分布式系统工程实践

杨传辉
日照@淘宝

V 0.1

2010-10

1 引言

NOSQL 的资料很多，不过不成体系，让分布式系统开发工程师无所适从。笔者根据过去跟着阳老师开发类似 Google GFS/MapReduce/Bigtable 的系统以及对 Dynamo, PNUTS 等典型系统的理解尝试梳理流行的分布式存储和计算系统的分类，设计及实现。本文结构安排如下：

- 基础知识：一个大规模数据处理系统工程师必备的基础知识；
- 关键技术实现：工程实践中遇到的典型问题的解决思路；
- 通用存储系统分类：讲述笔者关于存储系统如何划分的个人观点；
- 典型存储系统工程实现：选取典型的存储系统讲述大致实现；
- 通用计算系统分类：讲述笔者对于计算系统如何划分的个人观点；
- 典型计算系统工程实现：讲述典型计算系统的大致实现；
- 应用：存储&计算系统应用的一些实例；
- 工程实现注意事项：总结设计和开发过程中可能犯的一些错误；
- 致谢及参考资料：列出一些值得看的论文和网页资料；

每个章节涉及的话题都很大，由于笔者的水平实在是非常非常有限，只能说是尽力把自己知道并能够说明白的写下来，作为自己对过去工作的回忆。把其中任何一个话题讲明白都远远超出了我的能力范畴，写错的地方在所难免，各位同学发现问题尽管笑一笑，当然，欢迎任何形式的讨论，我会尽量和更多的同学讨论来不断完善这个文档。本文只是一个初始综述，后续将细化每一个问题并发表到博客中。

2 基础知识

本章描述工程实现需要的一些基础知识，由于篇幅的关系，只抽取一些认为对理解和设计大规模系统必要的基础知识进行描述。另外，假设读者了解 NOSQL 基本概念，做过或者看过一两个类似的系统，阅读过 GFS/Bigtable/Paxos 相关的论文。分布式理论有一个特点是：大致的做法是很容易想到的，但是完全没有问题的做法非常难想，理解理论的用处就在于区分出想法的问题在哪儿以及实现的难度。

2.1 硬件基础

分布式系统开发工程师需要了解硬件的大致价格，熟记硬件的性能。
硬件大致性能如下：

L1 cache reference	0.5ns
Branch mispredict	5ns
L2 cache reference	7ns
Mutex lock/unlock	100ns
Main memory reference	100ns
Send 1M bytes over 1Gbps network	10ms
Read 1M sequentially from memory	0.25ms
Round trip within data center	0.5ms
Disk seek	8~10ms
Read 1MB sequentially from disk	20~25ms

标记为红色性能参数比较常用，其中，磁盘的性能指标专指分布式平台专用的大容量SATA 磁盘，寻道时间为 8~10ms，顺序读取速率为 40~50MB。某些应用使用 SAS 磁盘或者 Flash 盘，性能较好，评估时需查看硬件的性能参数。磁盘和网络都有一个特征，一次读写的数据量越大性能越好，这是由硬件特征及底层软件算法决定的，如 tcp 慢连接和磁盘寻道时间长。

2.2 性能估算

给定一个问题，往往会有多种设计方案，而方案评估的一个重要指标就是性能，如何在系统设计时估算而不是程序执行时测试得到性能数据是系统架构设计的重要技能。性能估算有如下用途：

1) 多种设计方案选择；
2) 评价程序实现是否足够优化；
3) 向框架/服务提供方提出性能要求的依据；

很多同学喜欢通过查看程序运行时 CPU 及网络的使用情况来评价程序是否足够优化，这也是一种很重要的方法。然而，这种方法掩盖了不优化的实现，如 O(N)的算法被错误实现成 O(N^2)，网络收发冗余数据等。

性能评估需要假设程序的执行环境，如集群规模及机器配置，集群上其它服务占用资源的比例。对硬件性能指标有了初步认识以后，我们可以做出一些简单的判断，如：

某 K-V 引擎 RD：我们的 K-V 引擎单客户端同步读取每秒可以达到 18000/s。

问：是否批量读取？

答：是，每批读取 10 个记录。

由于 tcp Round trip 时间为 0.5ms，读取请求个数的理论极限为 2000/s，而上例中 K-V 引擎的 RD 却说单客户同步读取可以达到 18000/s，可以断定该 RD 指的是批量读取方式。且这已经是单机能够做到的极限值了。

下面我们通过几个实例说明如何进行性能评估。

1. *1GB 的 4 字节整数，内存排序时间为多少？*

拿到这个问题，我们往往会计算 CPU 运算次数，如快排的运算次数为 1.4 * N * log(N)，其中 1.4 为快排的系数，再根据 CPU 的运算频率计算出排序耗时。不过这种方法很土也不是很准，Jeff Dean 告诉我们可以这样估算：排序时间 = 比较时间（分支预测错误） + 内存访问时间。快排过程中会发生大量的分支预测错误，所以比较次数为 2^28 * log (2^28) ≈ 2^33，其中约 1/2 的比较会发生分支预测错误，所以比较时间为 1/2 * 2 ^ 32 * 5ns = 21s，另外，快排每次找到分割点都需要一遍内存移动操作，而内存顺序访问性能为 4GB/s，所以内存访问时间为 28 * 1GB / 4GB = 7s。因此，单线程排序 1GB 4 字节整数总时间约为 28s。

2. *Bigtable 设计的性能指标分析*

假设 Bigtable 总体设计中给出的性能指标为:

系统配置：50 台 4 核 8GB 内存 12 路 SATA 硬盘，同样数量的客户端；

Table：row name：16-byte，column：16-byte，value：1KB；64KB data block；no compression；

*Random reads (in disk)： 1KB/item*300item/s*50=15MB/s*

*Random reads (in memory)：1KB/item*4000item/s*50=200MB/s*

*Random writes：1KB/item*2000item/s*50=100MB/s*

*Sequential reads(in disk)：1KB/item*1000item/s*50=50MB/s*

*Sequential writes：1KB/item*2000item/s*50=100MB/s*

先看磁盘中的随机读取性能，由于在 Bigtable 的设计中每个随机读取都要读取一个 64KB 的大块，而磁盘中读取 64KB 数据时间为：磁盘寻道时间 + 读取时间 = 10ms + 64KB / 50MB/s = 12ms。所以每秒读取 300 个记录指多客户端读取或者单客户端异步/批量读取。由于每台机器有 12 个 SATA 大容量磁盘，随机读的理论值为 12 * 1s / 12ms = 1000 个/s。设计为每秒读取 300 个是考虑到有负载平衡等因素简单地打了一个折扣。

再看内存中的随机读取。一般来说，内存操作都是每秒 1W~10W。由于网络发送小数据有较多 overhead 且 Bigtable 内存操作有较多的内存开销，所以保守设计为单机每秒读取 4000 个记录。

其它的可类似分析。性能分析可能会很复杂，因为不同的情况下决定性能的瓶颈不一样，

有的时候是网络，有的时候是磁盘，有的时候甚至是机房的交换机。这种性能分析的经验是需要慢慢积累的。

最后，我们再看看某一个 MapReduce 应用的例子。MapReduce 可以简单地分为几个过程：Map 处理时间 + shuffle 和排序时间 + reduce 处理时间，虽然 shuffle、map 处理和排序可以部分并行，但性能估算的时候不必考虑。假设 50 台机器，原始输入为 50G，例中 MapReduce 应用的 map 函数处理时间为 100s，reduce 函数处理时间为 60s，shuffle 的中间结果数据量为 300G，reduce 输出的最终结果大小为 600M。

Map 处理时间 = 输入读取时间 + Map 函数处理时间 + 输出中间结果时间

其中，输入读取时间 = 50G / 2.5G = 25s (50 台机器，假设每台机器读取带宽为 50M/s)，

Map 函数处理时间 = 60s，

输出中间结果时间 = 300G / 15G = 20s (50 台机器，每台机器 12 个磁盘，假设用满 6 个磁盘，带宽为 6 * 50M = 300M)

所以，Map 处理时间 = 25s + 60s + 20s = 105s

Shuffle 和排序时间 = shuffle 时间 + 排序时间

其中，shuffle 时间 = 300G / 2G = 150s (50 台机器，假设每台机器的读取和写入带宽均为 40M，单机总带宽为 80M)

