得到最终的结果。其特点是任务并行执行，分布式计算。

各种数据库架构的特点对比如表 3-1 所示。

表 3-1　数据库架构特点对比

特　点	单　机	主　备	主　从	多　主	分　片
高可用性	差	一般	较好	好	好
读写性能	依赖于单主机的硬件性能瓶颈	依赖于单主机的硬件性能瓶颈	利用读写分离，写性能受主机限制，读性能可以通过增加从机数量来提升并发能力	多个主机能够同时提供读写服务，具备较好的读写能力	Shared-Nothing 架构提供了出色的分布式计算能力，具备强大的并行处理能力
数据一致性	不存在数据不一致问题	利用数据同步机制在主备机之间进行同步，存在数据延迟问题和数据丢失风险	同主备模式，而且随着从机数量的增加，数据延迟问题和数据丢失风险更为突出	多主机之间需要进行数据双向同步，所以容易产生数据不一致问题。但对于 Shared-Disk 架构不存在数据不一致问题	基于分片技术，数据分散在各节点上，节点之间不需要数据同步，所以不存在数据一致性问题

续表

特　点	单　机	主　备	主　从	多　主	分　片
可扩展性	只能纵向扩展，会遇到单机硬件性能瓶颈	只能纵向扩展，同样会遇到单机硬件性能瓶颈	从机可以通过横向扩展来提升并发读能力	扩展性好，但是主机数量增加，会导致数据同步的复杂性急剧升高	理论上可以实现线性扩展，扩展性最好

3.1.2　NoSQL 与 NewSQL 的基本原理

1. 存储引擎

在数据库中，存储引擎属于底层数据结构，直接决定了数据库所能够提供的性能和功能。常见的存储引擎有哈希存储引擎、B－树存储引擎、B+树存储引擎、LSMT 存储引擎等。

1）哈希存储引擎

代表数据库有 Redis、Memcached 等。

哈希存储引擎是哈希表的持久化实现，一般用于键值类型的存储系统。哈希存储的基本思想是以 Key(键)为自变量，通过一定的函数关系(哈希函数)，计算出对应函数值(哈希地址)，以这个值作为数据元素的地址，并将数据元素存入相应地址的存储单元中。查找时再根据要查找的关键字采用同样的函数计算出哈希地址，然后直接到相应的存储单元中去取要找的数据元素。

采用哈希结构的存储数据，其检索效率非常高，索引的检索可以一次定位，I/O 次数可以达到很小，不像 B－树索引需要从根节点到枝节点，最后才能访问到页节点这样多次的 I/O 访问，所以哈希索引的查询效率要远高于 B－树索引。

虽然哈希索引效率高，但是哈希索引本身由于其特殊性也带来了很多限制和弊端，如哈希索引不能使用范围查询；存在相同哈希值(哈希碰撞现象)。

2）B－树存储引擎

代表数据库为 MongoDB。

B－树存储引擎是 B－树的持久化实现，相比较哈希存储引擎，B－树存储引擎不仅支持随机读取，还支持范围扫描。

B－树为一种多路平衡搜索树，与红黑树最大的不同在于，B－树的节点可以有多个子节点，从几个到几千个。B－树与红黑树相似，一棵含 n 个节点的 B－树的高度也为 $O(\log n)$，但可能比一棵红黑树的高度小许多，因为它的分支因子比较大。所以 B－树可在 $O(\log n)$时间内，实现各种如插入、删除等动态集合操作。

B－树的搜索从根节点开始，对节点内的关键字(有序)序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子节点；重复操作，直到所对应的儿子指针为空，或已经是叶节点。

3）B+树存储引擎

B+树是关系数据库常用的索引存储模型，能够支持高效的范围扫描叶节点相关连接并且按主键有序，扫描时避免了耗时的遍历树操作。B+树的局限在于不适合大量随机写场景，会出现“写放大”和“存储碎片”。

写放大：为了维护 B+树结构产生的额外 I/O 操作，如逻辑上仅需要一条写入记录，实

际需要多个页表中的多条索引记录。

存储不连续：新增叶节点会加入原有叶节点构成的有序链表中，整体在逻辑上是连续的。但磁盘存储上，新增页表申请的存储空间与原有页表很可能是不相邻的。这样，在后续包含新增叶节点的查询中，将会出现多段连续读取，磁盘寻址的时间将会增加。进一步而言，在 B+树上进行大量随机写会造成存储的碎片化。

