DDIA读书笔记

分布式系统的世界

数据复制

面临的挑战

- 采用分布式存储系统的原因
  - 扩展性：当数据量或读写负载很大时，完全超出单机处理极限，需要将负载分散到多台机器上。
  - 高可用性：多台机器可以提供冗余，当某台机器出现故障时，可以迅速切换到其他机器，系统仍然能正常工作。
  - 降低延迟：当用户分布范围很广时，如全球服务，那么在多地建立数据中心，用户就近访问可以享受较低延迟。
- 即如何在多个节点上保存相同数据的副本
- 挑战在于，复制的同时数据也在持续的修改，如何使得复制完毕后，副本与数据源一致

三种方案

- 主从复制
  - 原理
    - 指定一个主节点，其余为从节点。主节点负责处理写请求，从节点负责处理读请求读请求可以被主节点或从节点处理
    - 写请求只能由主节点处理，主节点本地完成写入后，将数据更改日志或者更改流发送给所有从节点。从节点据此进行本地修改，且严格保持与主节点相同的写入顺序
  - 数据复制的过程分为
    - 同步复制：主节点需等待从节点确认完成写入后，再返回给客户端完成
    - 异步复制：主节点发完消息直接返回
    - 折中方案，半同步复制：只配置一个同步从节点，其余节点均为异步
  - 异常处理
    - 配置新节点
      - 在时刻T对主节点数据做一个快照
      - 将快照数据复制到新配置的从节点
      - 从节点连接到主节点并请求时刻T之后的数据更改日志，将日志中的变更操作应用到本地数据，从而将从节点数据追赶上主节点。之后再正常接受来自主节点的数据流变化。
    - 处理失效节点
      - 如何确定节点失效？节点间会定时互相发送消息，如果发现某个节点在某段时间内没有响应，则认为该节点失效。该方式称为心跳机制。
      - 从节点失效：从节点的本地磁盘上保存了所有的数据变更日志。当从节点崩溃恢复后，根据本地磁盘日志的最后一笔事务，连接到主节点，并请求从该事务后中断期间的所有数据变更，然后在本地追赶恢复。
      - 主节点失效
        - 主节点失效后，必须立刻切换到其他可用从节点，并将其提升为新的主节点。
        - 候选节点应与原主节点的数据差异最小，从而最小化数据丢失的风险。
        - 当原主节点恢复时，也应降级为从节点，不再向外发送数据变更流。
        - 这个切换过程涉及多个操作，难以做到原子，因此会有很多问题。
          - 2. 脑裂：即选举过程有可能选出两个主节点。
          - 3. 超时时间设置：系统通过心跳机制来确定某个节点是否失效。超时时间不能设置的太长，因为这样系统恢复的时间就越长（因为失效更晚被界定），也不能设置得太短，因为这会导致很多不必要的切换，徒增系统负担。
          - 1. 切换到新主节点后，原主节点发来数据变更流

- 多主节点复制
  - 主从复制模式中，整个系统的写请求都依赖单个主节点，因此存在单点故障的隐患。
  - 多主复制则是在系统中建立多个主节点，适用于多个数据中心的情况。
  - 优势
    - 1. 性能提升：多数据中心，可就近完成用户读写，延迟低
    - 2. 容忍数据中心停机：某个数据中心故障时，其他数据中心继续运行，并且当发生故障的数据中心归队时，复制会自动赶上。
    - 3. 写请求更能容忍网络问题：数据中心之间的通信通常穿过公网，不如数据中心内的专网可靠。单主配置时，不同数据中心的写请求都是通过公网请求到唯一的主节点进行处理。而多主配置时，写请求在数据中心内即可完成，只有不同数据中心的数据同步时需要通过公网。
  - 劣势
    - 多主冲突
      - 方案1: 给每个写入一个唯一的ID。当写入冲突时，挑选最新ID的写入，并丢弃其他写入。但这种会造成修改丢失。
      - 方案2: 保留冲突的所有信息，先进行合并，不能合并的由用户来指定保留哪个。