排序时间 = 单机排序 6G 的时间，假设每条记录为 1KB = 排序比较时间 + 访问时间，约为 25s

所以，shuffle 和排序的时间 = 150s + 25s = 175s

Reduce 处理时间 = reduce 函数处理时间 + 最终结果输出时间

其中，reduce 函数处理时间 = 100s，

最终结果输出时间 = 600M / 500M (50 台机器，单机写 DFS 假设时间为 10M/s) = 1s (忽略)

所以，例中的 MapReduce 应用运行一遍大致需要的时间 = Map 处理时间 + shuffle 和排序时间 + Reduce 处理时间 = 105s + 175s + 100s = 380s，当然，MapReduce 过程中还有框架的开销和其它应用的影响，我们可以简单地认为影响为 20%，所以总时间 = 380s + 380s * 20% = 456s，约为 7~8 min。

当然，MapReduce 应用实际的性能估算不会如此简单，实际估算时需要考虑每台机器上启动的 Map 和 Reduce 个数等因素，且需要根据实验的结果不断地验证和重新调整估算。但是，我们至少可以保证，估算的结果和实际不会相差一个数量级，估算结果可以用来指导初期的设计和 Map/Reduce Worker 的个数、Map/Reduce 任务数选择，评估应用的可优化空间并作为向 MapReduce 框架提供小组提出需求的依据。

性能估算是大规模系统设计中较难掌握的技能，开始性能估算时可能估计得很不准，不过不要气馁，通过在项目中不断练习，大规模系统的分析和设计能力才能做到有理可依。

2.3 CAP

CAP 是一个很时髦的概念，然而，对于设计和实现大规模分布式系统而言，只需要在脑海里面有一个粗略的概念即可。我们先看看 CAP 是怎么回事。

CAP 理论由 Berkerly 的 Brewer 教授提出，在最初的论文中，三者含义如下：

- 一致性(**C**onsistency)：任何一个读操作总是能读取到之前完成的写操作结果；

- 可用性(**A**vailability)：每一个操作总是能够在确定的时间内返回；
- 分区可容忍性(Tolerance of network **P**artition)：在出现网络分区的情况下，仍然能够满足一致性和可用性；

CAP 理论认为，三者不能同时满足，并给出了证明，简单阐述如下：假设系统出现网络分区为 G1 和 G2 两个部分，在一个写操作 W1 后面有一个读操作 R2，W1 写 G1，R2 读取 G2，由于 G1 和 G2 不能通信，如果读操作 R2 可以终结的话，必定不能读取写操作 W1 的操作结果。

然而，这种对一致性及可用性的定义方法在工程实践上意义不大，CAP 理论只是粗略地告诉我们“天下没有免费的午餐”。比如 Availability 的定义，10 秒钟停服务和 1 个小时停服务在工程实践中完全是两个概念。因此，我们往往会修改 CAP 的定义如下：

- 一致性(**C**onsistency)：读操作总是能读取到之前完成的写操作结果，满足这个条件的系统称为强一致系统，这里的“之前”一般对同一个客户端而言，但可能是一个客户端的多个 Session；
- 可用性(**A**vailability)：读写操作在单台机器发生故障的情况下仍然能够正常执行，而不需要等到机器重启或者机器上的服务分配给其它机器才能执行；
- 分区可容忍性(Tolerance of network **P**artition)：机房停电或者机房间网络故障的时候仍然能够满足一致性和可用性；

工程实践对网络分区考虑较少，一般可以认为：一致性和写操作的可用性不能同时满足，即如果要保证强一致性，那么出现机器故障的时候，写操作需要等机器重启或者机器上的服务迁移到别的机器才可以继续。

2.4 一致性模型

Amazon 的 CTO 专门在官网中阐述了一致性模型，足见其重要性，可以认为，一致性要求直接决定了存储系统设计和实现的复杂度。

为了更好的描述客户端一致性，我们通过以下的场景来进行，这个场景中包括三个组成部分：

- 存储系统

  存储系统可以理解为一个黑盒子，它为我们提供了可用性和持久性的保证。

- Process A

  Process A 主要实现从存储系统 write 和 read 操作

- Process B 和 Process C

  Process B 和 C 是独立于 A，并且 B 和 C 也相互独立的，它们同时也实现对存储系统的 write 和 read 操作。

下面以上面的场景来描述下不同程度的一致性：

- 强一致性

  强一致性（即时一致性） 假如 A 先写入了一个值到存储系统，存储系统保证后续 A,B,C 的读取操作都将返回最新值

- 弱一致性

假如 A 先写入了一个值到存储系统，存储系统不能保证后续 A,B,C 的读取操作能读取到最新值。此种情况下有一个“不一致性窗口”的概念，它特指从 A 写入值，到后续操作 A,B,C 读取到最新值这一段时间。

- 最终一致性

最终一致性是弱一致性的一种特例。假如 A 首先 write 了一个值到存储系统，存储系统保证如果在 A,B,C 后续读取之前没有其它写操作更新同样的值的话，最终所有的读取操作都会读取到最 A 写入的最新值。此种情况下，如果没有失败发生的话，“不一致性窗口”的大小依赖于以下的几个因素：交互延迟，系统的负载，以及复制技术中 replica 的个数（这个可以理解为 master/salve 模式中，salve 的个数）。

一致性模型的变体如下：

- Causal consistency（因果一致性）

如果 Process A 通知 Process B 它已经更新了数据，那么 Process B 的后续读取操作则读取 A 写入的最新值，而与 A 没有因果关系的 C 则可以最终一致性。

- Read-your-writes consistency

如果 Process A 写入了最新的值，那么 Process A 的后续操作都会读取到最新值。但是其它用户可能要过一会才可以看到。

- Session consistency

此种一致性要求客户端和存储系统交互的整个会话阶段保证 Read-your-writes，数据库分库以后一般会提供这种一致性保证，使得同一个 Session 的读写操作发送到同一台数据库节点。

- Monotonic read consistency

此种一致性要求如果 Process A 已经读取了对象的某个值，那么后续操作将不会读取到更早的值。

- Monotonic write consistency

此种一致性保证系统会序列化执行一个 Process 中的所有写操作。

为了便于后续的说明，我们修改 Amazon CTO 关于最终一致性的定义。Dynamo 通过 NWR 策略提供的最终一致性主要是针对 Dynamo 的多个副本而言的，它们之间保持最终一致。不过对于用户，我们假设 N=3, W=2, R=2 的一种情况，用户先调用 W1 写 A 和 B 两个副本后成功返回，接着调用 W2 写 B 和 A 两个副本后成功返回，可能出现在副本 A 上 W1 先于 W2 执行，而在副本 B 上 W2 先于 W1 执行，虽然副本 A 和 B 都能够通过执行满足交换律的合并操作，比如基于”last write wins”的策略进行合并使得最终副本 A 和 B 上的数据完全一致，但是可能出现一些异常情况，比如副本 A 和 B 所在的机器时钟不一致，合并的结果是 W1 把 W2 给覆盖了，W2 的操作结果消失了。这显然与用户的期望是不一致的。

为了方便后续对系统进行划分，我们**把 Amazon Dynamo 这种需要依赖操作合并，可能会丢失数据的模型从最终一致性模型中排除出去**。最终一致性模型要求同一份数据同一时刻只能被一台机器修改，也就是说机器宕机时需要停很短时间写服务。Amazon Dynamo 提供的一致性模型我们归类到一般的弱一致性模型中。

2.5 NOSQL 与 SQL

NOSQL 可以认为是选取了 SQL 特性的子集，在扩展性和用户接口友好两个方面做了一个权衡。“越多选择，越多迷茫”，实践经验告诉我们，如果将 SQL 的功能完全暴露给用户，用户一定会使用一些我们不希望的功能，比如多表 join，外键，等等。NOSQL 的意义在于，我们预先定义一些特性，这些特性满足某一个应用的需求，并且只满足这些特性使得我们的系统很容易扩展。SQL 定义了一个功能全集，NOSQL 根据应用特点选取几种特定的应用定义不同的特性集合，以适应互联网数据量高速膨胀的需求。

一般来说，NOSQL 的应用会比 SQL 的应用更加注意可用性，所以 NOSQL 应用对外表现为经常可以选择最终一致性模型。不过，从通用系统的角度看，这里的最终一致性指：大多数操作允许读取老的数据，少数操作仍然希望读取最新的数据，并且应用不希望出现数据丢失的情况。所以，不能因为 NOSQL 就容忍数据丢失的情况，虽然这会极大地加大系统设计和实现的难度。