在实际应用 B+树的数据库产品，如 MySQL，通常提供了填充因子(Factor Fill)进行针对性的优化。填充因子设置过小会造成页表数量膨胀，增大对磁盘的扫描范围，降低查询性能；设置过大则会在数据插入时出现写扩大，产生大量的分页，降低插入性能，同时由于数据存储不连续，降低了查询性能。

4) LSMT 存储引擎

代表数据库有 HBase、Cassandra 等。

主流的 NoSQL 数据库则使用 LSMT(Log-Structured Merge Tree，日志结构合并树)来组织数据。LSMT 存储引擎和 B－树一样，支持增、删、改、随机读取以及顺序扫描。通过批量转储技术规避磁盘随机写入问题，极大地改善了磁盘的 I/O 性能，被广泛应用于后台存储系统。

LSMT 的思想非常朴素，就是将对数据的修改增量保持在内存中，达到指定大小限制后将这些修改操作批量写入磁盘，读取时需要合并磁盘中的历史数据和内存中最近的修改操作。

LSMT 是一种基于硬盘的数据结构，与 B－树相比，能显著地减少硬盘磁盘臂的开销，并能在较长的时间内提供对文件的高速插入(删除)。然而 LSMT 在某些情况下，特别是在查询需要快速响应时性能不佳。通常 LSMT 适用于索引插入比检索更频繁的应用系统。

在实际应用中，为防止内存因断电等原因丢失数据，写入内存的数据同时会顺序地在磁盘上写日志，类似于 WAL，这也是 LSMT 这个词中 Log 一词的来历。

2. 一致性协议

1) 2PC(Tow-Phase Commit，两阶段提交)协议

在两阶段提交中，主要涉及两个角色，分别是协调者和参与者。

第一阶段：当要执行一个分布式事务时，事务发起者首先向协调者发起事务请求，然后协调者会给所有参与者发准备请求(其中包括事务内容)，参与者会开启事务，进行预提交，并返回是否可以提交。

第二阶段：如果这个时候所有参与者都返回可以提交的消息，协调者便向参与者发送提交请求，当参与者收到请求并提交完毕后会给协调者发送提交成功的响应。

两阶段提交可以保证数据的强一致性，但也存在诸多问题，如单点故障，协调者宕机则整个服务变成不可用；同步阻塞协议，参与者对事务处理后不提交，如果协调者不可用，那么资源就无法被释放。

2) 3PC(Three-Phase Commit，三阶段提交)协议

3PC 可以看作是 2PC 的改进，主要是弥补 2PC 存在的缺点。3PC 比 2PC 多了一个询问阶段，也就是询问准备、预提交、提交 3 个阶段。在询问准备阶段，所有参与者接收 canCommit 命令，并根据自身情况向协调者返回 yes 或 no，这里所有参与者没有事务锁定，如果参与者返回了 no 或者超时没响应，则协调者发送事务中断请求。

相比较 2PC，3PC 最大的优点是降低参与者的阻塞范围（2PC 第一阶段是阻塞的），其次能够在单点故障后继续达成一致。但是参与者收到了 preCommit 请求后出现网络分区故障，等待超时后，协调者给网络正常的参与者进行事务回滚，而网络故障的参与者会继续提交事务，导致数据不一致。

3）Paxos 算法

Paxos 算法是一种基于消息传递且具有高度容错特性的一致性算法，是目前公认的解决分布式一致性问题最有效的算法之一。

在该算法中有 3 种角色，用提议者（Proposer）、决策者（Acceptor）、决策学习者（Learner）来表示。

- 提议者：提案的提出者，可以提出多个提案。而所说的提案，可以指任何想在该分布式系统中想要执行的操作，在整个 Paxos 执行周期中，只会有一个提案被批准。在一个提案中，不仅包含提案的值，还有全局唯一编号，且必须全局递增，如可以考虑使用时间戳＋serverID 来实现。
- 决策者：提案的决策者，会回应提议者的提案，来接收和处理 Paxos 的两个阶段的请求。
- 决策学习者：不会参与提案的决策，会从提议者/决策者学习最新的提案来达成一致。它的存在可用于扩展读性能，或者跨区域之间读操作。