- 无主节点复制
  - 写入时只需大多数节点写入成功，我们便认为写入成功
  - 问题
    - 读写同时发送，写操作可能只在部分节点完成，此时读取返回无法确定是否为新值
    - 写入不原子：写入时部分节点成功，部分失败，且写入的总成功数小于w，则对外界看来写入失败，不会触发回滚。但事实上已经有部分节点写入新值，
    - 写入w个新值成功，但之后其中部分节点失效，并恢复为旧数据，造成实际具有新值的节点小于w。

数据切分

- 当单张表的数据越来越多时，例如达到100G时，其读性能会明显下降。此时，就需要对数据进行切分。
- 垂直切分
  - 本质是做归类
  - 垂直分库
    - 垂直分库基于功能来分，将某个功能涉及的表放在同一库中。垂直分表要更细致一些。
    - 不同业务功能解耦：就是一个微服务一个库类似概念
  - 垂直分表
    - 当某个表字段较多时，可以新建一张扩展表。将不常用或字段长度较长的字段拆分出去
    - MySQL底层是通过数据页存储的，一条记录占用空间过大会导致跨页，造成额外的性能开销，大表拆小表，每行的体积变小了，能避免跨页。
  - 缺点
    - 写入一条记录可能需要写多张表
    - 本质上没有解决单表数据量过大的问题
- 水平切分（分片）
  - 从根本上个解决数据量大的问题
  - 一张表可以拆成三张表，表结构均相同，只是数据平均分了。三张表一般分散到三台机器上，整体上具有了更多的计算资源
  - 切分包含两次映射，一次是关键字到逻辑分片的映射，一次是逻辑分片到物理节点的映射。下面讨论的是关键字到逻辑分片的映射，集群中逻辑分片到物理节点的映射一般由Zookeeper这种工具来维护。
  - 缺点
    - 跨分片的事务一致性难以保证。假如退款表和支付表不在一个分片内，就没法一同回滚
    - 跨分片join问题
  - 基于关键字区间划分
    - 适合区间查询
    - 存在热点问题
      - 解决方案：多个列组成关键字，降低关键字称为热点的概率。或者，如果关键字成为热点，则在关键字开头或结尾增加一个随机数，强制分配到不同分区。但这样需要额外记录哪些关键字进行来特殊处理，否则查询时就会失败。
  - 基于关键字哈希划分
    - 数据分布均匀，不易出现热点
    - 难以区间查询
    - 扩容需要迁移旧数据(一致性哈希算法能较好避免此问题)
- 数据分区
  - Hash分区
  - 范围分区
  - 方式
  - 分区再平衡
  - 请求路由

分布式系统的麻烦

- 不可靠的网络
- 不可靠的时钟
- 没有统一的共识

如何让分布式系统看起来像单机系统

- 线性一致性
  - CAS寄存器
  - 原子事务提交
  - 全序广播
  - 锁和租约
  - 成员/协调服务
  - 唯一性约束

派生系统: 离线分析

- 核心问题
  - 容错
  - 数据分片
- Unix哲学
  - 简单, 组件化
  - mapReduce
- Join

派生系统: 实时分析

- 消息处理
  - AMQP/JMS风格的消息代理
  - 基于日志的消息代理
- 流处理中的Join

数据系统的未来

指标: 如何评价一个数据系统

- 可靠性：即使发生故障, 系统也能正常运行的能力
- 可扩展性：负载增加时, 系统能有效的保持其性能的能力
- 可维护性：简单, 易操作, 可演化

数据模型: 如何理解数据世界

- 关系模型
  - 关系型数据库（RDB）
  - 支持一对多，多对一，多对多（尽管实现起来比较繁琐）
  - 强事务
  - 由于应用广泛，目前有很多查询优化器
  - 采用 schema on write，比较笨重：在存储之前就制订好存储格式
- 文档模型
  - 适合一对多的场景，便于一次性将所有数据读出
  - 采用 schema on read，比较灵活
    - 在读的时候才知道存储格式
    - write的时候直接存储
- 图模型：适合多对多场景并能记录路径关系