另外，NOSQL 不等于必须用 MapReduce 做计算模型，虽然二者经常结对出现，不过本质上是不相关的。

NOSQL 比较常见的模型包括：

- KV 模型：只支持最简单的针对<key, value>对的操作
- 支持简单 table schema 的模型，如 Bigtable 模型

由于 NOSQL 相对 SQL 而言更加注重扩展性、成本等，NOSQL 有一些共同的设计原则：

- 假设失效是必然发生的：NOSQL 注意扩展性和成本，机器数变多时，原本属于异常现象的机器故障变成一种正常现象，NOSQL 也采用一些比较便宜的普通 PC 机，要求通过软件的方法处理错误。
- 限定应用模式。从最为简单的 KV 应用模型，到复杂的支持用户自定义 schema 的 Bigtable 模型，NOSQL 支持的接口永远不可能和 SQL 相比。一般来说，NOSQL 系统都只支持随机读和顺序读，少量系统支持表索引，类似外键这种影响扩展性且不实用的功能基本是不需要支持的。
- 扩容：数据库扩容一般是成倍增加机器的，而 NOSQL 系统一般是一台或者少量几台构成一个机器组加入系统。一般有两种数据分布方法，一种是一致性 Hash，这个算法在 Dynamo 论文中有详细的介绍，另外一种方法是将整个表格分成连续的小段，每个小段是一个子表，由全局管理机器负责将每个小段分配到新加入的数据读写服务机器。

用一个例子说明取舍 SQL 的部分特性带来的好处。比如单机 SQL 的 add 操作，这是非常容易的，然而，在多机上的实现变得非常困难。因为我们需要操作多个副本，可能出现某些操作成功，某些永远不成功的情况，我们只能通过一些锁的方法来解决，比如分布式事务的两阶段悲观锁或者另外一种乐观锁。

Mysql 团队也有部分同学开始通过削减 SQL 模型不必要的特性来满足互联网数据高速增长的需求，它们发起了一个叫做 Drizzle 的项目。Drizzle 诞生于 MySQL（6.0）关系数据库的拆分。在过去几个月里，它的开发者已经移走了大量非核 心的功能（包括视图、触发器、已编译语句、存储过程、查询缓冲、ACL 以及一些数据类型），其目标是要建立一个更精简、更快的数据库系统。

2.6 Two-Phase commit

两阶段提交用于解决分布式事务，虽然分布式事务解决的代价比较大，不过理解两阶段锁协议能加深我们对“分布式系统哪些问题是困难的？”的理解。

Two-phase commit 的算法实现 (from <<Distributed System: Principles and Paradigms>>)：

协调者(Coordinator)：

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

参与者(Participants)

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked*/
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
```

```
    else if DECISION == GLOBAL_ABORT
      write GLOBAL_ABORT to local log;
  } else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
  }
```

另外，每个参与者维护一个线程专门处理其它参与者的 DECISION_REQUEST 请求，处理线程流程如下：