Paxos 算法流程大致分为两个阶段。

阶段一：提议者选择一个提案编号（假设为 n），向半数以上决策者广播 prepare（n）请求。对于集合中的决策者需做出如下回应。

- 若 n 大于该决策者已经响应过提案的最大编号，那么它作为响应返回给提议者，并且将不再被允许接受任何小于 n 的提案。
- 如果该决策者已经批准过提案，那么就向提议者反馈当前已经批准过提案编号中最大但小于 n 的那个值。

阶段二：如果提议者收到来自半数以上的决策者对于其发出的编号为 n 的准备请求响应，那么它就会发送一个针对[n，Vn]提案的接收请求给决策者。

注意：*Vn 就是收到的响应中编号最大的提案的值，如果响应中不包含任何提案，那么 Vn 就由提议者决定。*

如果决策者收到一个针对编号为 n 的提案的接收请求，只要该决策者没有对编号大于 n 的准备请求做出过响应，它就接受该提案。

决策学习者如何获取提案？

- 当决策者批准了一个提案时，将该提案发送给其所有的决策学习者，该方案非常简单直接。
- 选定一个主决策学习者，在决策者批准了一个提案时，就将该提案发送给主决策学习者，主决策学习者再转发给其他决策学习者。虽然该方案较方案一通信次数大大减少，但很可能会出现单点故障问题。
- 将主决策学习者的范围扩大，即决策者可以将批准的提案发送给一个特定的决策学习者集合，然后集合中的决策学习者再转发通知给其他决策学习者。该方案可以提高可靠性，但也会增加其网络通信的复杂度。

4) Raft 算法

Raft 算法是一种为了管理复制日志的一致性算法。分布式系统中节点分为领导者(Leader)、候选者(Candidate)、跟随者(Follower)。Raft 算法提供了和 Paxos 算法相同的功能和性能,但是 Raft 算法更加容易理解且更容易构建实际的系统。

Raft 算法主要分两部分:领导者选举和日志复制。

(1) 领导者选举。

Raft 算法将时间划分成为任意不同长度的任期(Term)。任期用连续的数字进行表示。每一个任期的开始都是一次选举(Election),一个或多个候选者会试图成为领导者。如果一个候选者赢得了选举,它就会在该任期的剩余时间担任领导者。在某些情况下有可能没有选出领导者,那么,将会开始另一个任期,并且立刻开始下一次选举。Raft 算法保证在给定的一个任期最多只有一个领导者。

Raft 算法使用心跳机制来触发选举。当服务器启动时,初始化为跟随者。领导者向所有跟随者周期性发送心跳信息。如果跟随者在选举超时时间(Election Timeout)内没有收到心跳信息或投票请求,就会发起领导者选举。值得注意的是,每个服务器的选举超时时间彼此不一样,以避免同时发起领导者选举。

跟随者将当前任期加一然后转换为候选者。它首先给自己投票并且给集群中的其他服务器发送投票请求。候选者如果收到多数的投票,则赢得选举,成为领导者;如果被通知已经选出领导者,则自动切换为跟随者。有可能一轮选举中,没有候选者收到超过半数节点投票,那么将进行下一轮选举。

(2) 日志复制。

领导者接收到客户端发来的请求,创建一个新的日志项,并将其追加到本地日志中,接着领导者通过发送追加条目 RPC(Remote Procedure Call,远程过程调用)请求,将新的日志项复制到跟随者的本地日志中,当领导者收到大多数跟随者的成功响应之后,则将这条日志项应用到状态机中,可以理解成该条日志写成功了,最后领导者返回日志写成功的消息响应客户端,流程如图 3-9 所示。