数据该如何存储

- 事务型数据库(OLTP)
  - 基于日志结构的存储引擎
    - 索引
      - Hash索引
        - 将key加载到内存，并建立key与文件中存储位置的映射（hash map）。查询时，通过key找到value在磁盘文件中的偏移量，定位到磁盘文件指定位置读取数据。
        - 随着时间的推移，磁盘文件会越来越大，可以通过分段控制文件的大小，将一个大文件，拆为一个个固定大小的段。每个段在内存中维护一个hash map。对于一次查询，会先检查最新的hash map，如果该hash map没有这个key，则去第二新的hash map去找，以此类推。
        - 崩溃恢复：如果数据库重启，内存中的hash map将丢失。如果重新扫描各个段文件并建立相应的索引将比较耗时，一个可行的方案是，将每个段的hash map快照存在磁盘上，恢复时只需将快照加载到内存中即可。
        - 优点
          - 追加、分段、合并都是顺序写，相比随机写要快的多，且不易碎片化
          - 并发和崩溃恢复时，更简单。因此每次写入，都是先在磁盘上追加，然后更新内存，即先持久化。内存丢失的数据，可以完全依赖磁盘恢复现场。
        - 缺点
          - 所有的key都必须放入内存：当key很多时，内存不够用
          - 区间查询低效：因为hash map是一对一映射，区间查询时只能遍历比较
        - 使用场景：key不多，且value更新频繁。如记录用户粉丝数，点赞数等
      - 基于SSTable的LSM-Tree
        - TiDB 底层索引结构
        - SSTable (sorted string table) 是一种存储模式
        - LSM-Tree（Log_Structured Merge-Tree）是基于该存储模式在内存中的数据结构。
        - 哈希索引在合并段的时候，需要将待合并的两个段的内容全部加载到内存，然后进行压缩合并，再持久化回磁盘中。这意味着文件大小不能超过内存大小，并且合并效率较低。同时哈希索引必须在内存中保存所有键的索引，且不支持范围查询。SSTable的出现便是解决上述三个问题。
        - 原理：段内每个key只出现一次，且按照key字母序排序
        - 相比哈希索引的优势
          - 合并更加简单高效
          - 无需在内存中保存所有键的索引，索引可以是稀疏的
    - 性能分析
      - 写：O(1)，每次在文件的末端追加一行
      - 读：O(n)，从后往前遍历文件中的每一行
    - 优点
      - 只支持追加的数据文件，每行为一个KV对
      - 如果更新某个key的值，旧版本的值不会被覆盖，而是新增一行记录，查询时某个key的值以最新的为准。
      - 充分利用了磁盘寻道特性，比随机写性能更优；
      - 读取时一次寻址，即可将value从磁盘读取到内存
  - 基于页的存储引擎
    - B/B+ Tree
      - 原理：众所周知的B_Tree结构。将数据分解成固定大小的页，一般为4KB，每个树节点的大小便是一页，每个页面使用地址来标识，以被其他的页引用。每个页包含若干个键和对子页的引用(图中的ref)，每个子页都负责一个连续范围的键。查询时，从根节点开始逐层向下跳转，直到叶子节点，找到key所对应的值
      - 为什么固定大小：可变大小不利于维护，固定大小虽然有浪费空间的可能，但是降低了维护难度
      - 为什么取4KB：因为页是系统读写的最小单位，不能更小。当需要存储的数据大于4KB时，可以将该页分裂为多个页来扩展。
- 分析型数据库(OLAP)
  - 又称数据仓库
  - 列式存储

数据该长什么样

- 普通文本：CSV, JSON, XML, 二进制
- 特定协议
  - Thrift
  - Protocol Buffer
- 特定格式：Avro

数据写到一半, 系统挂了怎么办

- 事务
  - ACID：隔离性(I)：并发进行多个事务时不应该互相干扰。例如某个事务多次写入，另一个事务观察到的应该是其全部完成或者一个都没完成的结果，而不应该看到中间的部分结果。
  - 隔离级别
    - 读未提交（read uncommited）
    - 读已提交（read committed）：解决脏读
    - 可重复读（repeatable read）
      - InnoDB默认级别
      - 可重复读一般基于快照来实现，因此又有快照隔离的说法。
      - 由于多个正在进行的事务可能会在不同的时间点查看数据库状态，所以数据库保留了对象多个不同的提交版本，并基于版本来控制事务读取时的可见性。该技术称为多版本并发控制(MVCC)
      - 并未解决幻读问题：比如新增数据，但此时原来的事务读取到的还是原来的数据。
      - 通过加行锁或间隙锁，可以避免幻读问题
    - 串行化
  - 实现可串行化隔离级别：2PL（两阶段锁）