```
  while true {
    wait until any incoming DECISION_REQUEST is received;
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
      send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT;
      send GLOBAL_ABORT to requesting participant;
    else
      skip; /* participant remains blocked */
  }
```

从上述的协调者与参与者的流程可以看出，如果所有参与者 VOTE_COMMIT 后协调者宕机，这个时候每个参与者都无法单独决定全局事务的最终结果(GLOBAL_COMMIT 还是 GLOBAL_ABORT)，也无法从其它参与者获取，整个事务一直阻塞到协调者恢复；**如果协调者出现类似磁盘坏这种永久性错误，该事务将成为被永久遗弃的孤儿**。

一种可行的解决方法是当前的协调者宕机的时候有其它的备用协调者接替，用于同一时刻只能允许一个协调者存在，二者之间有一个选举的过程，这里需要用到 Paxos 协议。Jim Gray 和 Lamport 有一篇论文专门论述协调者单点的解决方法。

分布式事务执行过程中是需要锁住其它更新的，因此工程实践中需要降低锁的粒度，实现起来极其复杂，也影响效率，所以几乎所有的 NOSQL 系统都回避这个问题。

2.7 Paxos

Paxos 基本可以认为是实现分布式选举的唯一方法，其它的正确协议都是 Paxos 变种。Paxos 最为常见的用途就是单点切换，比如 Master 选举。Paxos 协议的特点就是难，理解 Paxos 可以提高学习分布式系统的信心。Paxos 选举过程如下：

- **Phase 1**

(a) A proposer selects a proposal number *n* and sends a *prepare* request with number *n* to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number *n* greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than *n* and with the highest-numbered proposal (if any) that it has accepted.

- **Phase 2**

(a) If the proposer receives a response to its *prepare* requests (numbered *n*) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered *n*

with a value *v*, where *v* is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
(b) If an acceptor receives an *accept* request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than *n*.

Paxos算法的证明有两个方面：一个是正确性，一个是可终止性。正确性很容易理解，只要Proposer的提议被接受，至少保证超过一半的Acceptor接受提议。另外一个方面是可终止性，我们可以想象一下，Paxos算法总是往前走的，在Phase 1，Proposer至少收集超过半数Acceptor希望接受的提议信息；在Phase2，Proposer将收集到的编号最大的提议发送给这些Acceptor。如果中间其它的Proposer提出编号更大的建议，提议被接受。不断重试，总会碰到一次提议成功的情况。不过理论上也可能出现特别差的情况，例如：
1, Proposer A 提议编号为N的建议，进入Phase 1；
2, Proposer B 提议编号为N+1的建议，进入Phase 1；
3, Proposer A 提议编号为N的建议，进入Phase 2；
4, Proposer A 提议编号为N+2的建议，进入Phase 1；
5, Proposer B 提议编号为N+1的建议，进入Phase 2；
如此循环，最后的结果是永远不可能有提议被接受，算法不终止。这个问题在理论上确实是没法解决的，需要选择一个distinguished proposer，工程实践时可以通过一些超时机制来实现，比如Proposer A在第5到10s提建议，Proposer B在第10到15s提建议。

3 关键技术实现

大规模分布式系统工程实现时一般会采用朴实的技术，每个技术点看起来都非常简单，但是组合起来威力很大，引入复杂的技术之前我们应该先想想工程上的实现，因为分布式系统本来就不是研究问题，而是一个系统工程。本章我们看一下常用的一些技术是如何实现的。

3.1 网络编程框架

服务器编程都需要有一个可控的网络编程框架。Taobao 公司开源了一个 tbnet 框架，这个框架设计非常优雅，我们结合 tbnet 说明网络编程框架的设计。

网络编程包括客户端编程和服务器端编程。客户端有同步和异步两种模式：同步模式下，客户端往 socket 连接中发送请求后等待服务器端应答；异步模式下，客户端往 socket 连接附带的队列中拷贝请求内容（给每个请求分配唯一的请求号）后立即返回，等到服务器端应答时，客户端的接收线程会调用相应的回调函数。Tbnet 客户端实现的是异步模型，因为同步模型可以通过异步模型来封装。服务器端监听客户端的连接，接收到客户端的请求后放到 socket 连接附带的任务队列中，该任务队列所在的线程会不断地从任务队列取任务并调用用户定义的任务处理函数。

一个网络编程框架至少包含三个组件：连接管理，任务队列，线程池。具体实现时，客户端和服务器端都有网络线程负责发送和接收网络包，并有超时检查线程负责连接超时管理。Tbnet 的 Transport 就是负责网络传输层的一个类，它负责开两个线程，一个用来传输，一个

检查超时，另外，它提供两个方法 listen 和 connect，用于服务器端的监听和客户端的连接。Transport 传输数据的线程就是事件循环，不断监听发生的事件，并调用事件处理函数。事件处理函数一般是将接收到的任务放入到服务器端的任务队列中。Tbnet 中还有一个 channel 的概念，这里面封装了异步模型需要的请求号，回调函数及相应的参数，网络传输线程收到服务器端的回复包时调用 channel 上的处理函数。Tbnet 的 ConnectionManager 用于底层的连接池管理。

网络编程框架需要将某些逻辑暴露给用户重载。Tbnet 暴露了如下的逻辑：

1, 服务器端接收到网络包时需要根据网络包编号创建相应的 Packet 对象，创建 Packet 的操作用户可重载；

2, 服务器端创建 Packet 后默认的处理方法是加入任务队列，用户可重载，比如用户可能需要将读请求和写请求加入不同的任务队列；

3, 任务队列所在的线程不断取队列中的任务并调用任务处理函数，用户可重载任务处理函数；

4, 客户端可以异步发送请求时自定义回调函数；

一般来说，服务器处理逻辑是类似的，可以封装一个通用的服务器编程框架，用户只需定义一些必要的处理函数就可以写出一个服务器程序。

3.2 HA 与 Replication

为了保证可靠性，需要实现复制，一般有三种复制模式：异步，强同步，半同步。

异步复制是 Mysql 的 Replication 采用的模式，实现最为简单，不过如果 Master 宕机立即切换到 Slave 或者 Master 机器出现磁盘故障会丢失最后的更新，异步模式如果要保证完全不丢数据一般采用可靠的共享存储实现。异步模式实现简单，Master 有一个线程不断扫描操作日志文件，将最新的修改发送给 Slave，Slave 有线程接受 Master 发送的更新操作并回放，接受日志和回放一般采用两个线程。如果 Slave 宕机，以后重启时向 Master 注册，将 Slave 最新的日志点告知 Master，Master 为这个 Slave 启动一个同步线程从最新的日志点开始不断地传输更新操作。

强同步模式下，每一个修改操作需要先写入 Slave，然后写入 Master 的磁盘中才能成功返回客户端。普通的强同步模式有一个问题，那就是 Slave 宕机也必须停止写服务，只能保证可靠性而不能保证可用性。有一种比较合适的折衷，Master 保存一个 Slave 机器列表，每个写操作都需要同步到 Slave 列表的所有的机器，如果发现某个 Slave 连接不上，将它从 Slave 列表中删除。工程实现时，Master 有一个线程负责先将数据都同步到 Slave，然后写入本地磁盘和内存，为了提高性能，需要将操作日志批量发送给 Slave，如果 Slave 宕机，将它提出 Slave 列表。比较麻烦的是 Slave 宕机重启后向 Master 注册，Master 可以选择立刻将 Slave 加入 Slave 列表并往新加入的 Slave 同步数据，与此同时，Slave 从 Master 获取落后的日志，当落后的日志全部获取完成时，Slave 和 Master 保持同步，如果这时 Master 宕机，Slave 是可以切换为 Master 继续提供服务的。