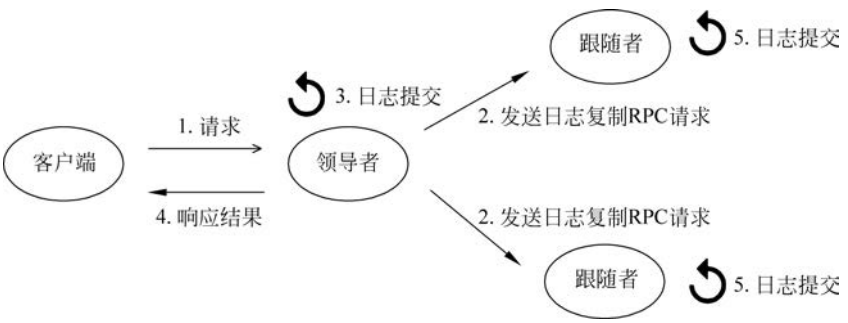

图 3-9　日志复制流程

3. 分布式数据库理论

分布式数据库一般遵守 CAP 理论和 BASE 理论。

1) CAP 理论

CAP 理论指的是在一个分布式系统中,一致性(Consistency)、可用性(Availability)和

分区容错性(Partition Tolerance)这三个要素最多只能同时实现两点,不可能三者兼顾。

一致性(C):在分布式系统中,所有节点在同一时刻的数据都是一致的。

可用性(A):在集群中一部分节点故障后,集群整体还能响应客户端的读写请求,即每个请求不管成功与否都能得到响应。

分区容错性(P):由于分布式系统通过网络进行通信,网络是不可靠的。当任意数量的消息丢失或延迟到达时,系统仍会继续提供服务。

(1) AP。也就是强调可用性和分区容忍性,放弃一致性。允许系统返回不一致的数据。当然,在采用 AP 设计时,也可以不完全放弃一致性,转而采用最终一致性。

(2) CP。也就是强调一致性和分区容忍性,放弃可用性。当出现网络分区的情况时,受影响的服务需要等待数据一致,在等待期间无法对外提供服务。

(3) CA。也就是强调一致性和可用性,放弃分区容忍性,也就意味着系统不再是分布式系统了。传统的关系数据库都采用这种设计原则,因此扩展性比较差。

2) BASE 理论

BASE 理论是由 eBay 架构师提出的,是 Basically Available(基本可用)、Soft State(软状态)和 Eventually Consistent(最终一致性)三个要素的缩写。

Basically Available(基本可用):分布式系统在出现不可预知故障时,允许损失部分可用性,相比较正常的系统而言会有响应时间上的损失和系统功能上的损失。

响应时间上的损失:如正常情况下,一个在线搜索引擎需要在 0.5 秒之内返回给用户相应的查询结果,但由于出现故障,查询结果的响应时间增加了 1～2 秒。

系统功能上的损失:如正常情况下,在一个电子商务网站上进行购物时,消费者几乎能够顺利完成每一笔订单,但是在一些节日大促购物高峰时,由于消费者的购物行为激增,为了保护购物系统的稳定性,部分消费者可能会被引导到一个降级页面。

Soft State(软状态):允许系统中的数据存在中间状态,并认为该中间状态的存在不会影响系统的整体可用性,即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

Eventually Consistent(最终一致性):所有的数据副本,在经过一段时间的同步之后,最终都能够达到一个一致的状态。因此,最终一致性的本质是需要系统保证最终数据能够达到一致,而不需要实时保证系统数据的强一致性。

3.2 数据库的设计原则

3.2.1 关系数据库的设计原则

1. 数据安全可靠

关系数据库的特点之一就是数据库管理系统能提供统一的数据保护功能来保证数据的安全可靠和正确有效。数据库的数据保护主要包括数据的完整性和安全性:数据库的完整性是指数据的正确性和相容性,防止数据库中存在不符合语义的数据,也就是防止数据库中存在不正确的数据;数据库的安全性是指保护数据库以防止不合法的使用造成的数据泄露、更改或破坏。数据库中的数据是从外界输入的,而数据的输入由于种种原因,会发生

输入无效或产生错误信息,保证输入的数据符合规定,成为数据库系统尤其是多用户的关系数据库系统首要关注的问题,数据完整性因此而提出。另外,数据库中大量数据的集中存放和管理,日渐成为非法入侵者攻击的焦点,数据库的安全越来越多地受到网络安全、操作系统安全、用户等多方面因素的影响,已经成为了信息安全的重要研究课题之一。

1) 关系数据库的完整性

为了保证数据库的完整性,数据库管理系统一般都提供了对数据库完整性进行定义、检查及违规处理的机制,并把用户定义的数据库完整性约束条件作为模式的一部分存入字典中。数据库管理系统检查数据是否满足完整性约束条件的机制称为完整性检查,一般在对数据库中的数据进行增、删、改操作后开始检查,也可以在事务提交时检查,检查这些操作执行后数据库中的数据是否违背完整性约束条件。当数据库管理系统发现用户的操作违背了完整性约束条件时,就会采取一定的动作,如拒绝执行该操作或级联执行其他操作,进行违约处理以保证数据的完整性。

关系数据库的完整性控制主要包括 3 类:实体完整性、参照完整性与用户自定义完整性。实体完整性规定关系中的主码取值唯一且非空;参照完整性规定关系的外码取值为被参照关系主码的值或取空值;除此两类必须满足的完整性约束条件以外,关系在不同的应用环境当中,往往还需要一些特殊的约束条件,用户自定义完整性就是针对某一具体关系数据库的约束条件,它反映某一具体应用所涉及的数据必须满足的语义要求。

2) 关系数据库的安全性

数据库必须具有坚固的安全系统,才能控制可以执行的活动以及可以查看和修改的信息。无论用户如何获得对数据库的访问权限,坚固的安全系统都可以确保对数据进行保护。

在 SQL Server 中,用户要经过两个安全性阶段:身份验证和授权。身份验证决定用户能否连接到服务器。它有 Windows 认证和 Windows 与 SQL Server 混合认证两种模式,其中 Windows 认证更为安全,因为 Windows 操作系统具有较高的安全性,而 SQL Server 认证管理较为简单。授权阶段验证已登录服务器的用户能否连接 SQL Server 实例的权力,只有授权的用户和系统才能访问被保护的数据。在身份验证阶段利用账号连接到服务器后,只表明该账户通过了 Windows 认证或 SQL Server 认证,并不代表用户就能访问数据库,登录者必须要有用户账号才能进一步操作数据库中的数据。

2. 高效查询机制

构建索引机制是实现关系数据库高效查询的一种常用的方式。索引是数据库管理系统中一个排序的数据结构,以协助快速查询数据库表中的数据。索引类型有 3 种:普通(Normal)索引、唯一(Unique)索引和全文(Fulltext)索引。

普通索引:也叫非唯一索引,是最普通的索引,没有任何的限制。

唯一索引:要求键值不能重复。另外需要注意的是,主键索引是一种特殊的唯一索引,它还多了一个限制条件,要求键值不能为空。主键索引使用主键创建。

全文索引:针对比较大的数据,如果要解决模糊查询效率低的问题,可以创建全文索引。只有文本类型的字段才可以创建全文索引,如 char、varchar、text。

目前比较常用的两种索引实现方式是B+树和哈希索引。B+树采用平衡树结构，由非叶节点和叶节点两种节点组成，数据存储在叶节点中，非叶节点只包含指针。B+树查找速度快且稳定，相较于B−树，相同数据量访问磁盘次数更少。同时，每个叶节点包含到相邻节点的链接，方便进行范围查询。哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值。哈希索引在检索时不需要像B+树那样从根节点到叶节点逐级查找，只需一次哈希算法即可定位到相应的位置，速度非常快。但是哈希索引也存在一定的弊端，如无法进行范围查找与前缀查找等。

除了索引机制外，关系数据库还可以将用户访问频繁的数据构建缓存，在用户访问时先查询缓存中数据。同时缓存是内存级，访问速度快。

3. 良好的扩展性

关系数据库在面对数据访问的压力时，通常采用扩展数据库的技术来应对，如备份(Replication)、联合(Federation)、分片(Sharding)、去规格化(Denormalization)和SQL调优(SQL Tuning)。

备份通常指的是一种允许用户在不同的机器上存储相同数据的多个副本的技术。联合(也称为功能分区)是指按功能对数据库进行分割，而不是使用一个单一的、整体的数据库，从而减少备份延迟，提高吞吐量。分片(也称为数据分区)是一种与分区相关的数据库架构模式，将数据的不同部分放到不同的服务器上，不同的用户将访问数据的不同部分，从而有助于提高系统的可管理性、性能、可用性和负载均衡。去规格化试图以牺牲部分写性能为代价来提高读性能，通过在多个表中写入数据来避免昂贵的数据连接操作。在大多数系统中，读操作的数量可能远远超过写操作，达到100∶1，甚至1000∶1，导致依赖于复杂数据库连接操作的读操作代价极大，需要在磁盘操作上花费大量时间。一旦数据通过联合和分片等技术分布，管理跨数据中心的连接操作将进一步增加复杂性。去规格化可以避免对这种复杂连接操作的需要。

3.2.2　NoSQL的设计原则

1. CAP理论与BASE理论

CAP理论说的就是一个分布式系统最多只能同时满足一致性、可用性和分区容错性这三项中的两项，不可能三者兼顾。在分布式系统中，由于分区容忍性是必须实现的，因此只能在一致性和可用性之间进行权衡。

BASE理论是对CAP中一致性和可用性权衡的结果，是基于CAP理论逐步演化而来的，其核心思想是即使无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