半同步模式指的是有 N 个 Slave，数据写入到其中的 K 个 Slave 并写入 Master 本地就可以成功返回客户端，只要 K >= 1，就可以保证当 Master 宕机时，可以通过分布式选举，比如 Paxos 选举算法选取数据最新的 Slave 继续提供服务。Berkerly DB 实现了这种半同步模式，有兴趣的同学可以参考下。一种可能的做法是：Master 对每个 Slave 创建一个同步线程用于数据同步，并维护一个协调者线程，每个 Slave 线程将数据同步完成后通过信号量通知协调者线程，协调者线程发现 K 个 Slave 已经同步完毕时可以返回客户端。工程实现是需要考虑

对多个 Slave 同时上下线，Slave 瞬断等各种异常情况模拟测试，这是非常复杂的。

HA 还有一个 Master 宕机后的切换问题。如果是数据库应用，我们一般会把数据库的数据写入到共享存储中使得数据库本身没有状态，这时，可以从机器池中任意选择一台机器作为 Slave 加载数据。而考虑到成本问题，在 NOSQL 系统中，Master 和 Slave 是 share-nothing 的。在强同步模式下，我们需要让 Slave 有识别自己是否和 Master 状态一致的能力，从而 Master 宕机时可以通过 DNS 或者 VIP 等手段迁移到 Slave，这里可以引入 Lease 机制；在半同步模式下，我们需要保证有 Slave 和 Master 状态一致，从而 Master 宕机时出发一次选举过程，通过 Paxos 协议产生新的 Master。

3.3 分裂

分裂一般对提供强一致性和最终一致性的系统而言。我们假设按照类似 Bigtable 中的数据划分方法对表格进行拆分，一个大表被拆分成大小接近 100MB ~ 200MB 的子表 tablet，每个子表服务一段连续的数据范围。但某个子表访问过于频繁时，需要分裂为两个子表，从而将服务迁移到其它机器。一个子表变成两个大小相近的子表的过程，就叫分裂。

子表分裂单机上的做法是不难的。通用的做法可以将需要分裂的子表拷贝出来，按照确定的分裂点写成两份，拷贝过程中新来的写操作记录操作日志，等到拷贝完成时锁住写操作，将记录到操作日志的写操作应用到新生成的两个子表中，释放写锁，分裂成功。如果单机存储引擎支持快照功能，做法会更加简单高效。分裂时只需要生成一个快照并将快照中的数据按照确定的分裂点写成两份，接着锁住写操作并将拷贝过程中新来的写操作应用到新生成的两个子表中。

Merge-dump 存储引擎分裂时还有更加简单的做法。我们以 Bigtable 的 Merge-dump 存储引擎为例，写操作以操作日志的形式写入到 SSTable 中，compaction 合并过程将多个 SSTable 合并为一个 SSTable，原来的 SSTable 通过定期的垃圾回收过程删除。Merge-dump 存储引擎上执行分裂操作不需要进行实际的数据拷贝工作，只需要将内存中的索引信息分成两份，比如分裂前子表的范围为(start_key, end_key]，内存中的索引分成两个范围(start_key, split_key]和(split_key, end_key]，compaction 合并的过程中为两个分裂后的子表生成不同的 SSTable 文件。

分裂的难点在于如何使得多机在同一个分裂点原子地分裂。这个问题涉及到多机之间协调，所以我们一般可以采用两阶段锁来解决，Yahoo 的 PNUTS 系统正是采用了这种做法。两阶段锁的性能问题倒是其次，工程实现中更为重要的是这个协议实现很复杂，非常难以构造各种异常 case 进行测试，很容易出现代码错误导致死锁的问题。

Bigtable 采用了另外一种思路。既然多机原子地执行分裂很难做，我们就做单机分裂好了。Bigtable 设计中，同一个子表 tablet 同一个时刻只能被一台工作机 Tablet Server 服务。由于只在一个服务节点进行分裂，问题得到了解决，底层的 GFS 系统会负责文件系统的可靠性和可用性保证。

当然，我们也可以将机器分成一个一个的 group，每一个子表都在某个 group 的每台机器存放一个备份，也就是说，如果数据复制 N 份，一个 group 就有 N 台机器。同一个 group 同一时刻只有一台机器提供写服务，其它机器提供读服务，当 group 中提供写服务的机器宕机时，由 group 中的其它机器顶替。加入新机器以 group 为单位，每次新增 N 台。分裂操作由 group 中提供写服务的 Master 节点执行，分裂操作写日志并同步到 Slave 节点，Slave 节点接收到分裂日志也执行分裂。如果 Master 分裂成功但是 Slave 分裂失败，也不会出现问题。因为只要不出现子表的迁移，在单机上分裂成功与否是不会影响正确性的。简单来说，同一

个 group 同一时刻总是有一个 Master 节点作为代表，Slave 节点上的状态与 Master 不一致时以 Master 为准。

工程实践中，分裂仍然是很复杂的，因此国内几乎所有的分布式存储系统都采用预先切分好 tablet 的方法。只要切分得比较细，系统支撑一两年是没有问题的，等到出现问题时可以整个系统停服务对数据重新划分。

3.4 迁移

我们仍然假设整个大表按照类似 Bigtable 中的方法被划分为很多的子表 tablet。子表迁移在集群主控机的指导下进行，迁移的做法和分裂有很多共通之处。

假设机器 A 需要将子表迁移到机器 B，迁移的做法与单机子表分裂时拷贝数据的方法类似。分为两个阶段，第一个阶段将机器 A 的待迁移子表的数据拷贝到机器 B，这个阶段新来的修改操作只记录操作日志；第二个阶段停止写服务，将第一个阶段拷贝数据过程中接收到的修改操作拷贝到机器 B；数据迁移完成时主控机修改被迁移子表的位置信息，整个迁移过程结束。同样，如果单机存储引擎支持快照功能，整个流程会更加容易和高效。

Bigtable 的迁移依赖于底层 GFS 提供可靠的文件存储，Bigtable 写操作的操作日志持久化到 GFS 中，且每个 tablet 由一台 Tablet Server 提供服务。当 Tablet Server 出现宕机或者负载平衡需要执行子表迁移操作时，只需要停止源 Tablet Server 对待迁移 tablet 的服务并在目的 Tablet Server 上重新加载 tablet 即可。由于 Bigtable 有 GFS 提供可靠存储，我们可以认为 Tablet Server 服务节点是无状态的。

我们在这里提出一种设计方案：将机器分成一个一个的 group，每一个子表都在某个 group 的每台机器存放一个备份，同一个时刻一个 group 中只有一台机器提供写服务，其它机器都提供读服务。将子表从 group A 迁移到 group B 其实就是将子表从 group A 中的 Master 机器迁移到 group B 中的 Master 机器，整个过程由集群的主控机来协调。下面我们考虑一下迁移过程中发生的各种异常情况：

1, 迁移的第一个阶段 group A 中 Master 宕机：group A 中某台与 Master 保持强同步的 Slave 接替 Master 对外服务，整个迁移过程失败结束；

2, 迁移的第二个阶段 group A 中 Master 宕机：group A 中某台与 Master 保持强同步的 Slave 接替 Master 对外服务，整个迁移过程失败结束；

3, 迁移过程中 group B 中 Master 宕机：整个迁移过程失败结束；

4, 拷贝数据完成后集群主控机修改子表位置信息失败：此时被迁移 tablet 在 group A 和 group B 中的数据完全一样，任意一个 group 提供服务均可；

5, 迁移完成后 group A 中 Master 宕机：group A 中某台与 Master 保持强同步的 Slave 接替 Master 对外服务，这个 Slave 可能不知道子表已经迁移的信息。子表迁移后客户端写操作需要重新建立连接，这个过程会请求集群的主控机，但是 group A 的机器可能使用老数据继续提供读服务，这就需要 Master 将子表迁移信息告知 group A 中的其它机器。

上述的机器同构的做法有一个问题：增加副本需要全部拷贝一台机器存储的数据，如果数据总量为 1TB，拷贝限速 20MB/s，拷贝时间为十几个小时，另外，子表迁移的工程实现也比较麻烦。因此，工程上多数系统静态分配好每个子表所在的机器并且不迁移，如数据库 sharding 预先分配好每一份数据所在的机器。另外一种做法是设计的时候分离静态数据和修改数据，定期合并，迁移的时候只迁移静态数据，这个思想在淘宝最近研发的 Oceanbase 系统里面有所体现。