2. 灵活的数据模型选择

随着信息化技术的高速发展，数据量呈现爆发性增长，其中以非结构化数据为主。大数据中约80%是图片、音频等半结构化、非结构化数据。传统的关系数据库不能有效地存储这种数据。而NoSQL提供灵活的数据模型，可以很好地处理非结构化/半结构化数据。根据数据模型，NoSQL可分为4类，可以结合需求选择所需要的模型。

1）键值数据库

键值数据库适用于涉及频繁读写、数据模型简单的应用、内容缓存相关应用，如购物车、存储配置和用户数据信息的移动应用。但是有一点需要注意，键值数据库没有通过值查询的途径。

2）列族数据库

列族数据库适用于分布式数据存储与管理、数据在地理上分布于多个数据中心的应用、可以容忍副本中存在短期不一致情况的应用、拥有动态字段的应用以及拥有潜在大量数据的应用。

3）文档数据库

文档数据库适用于存储、索引和管理面向文档的数据或者类似的半结构化数据，如使用 JSON 数据结构的应用、使用嵌套结构等非规范化数据的应用。但是文档数据库不支持文档间的事务，若对这方面有需求则不能选择该方案。

4）图数据库

图数据库专门用于处理具有高度相互关联关系的数据，如适合于社交网络、依赖分析、推荐系统以及路径寻找等。

3.2.3 NewSQL 的设计原则

1. 设计出发点

由于 NoSQL 数据库系统不具备高度结构化查询的特性，也不能提供 ACID 的操作，NewSQL 逐渐登场。NewSQL 数据库的设计与开发通常要关注以下内容。

（1）事务属性（Transactional Property）：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

（2）容错性（Fault Tolerance）：故障处理（Failure Handling）、恢复（Recovery）、高可用（High Availability）、支持复制（Support For Replication）、复制方法（Replication Method）、复制类型（Replication Type）和副本同步（Replica Synchronization）。

（3）数据存储（Data Storage）：数据模型（Data Model）、数据存储（Data Storage）、内存支持（In-memory Support）、分区（Partitioning）和对二级索引的支持（Support For Secondary Index）。

（4）数据处理（Data Handling）：数据过期控制（Data Expiration Control）、数据压缩（Data Compression）、触发器（Trigger）、对 MapReduce 的支持（Support For MapReduce）、并发控制（Concurrency Control）、分析支持（Analytics Support）、数据库即服务（Database as a Service）和机架位置感知（Rack Locality Awareness）。

2. 决策重点

一致性（Consistency）、性能（Performance）、持久性（Durability）、故障处理（Failure Handling）、复制（Replication）、存储（Storage）、处理（Processing）、审计（Auditing）和并发性（Concurrency）被认为是决策的核心应用问题。可以通过权衡每个关注点的重要性来选择最终系统。

3.3 数据库的评价标准

3.3.1 吞吐量

吞吐量指的是系统在单位时间内处理请求的数量，其中 TPS(Transactions Per Second)、QPS(Queries Per Second)是吞吐量常用的量化指标。

1. TPS

TPS 即每秒执行的事务总数，也即事务数/秒。一个事务是指客户端向服务器发生请求然后服务器做出反应的过程。若在一秒内，系统完成 N 个事务，则系统的 TPS 为 N。

2. QPS

QPS 即每秒执行的查询总数，也即查询数/秒，是对一个特定的查询服务器在规定时间内所处理查询量多少的衡量标准。

3. 响应时间

响应时间指的是执行一个请求从开始到最后收到响应数据所花费的总体时间，即从客户端发起请求到收到服务器响应结果的时间。直观上看，这个指标与人对性能的主观感受是非常一致的，因为它完整地记录了系统处理请求的时间。

4. IOPS

IOPS (Input/Output Per Second)即每秒的输入输出量(或读写次数)，是衡量磁盘性能的主要指标之一。IOPS 是指单位时间内系统能处理的 I/O 请求数量，一般以每秒处理的 I/O 请求数量为单位，I/O 请求通常为读或写数据操作请求。随机读写频繁的应用，如 OLTP，IOPS 是关键衡量指标。

3.3.2 数据的一致性

在分布式环境中，一些应用为了提高可靠性和容错性，通常会将数据备份，同一份数据保存几个副本分别存储在不同的机器。由于分布式环境的复杂性，通常会出现网络、机器故障等情况，导致同一份数据的各个副本在同一时间可能有多种值，即数据不一致。