3.5 负载均衡

负载平衡是一个研究课题，难点在于负载平衡的策略和参数调整，工程化的难度不大，和数据挖掘相关的项目有些类似，需要不断地做假设并做实验验证。

负载平衡有两种思路，一种是集群总控机根据负载情况全局调度，另一种思路是采用 DHT 方法。

第二种思路可以参考 Amazon Dynamo 的论文，DHT 算法中每个节点分配的 token 决定了数据及负载的分布。假设 DHT 环中有 S 个节点，一种比较好的 token 分配方法是将整个 Hash 空间分成 Q 等份，Q >> S，token 分配维持每个节点分配 Q/S 个 token 的特性。当节点下线时，需要将它所服务的 token 分配给其它节点，从而保持每个节点包含 Q/S 个 token 的特性；同样，当新节点上线时，也需要从集群中已有的节点获取 token 使得最终维持每个节点 Q/S 个 token 的特性。

第一种思路需要工作机通过 heartbeat 定时将读、写个数，磁盘，内存负载等信息发送给主控机，主控机根据负载计算公式计算出需要迁移的数据放入到迁移队列中等待执行。负载平衡的时候需要注意控制节奏，比如一台工作机刚上线的时候，由于负载最轻，如果主控机将大量的数据迁移到新上线的机器，由于迁移过程不能提供写服务，整个系统的对外表现性能会因为新增机器而变差。一般来说，从新机器加入到集群负载达到比较均衡的状态需要较长一段时间，比如 30 分钟到一个小时。

3.6 Chubby

Chubby 是 Google 的 Paxos 实现，Paxos 靠谱的实现不多，Chubby 毫无疑问是做的最优秀的。Chubby 通过类似文件系统接口的方式给用户暴露分布式锁服务。我们先看看应用是如何使用 Chubby 服务的。

在 GFS/Bigtable 论文中，我们至少能够看到有如下几处使用了 Chubby。

1, Master 选举。Master 宕机时，与 Master 保持强同步的 Slave 切换为 Master 继续提供服务。在这个过程中，Master 和 Slave 都定时向 Chubby 请求成为 Master 的锁，Master 锁有一个 Lease 的期限，如果 Master 正常，一定会在 Master 锁没有过期的时候申请延长锁的时间，继续提供服务。当 Master 宕机且锁的 Lease 过期时，Slave 将抢到 Master 锁切换为 Master。

2, tablet 服务。为了保证强一致性，一个 tablet 同一时刻只允许被一个 Tablet Server 加载提供服务。每个 tablet server 启动时都向 Chubby 服务获取一个锁，每当 Master 发现 tablet server 出现异常时，它也尝试获取该 Tablet server 的锁。Master 和 Tablet Server 二者只有一个节点能够获取到锁，如果锁被 Master 获取，可以确定 Tablet Server 已经宕机，此时可以将它服务的 tablet 分配给其它机器。

3, 存储 Bigtable 表格的 schema 信息。由于 Chubby 可以认为是一个一致的共享存储，并且 schema 的访问压力不大，Chubby 可以存储 schema 信息。

我们再来看看 Chubby 内部大致是如何实现的。Chubby 一般有五台机器组成一个集群，可以部署成两地三机房，这样任何一个机房停电都不影响 Chubby 服务。Chubby 内部的五台机器需要通过实现 Paxos 协议选取一个 Chubby Master 机器，其它机器是 Chubby Slave，同一时刻只有一个 Chubby Master。Chubby 相关的数据，比如锁信息，客户端的 Session 信息都需要同步到整个集群，采用半同步的做法，超过一半的机器成功就可以回复客户端。每个

Chubby Master 和 Chubby Slave 都希望成为 Chubby Master，Chubby Master 有一个 Lease 期限，如果 Chubby Master 正常，它将在 Lease 快到期的时候延长 Lease 期限，如果 Chubby Master 宕机，Chubby 集群内部将触发一次 Paxos 选举过程。每个 Chubby Slave 都希望自己成为 Chubby Master，它们类似于 Paxos 协议中的 Proposer，每个 Chubby 集群中的节点都是 Acceptor，最后可以确保只有一个和原有的 Chubby Master 保持完全同步的 Chubby Slave 被选取为新的 Chubby Master。当然，无论是 Paxos 选举还是 Session，锁信息同步，Chubby 集群内部机器故障检测都远没有这么简单，这里的实现也是笔者的揣测，如果有同学感兴趣，可以参考 Berkerly DB 中半同步（包括选举过程）的实现，这部分代码是由 Google 内部开源出来的。

3.7 分布式事务

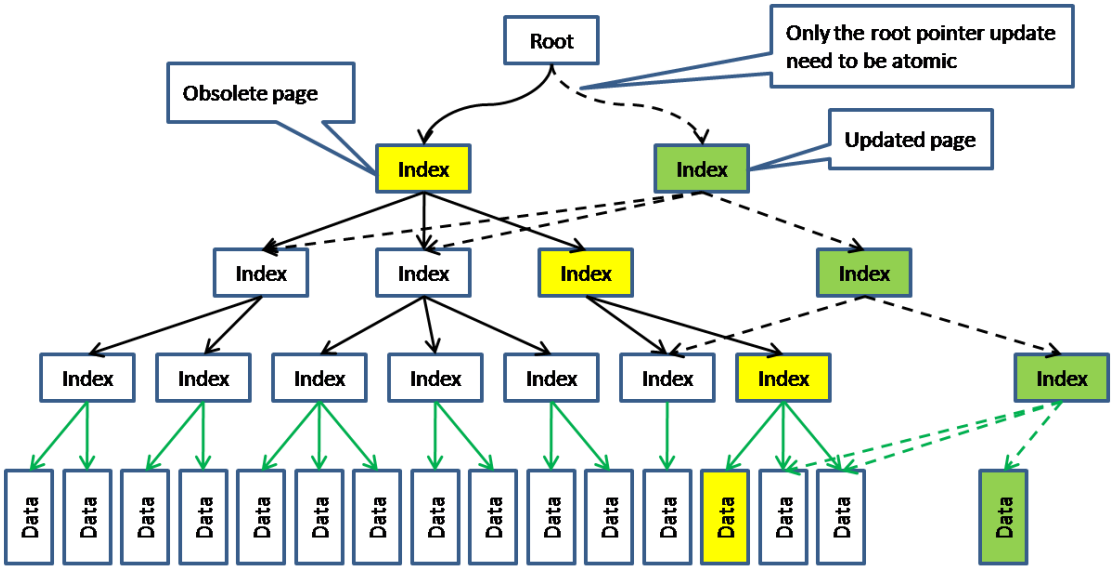
对于分布式事务，大多数情况下我们应该想的是如何回避它，两阶段锁的方法不仅效率低，而且实现特别复杂。有的时候，我们需要和业务方一起探讨如何规避分布式事务。这里我们会用到流行的概念 BASE，即基本可用，柔性状态，柔性一致和最终一致等。对一个“基本可用”系统来说，我们需要把系统中的所有功能点进行优先级的划分，比如转账业务和淘宝的收藏夹业务两者对一致性的要求肯定是不同的。柔性状态对用户来说是一个完整的系统，它的一致性是不允许有任何损失的，就是说用户支付了 10 块钱，那么他的帐户上必然是只扣掉了 10 块钱；但是对于系统内部的状态，我们可以采用一种柔性的策略，比如说系统内分布了 ABC 三个功能模块，我们允许它们在某一时刻三个模块的状态可以不一致。我们会通过业务和技术的手段，比如说异步机制或者批处理方式来保证系统通过柔性状态一致来获得可用性。

目前底层 NOSQL 存储系统实现分布式事务的只有 Google 的系统，它在 Bigtable 之上用 Java 语言开发了一个系统 Megastore，实现了两阶段锁，并通过 Chubby 来避免两阶段锁协调者宕机带来的问题。Megastore 实现目前只有简单介绍，还没有相关论文。在这个问题上，我们只能说是 Google 的同学工程能力太强了，我们开发 NOSQL 系统的时候还是走为上策。

3.8 Copy-on-write 与 Snapshot

Copy-on-write 技术在互联网公司使用比较多，这时因为大多数应用的读写比例接近 10：1，Copy-on-write 读操作不用加锁，极大地提高了读的效率，特别是现在服务器一般都有 8 个或者 16 个核。Copy-on-write 技术还带来了一个好处，那就是 Snapshot 的时候不需要停服务，而 Snapshot 功能对于分布式文件系统非常重要。

Copy-on-write 技术在树形结构中比较容易实现，假如我们实现一个支持 Copy-on-write 的 B 树，基本可以用来作为大多数管理结构的内部数据结构，比如 GFS 的 chunk 管理，文件名管理，Bigtable 中的子表管理。Copy-on-write 的示意图如下：



Copy on modify. Everyone sees his own copy of update
Finally the root pointer is swapped and everyone's view is updated
Yellow page becomes garbage over time.
File will be compacted periodically by copying to a different file.

如上图，B+树每次执行更新操作时，将从叶子到根节点路径上的所有节点先拷贝出来，并在拷贝的节点上执行修改，更新操作通过原子地切换根节点的指针来提交。由于使用了 Copy-on-write 技术，如果读取操作发生在写操作生效前将读取老的数据，否则读取新的数据，不需要加锁。Copy-on-write 技术需要利用到引用计数，当节点没有被引用，也就是引用计数减为 0 时被释放，引用计数极大地增加了数据结构的复杂度。如果需要对 B+树的某一个子树进行 Snapshot 操作，只需要增加这个子树的根节点的引用计数就可以了，后续的读取操作都将读取执行 Snapshot 操作时的数据。

对于 Hash 这样的常用数据结构，由于不支持 Copy-on-write，如果需要支持 Snapshot 操作，需要进行一定的处理。常用的做法是封装一个临时的缓冲区，执行 Snapshot 到删除 Snapshot 的过程中，写操作写入到临时缓冲区，读操作需要合并静态数据和临时缓冲区中的动态数据。

分布式系统中的 Snapshot 操作比单机操作要复杂很多，我们以 GFS 文件系统的 Snapshot 为例进行说明。为了对某个文件做 Snapshot，我们首先需要停止这个文件的写服务，接着增加这个文件的所有 chunk 的引用计数，从而以后相应的 chunk 被修改时会先对 chunk 执行一次拷贝。

对某个文件执行 Snapshot 的大致步骤如下：

1, 通过 Lease 机制收回对文件每一个 chunk 写权限，停止对文件的写服务；

2, Master 拷贝文件名等元数据生成一个新的 Snapshot 文件；

3, 对执行 Snapshot 的文件的所有 chunk 增加引用计数；

当然，工程实现时远不止这么简洁，请设计时务必考虑清楚各种异常和细节。

3.9 操作日志与 checkpoint

无论是数据库还是 NOSQL 系统，都会涉及到操作日志。这是因为磁盘和内存不一样，它是一个顺序访问设备，每一个磁盘读写操作都需要寻道，而这个寻道时间占整个读写操作的百分比很大。操作日志的原理很简单：为了利用好磁盘的顺序读写特性，将客户端的写操作先顺序写入到磁盘中，然后应用到内存中，由于内存是随机读写设备，可以很容易通过各种数据结构，比如 B+树将数据有效地组织起来。当机器宕机重启时，只需要回放操作日志就可以恢复内存状态。为了提高系统的并发能力，系统会积攒一定的操作日志再批量写入到磁盘中。

如果每次宕机恢复都需要回放所有的操作日志，效率是无法忍受的，Checkpoint 正是为了解决这个问题。系统定期将内存状态以 checkpoint 文件的形式 dump 到磁盘中，并记录 checkpoint 时刻对应的操作日志回放点，以后宕机恢复只需要先加载 checkpoint 后回放 checkpoint 对应的日志回放点以后的操作日志。由于将内存状态 dump 到磁盘需要很长的时间，而这段时间还可能有新的写操作，checkpoint 必须找一个一致的状态。如果系统的数据结构支持 Copy-on-write，事情变得相当简单：只需要增加 B+树的根节点的引用计数对整棵树做一个快照，记录此时对应的操作日志回放点即可，然后将这个快照 dump 到磁盘中。

由于执行 checkpoint 的时候需要记录此时对应的日志回放点，一个小小的技巧就是生成 checkpoint 的时候切一下操作日志的文件，使得编号为 i 的 checkpoint 文件对应编号为 i+1 的操作日志文件。

3.10列式存储与压缩

列式存储主要应用在数据仓库类的应用，列式存储将同一个列的数据存放在一起，同一个列里面按照行有序。由于同一个列的数据重复读很高，列式存储压缩时有很大的优势，比如 Google Bigtable 列式存储对网页库压缩可以达到 10~15 倍的压缩率。而且，数据仓库类应用本身就是按照列来访问的，比如查询来自湖南的用户请求个数。由于用户的读写操作使用的 SQL 语句是按照行的顺序，列式存储在实现更新操作事务性，读取某一行等普通的数据库操作相对比较麻烦。

压缩是一个专门的研究课题，没有通用的做法，需要根据数据的特点选择或者自己开发新的算法。压缩的本质就是找数据的重复或者规律，用尽量少的字节表示。压缩算法一般有一个窗口的概念，在窗口的内部找重复或者规律。存储系统在选择压缩算法的时候有两个考虑点：压缩比和效率。读操作需要先读取磁盘中的内容再解压缩，写操作需要先压缩再将压缩结果写入到磁盘，整个操作的延时包括压缩/解压缩和磁盘读写的延迟，压缩比也大，磁盘读写的数据量越大，相应地压缩/解压缩的时间也长，所以这里需要一个很好的权衡点。

压缩算法的设计和选择远超出我的能力范围，常用的压缩算法有 gzip，lzo，lzw。在 Google 的 Bigtable 系统中，设计了 Zippy 和 BMDiff 两种压缩算法。Zippy 是和 LZW 类似的。Zippy 并不像 LZW 或者 gzip 那样压缩比高，但是他处理速度非常快。BMDiff 压缩速率可达到 100MB ~ 200MB，解压缩速率达到 400MB ~ 1000MB，压缩近似度高的数据压缩比也很不错。可见压缩算法的选择不能仅仅看压缩比，还要看算法执行效率。

4 通用存储系统分类

从本质上将，通用存储系统只需要一套就可以满足互联网公司95%以上的需求，因为从语义上讲，一套系统能够表达的语义和多套系统没有区别，二者都是完备的。根据公开的资料，google也正是朝着这个方向努力的，不过这不仅要求工程师有超强的能力，还要求公司有一个很好的制度保证工程师之间不要为了因为贪功而竞争，比如发现通用系统有某处不符合需求就另起炉灶开发一个专用系统。从工程的角度上看，一个工程代码量翻番，工程的复杂度至少是原来的四倍。

权衡互联网公司业务的需求及通用系统的工程复杂度，笔者认为互联网公司大致需要如下几类存储系统：

1, 数据库及类数据库

2, 线上最终一致性系统

3, 线上弱一致性系统

4, 半线上及线下系统

数据库及类数据库系统的存在毋庸置疑，NOSQL 不可能完全替换数据库系统。关系型数据库支持的操作丰富多样，厂商，工具支持也已经规模化，开发人员也认可了 SQL 模型。这种类型的存储系统提供复杂的语义，但是扩展性稍差。类数据库系统是数据库与 NOSQL 之间的一个折衷，比如 Drizzle 通过减少一些不必要的 Mysql 通过增强扩展性。类数据库系统是根据互联网应用的特点选择数据库的特性子集加以支持，如果说数据库的出发点是尽可能支持所有 SQL 特性，那么，类数据库系统的出发点是在满足应用的前提下尽量不支持不必要的 SQL 特性。与普通 NOSQL 系统不同的是，类数据库系统一般不会有过多的动态分裂和迁移，并且支持事务。

线上最终一致性系统指提供线上服务且保证最终一致性。线上系统对用户操作的延时一般在 ms 级别，比如 1 ~ 30ms，但是对于写操作，机器宕机的时候停止 10~20s 的写服务很多时候是可以接受的。线上最终一致性系统的读操作保证 1~30ms 的延迟，且读操作只允许有秒级别的延时，对同一份数据同一时刻永远只有一个节点提供写服务，因此，写节点宕机时，可能需要停止如 10~20s 的写服务。这种类型的系统，内部实现时需要动态分裂和迁移以进行负载均衡。

线上弱一致性系统指的是 Dynamo 及其变种 Cassandra 这样的系统。我们在 2.4 节一致性模型的定义中已经将 Dynamo 这种需要解决冲突的系统归类为弱一致性系统。线上弱一致性 KV 系统对读和写操作都需要保证 1 ~ 30ms 的延迟。线上弱一致性系统对于能够解决冲突的应用非常有效，比如能够容忍时钟不一致带来的问题而采用”last write wins”的策略，或者淘宝购物车这种可以直接合并购物车中的宝贝的应用。线上弱一致性的一个特别大的好处是可以采用 P2P 的方法实现，由于消除了单点，任何一个节点出点小问题，比如代码出现 bug 都不会给整个系统带来太大的影响。对于互联网公司，**线上最终一致性系统和弱一致性系统二者实现一个就可以了，这主要取决于不同公司业务的一致性要求**。Dynamo 这样的系统还有一个优势是提供云服务，因为消除了单点，个别代码错误或者机房故障不至于影响很大，而基于类似 “last write wins”这样的冲突解决策略很少会出现问题。而强一致性系统一般需要有 Master 主控机，这是一个单点，代码出现 bug 影响很大，整个机房故障这种问题解决起来也相对要复杂很多。

半线上及线下系统指的就是以 GFS + Bigtable 为代表的存储系统，适合数据挖掘，搜索等各种类型的应用。如果 Bigtable 做的足够好，是可以解决线上最终一致性 KV 系统的问题的，也就是说，一般的互联网公司只需要这一套系统就可以解决 95%的需求。假设没有 Google

那么好的工程师和适合平台化的制度，可以将GFS + Bigtable定位为解决半线上及线下问题，这将使得问题简化很多。这套系统的特点就是通用，不过整体工程量过于复杂。这套方案是贵族化解决方案，需要分别开发文件系统和表格系统，且为了解决单点问题，需要开发Chubby锁服务，且设计时总是追求通用性和性能极限。Chubby锁服务代码非常复杂，但是出了一点小问题会使得整个集群没法服务。

5 典型存储系统工程实现

本章首先大致描述单机上存储引擎的实现，接着根据通用存储系统的分类选取一些典型系统加以介绍。

5.1 单机存储引擎

简单来说，单机存储引擎解决如下几个问题：
1, 插入一条数据；
2, 删除一条数据；
3, 更新一条已有的数据；
4, 随机读取一条数据；(Read)
5, 顺序扫描一段范围的数据；(Scan)
磁盘时顺序性设备，适合批量顺序写入，而内存是随机访问设备，适合通过各种数据结构组织起来从而提供快速的插入、删除、查找性能。单机存储引擎需要解决的问题就是利用好磁盘和内存的特性。
一般来说，互联网对单机存储引擎的需求有两种：一种应用只需要随机读取功能，没有顺序读取需求，另一种应用需要兼顾随机读取和顺序扫描需求。另外，单机层面经常需要设计通用的缓存子系统。

5.1.1 随机访问存储引擎

随机访问存储引擎不支持顺序扫描，因此，这种类型的存储引擎不需要将数据在存储引擎中连续存放。由于磁盘是顺序读取设备，可以采用Append-only的方式：添加一条数据的时候追加到文件的末尾，删除时索引做标记，修改数据时追加新数据到数据文件末尾并修改索引指向新位置。修改或者删除的data的空间不实时回收，而是通过在系统比较空闲的时候执行数据重写合并操作。索引存放在内存，可以通过B+树或者Hash的方式组织。这里有一个问题，如果机器宕机，需要将所有的数据都读取出来才能恢复内存中的全部索引，这个数据量是很大的。