1. 强一致性(线性一致性)

系统中的某个数据被成功更新后，后续任何对该数据的读取操作都将得到更新后的值。

2. 弱一致性

数据更新后，如果能容忍后续的访问只能访问到部分或者全部访问不到，则是弱一致性。

3. 最终一致性

最终一致性是弱一致性的特殊形式，不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化。简单

说,就是在一段时间后,节点间的数据会最终达到一致状态。

最终一致性根据更新数据后各进程访问到数据的时间和方式的不同,又可以区分为如下几类。

(1) 因果一致性。如果进程 A 通知进程 B 它已更新了一个数据项,那么进程 B 的后续访问将返回更新后的值,且一次写入将保证取代前一次写入。与进程 A 无因果关系的进程 C 的访问,遵守一般的最终一致性规则。

(2) "读己之所写"一致性。当进程 A 自己更新一个数据项之后,它总是访问到更新过的值,绝不会看到旧值。这是因果一致性模型的一个特例。

(3) 会话一致性。这是上一个模型的实用版本,它把访问存储系统的进程放到会话的上下文中。只要会话还存在,系统就保证"读己之所写"一致性。如果由于某些失败情形令会话终止,就要建立新的会话,而且系统的保证不会延续到新的会话。

(4) 单调读一致性。如果进程已经看到过数据对象的某个值,那么任何后续访问都不会返回在那个值之前的值。

(5) 单调写一致性。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性,就非常难以编程了。

上述最终一致性的不同方式可以进行组合,如单调读一致性和"读己之所写"一致性就可以组合实现。并且从实践的角度来看,这两者的组合,可以实现读取自己更新的数据。

3.3.3 可用性

可用性是指任何客户端的请求都能得到响应数据,不会出现响应错误。换句话说,可用性是站在分布式系统的角度,对访问本系统的客户的另一种承诺:一定会返回数据,不会返回错误,但不保证数据最新,强调的是不出错。如果系统每运行 100 个时间单位,就会出现 1 个时间单位无法提供服务,那么该系统的可用性是 99%。

1. 数据的持久度

持久度是指发生故障时,数据丢失的概率。数据库系统可以通过副本备份等方式有效提高数据持久度,抵御磁盘损坏等故障造成数据丢失的风险。

2. RTO 与 RPO

RTO(Recovery Time Object)和 RPO(Recovery Point Object)是传统数据库领域常见的两个衡量高可用的指标。

RTO 是指系统从灾难状态恢复到可运行状态所需的时间。RTO 数值越小,代表系统的数据恢复能力越强。

RPO 是指系统所允许的在灾难过程中的最大数据丢失量,是指当业务恢复后,恢复得来的数据所对应时间点。RPO 取决于数据恢复到怎样的更新程度,这种更新程度可以是上一周的备份数据,也可以是昨天的数据,这和数据备份的频率有关。为了改进 RPO,必然要增加数据备份的频率。

3. MTTR、MTTF、MTBF

MTTR (Mean Time To Repair,平均修复时间)指系统从发生故障到维修结束之间的时间段的平均值。

MTTF（Mean Time To Failure，平均无故障时间）指系统无故障运行的平均时间，取所有从系统开始正常运行到发生故障之间的时间段的平均值。

MTBF（Mean Time Between Failure，平均失效间隔）指系统两次故障发生时间之间的时间段的平均值。

系统可用时间用 MTBF 和 MTTR 来计算，即 MTBF/(MTBF + MTTR)。

3.3.4　并发性

并发性通常是指系统能够同时并行处理很多请求。高并发相关常用的一些指标有响应时间（Response Time）、吞吐量（Throughput）、QPS、并发数等。

并发数是指系统同时能处理的请求数量，这个也反映了系统的负载能力。

3.3.5　可扩展性

可扩展（Scalable）指数据库系统在通过相应升级（包括增加单机处理能力或者增加服务器数量）之后，能够达到提供更强的服务能力，提供更强处理能力。扩展性（Scalability）指一个数据库系统通过相应的升级之后所带来处理能力提升的难易程度。

思考题

1. 简述分布式数据库与集中式数据库的区别。
2. 数据库存储引擎不仅限于教材中罗列的，查阅相关资料，介绍其他存储引擎。
3. 对于 CAP 理论，为什么不能同时兼顾三者？举例说明情况。
4. 谈一谈如何优化 Raft 算法。
5. 查询其他关于数据库的评价标准。