索引数据也可以先写磁盘在修改内存索引数据结构，由于索引文件远小于数据文件，当机器出现故障时，只需要将全部的索引数据读取出来构造内存数据结构就可以了。

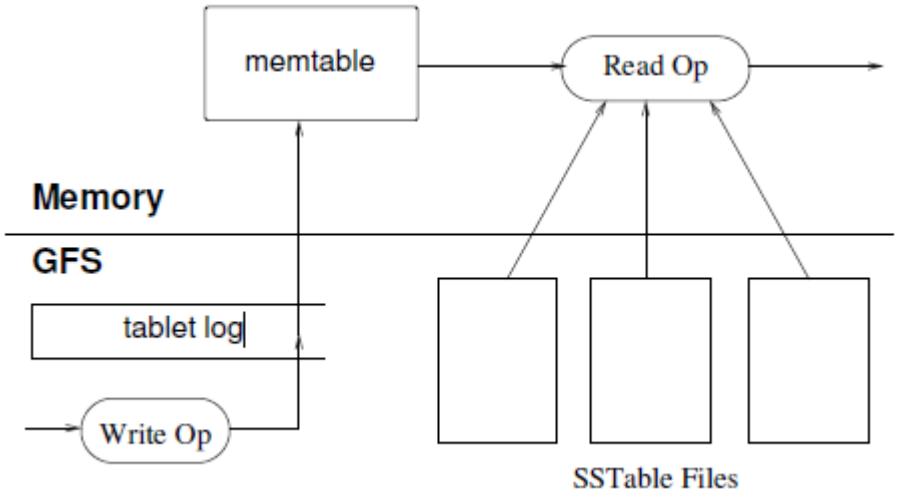
稍微麻烦点的是数据重写过程，数据重写过程相当于先对数据做一个Snapshot，然后重写这份Snapshot，回收删除和修改操作造成的文件空洞，而且，重写过程中还需要能够接受新的更新。当需要启动重写过程时，先等待以前的写操作结束，以后的写操作都写新文件（相

当于执行一次 Snapshot 操作），重写的数据直接 append 到新文件尾部，更新索引时如果发现索引最后更新时间在执行 Snapshot 之后，说明是重写过程中有新的更新，什么也不做；否则，更新索引。

由于索引全部装载在内存中，每次随机读取操作最多访问一次磁盘，而且，写操作也没有太多的冗余数据，因此，从理论上讲，这种做法对于随机访问已经最优了。

5.1.2 通用存储引擎

通用存储引擎既支持随机读取，又能很好地支持顺序扫描，这就要求数据在磁盘中连续存放，一个比较常见的做法就是 Merge-dump 存储引擎，下面以 Bigtable 为例进行说明。



数据写入时需要先写操作日志，成功后应用到内存中的 MemTable 中，写操作日志是往磁盘中的日志文件追加数据，很好地利用了磁盘设备的特性。当内存中的 MemTable 达到一定的大小，需要将 MemTable dump 到磁盘中生成 SSTable 文件。由于数据同时存在 MemTable 和可能多个 SSTable 中，读取操作需要按老到新合并 SSTable 和内存中的 MemTable 数据。数据在 SSTable 中连续存放，因此可以同时满足随机读取和顺序读取两种需求。为了防止磁盘中的 SSTable 文件过多，需要定时将多个 SSTable 通过 compaction 过程合并为一个 SSTable，从而减少后续读操作需要读取的文件个数。一般地，如果写操作比较少，我们总是能够使得对每一份数据同时只存在一个 SSTable 和一个 MemTable，也就是说，随机读取和顺序读取都只需要访问一次磁盘，这对于线上服务基本上都是成立的。

插入、删除、更新，Add 等操作在 Merge-dump 引擎中都看成一回事，除了最早生成的 SSTable 外，SSTable 中记录的只是操作，而不是最终的结果，需要等到读取（随机或者顺序）时才合并得到最终结果。Bigtable 的 Merge-dump 存储引擎对于 NOSQL 存储系统来说基本已经足够通用了，Cassandra 也采用了类似的做法，不过据说实现得不好，没有控制 SSTable 的最大个数从而可能出现 SSTable 个数特别多以至于永远都合并不成功的问题。

5.1.3 单机存储优化

软件有一个特点，就是最大限度地利用硬件资源，随着 SSD，Fusion IO 等各种技术的发展，可以考虑在单机层面上通过搭配不同类型的硬件来整体优化存储系统，在性能和价格上取得一个很好的折衷。

我所在部门的 CDN 团队做了极其出色的工作。他们在 Squid 服务器上使用 SSD + SAS + SATA 混合存储，按照访问热点进行迁移：最热的进 SSD，中等热度的放 SAS，轻热度的放 SATA，并适当考虑数据大小，最后的效果是绝大部分的读取操作走 SSD，在保证性能的前提下节省了成本。

百度在 Mysql 的针对 SSD 的优化也做了一个工作，简单来说就是将对 SSD 的随机写变成顺序写。当 mysql 需要对数据库的数据进行写操作时，它并不是直接写原来的 data file，而是把写好的 page 放在内存中，当内存中攒满了几个 page 后，它就将这些 page 组织成一个 block，将这个 block 以 append write 的方式写入到另一个 cache file 中。这一步很重要，因为它将本来是随机写入原始 data file 中的操作编程了对 cache file 的追加写。再看读数据，内存中会维护一份 page mapping，这样当 database 需要读某个 page 的数据的时候，它会写在 page mapping 中查找这个 page 是在 data file 还是在 cache file 中，然后再从相应的 file 中将数据取出来。虽然 cache file 中的数据是非常凌乱的组织，但由于 SSD 的随机读性能很好，所以这一点就不成问题。最后，cache file 在某个时刻要合并到原始的 data file 中。这一步其实会产生大量的随机写。但是，系统可以通过调度和控制，找到一个系统负荷很小的时刻来执行这个 merge 操作，比如深夜。这样，就避免了大量的随机写对系统的性能造成影响。

SSD 或者其它硬件会发展成什么样还是一个未知数，不过单机层面根据性价比选择合适的硬件并针对硬件优化也是一个不错的选择。

5.2 SQL 数据库

SQL 数据库已经很成熟了，如果是开源的数据库，比如使用最为广泛的 Mysql，我们可以在单机层面做一些工作，比如 5.1.3 提到的针对 SSD 的优化。

由于 SQL 定义的语法集合过于复杂，其中很多语法支持都影响扩展性，比如存储过程，外键，因此，要完全实现一个既支持 SQL 语法集合的，又能扩展到成百上千台机器且性能不错的分布式数据库系统至少在五年之内都是不现实的。互联网公司主要对数据库做的主要的工作还是对数据进行垂直切分和水平切分。垂直切分主要是将不同的表划分到不同的数据库（主机）之上；水平切分是将同一个表的数据按照某种条件拆分到多台数据库（主机）上。

作为存储系统开发人员，我们需要开发一套数据库中间层，它用于解析用户的 SQL 语句，并根据拆分规则转发到不同的数据库（主机），实现读写分离，合并多台数据库分库的返回结果。数据库中间层可以通过访问每个分库的主库来提供强一致性，一种可行的做法是同一个 session 内保持强一致性，也就是说，如果客户端同一个 session 内对某一个数据库分库有修改操作，后续的读取操作都转发到这个分库的主库。数据库中间层主要有几个部分：SQL 语句解析，SQL 转发，结果合并、排序、属性过滤以及数据库协议适配器。其中，SQL 语句解析相对比较复杂，数据库协议适配器相对比较啰嗦。

数据拆分后，不能很好地支持事务操作，某些跨表 join 和排序分页功能也不能高效地支持，但是给应用开发人员的接口却是 SQL 接口，应用开发人员需要仔细设计。

5.3 线上最终一致性系统

线上最终一致性系统保证最终一致性，读服务 1~30ms，正常时写服务 100ms 以内，机器出现故障时，某个数据分片的写服务可能出现 10~20s 的服务中断。为了便于介绍，将系统命名为 KO。

KO 系统由一个主控机 Master 和大量的机器 group 组成，每个 group 由 N（N 是数据的备份数）台工作机 Chunk Server，同一个 group 中同一时刻有且只有一台 Chunk Server 提供写服务，称为 CS Master，其它 Chunk Server 与 CS Master 保持同步，称为 CS Slave。一般来说，N = 3，任何一个写操作只需要写成功其中一台 CS Slave 和 CS Master 就可以返回客户端表示成功。当 CS Master 宕机时，同一个 group 中与它保持同步最快的 CS Slave 接替它成为 CS Master，并触发一个运维操作，运维人员可在合适的时间上线一台机器到 group 中。新机器加入集群以 group 为单位。KO 系统包含如下几个角色：

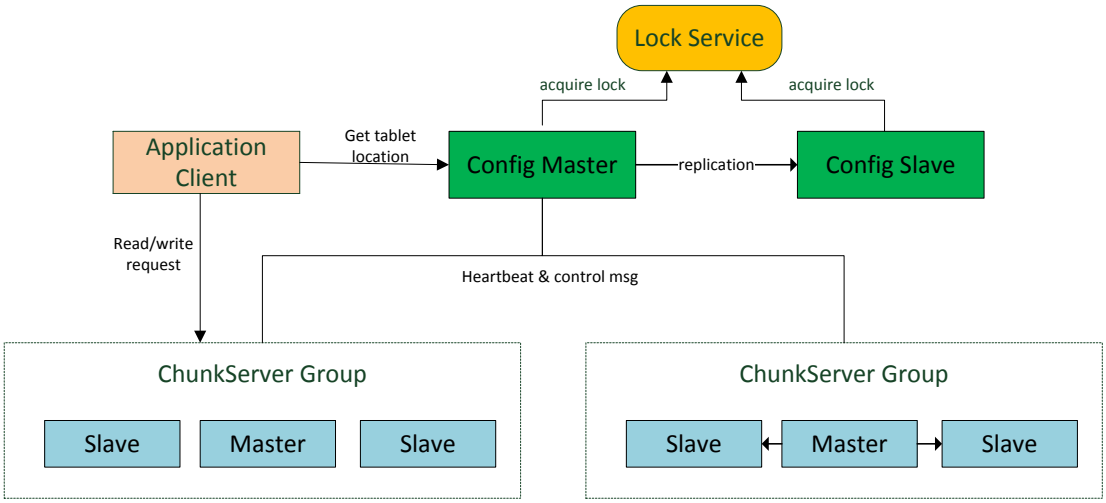
1, Master(Config Master)。集群主控机，通过 Master/Slave 强同步模式保证 HA。

2, CS Master。机器组中提供写服务的 Chunk Server。

3, CS Slave。机器组中提供 CS Master 的备份，可以提供读服务。

4, API。客户端请求 API。

整个系统大致的架构图如下：



写操作的大致流程为：

1，请求 Master 或者从客户端缓存中获取待操作行所在的 Chunk Server Master 位置；

2，往 Chunk Server Master 发送写操作；

3, Chunk Server Master 将写操作同步到同一个 group 下的 Chunk Server Slave；

4, Chunk Server Master 收到至少一个 Chunk Server Slave 的成功返回信息时写本地操作日志并应用到内存中；

5，返回客户端写操作成功或者失败原因；

如果 Chunk Server Master 返回客户端待操作行不在服务范围内，说明发生了 tablet 迁移，客户端重新请求 Master 获取 Chunk Server Master 位置信息并重试，同时更新客户端缓存。

读取操作与写操作流程类似，不同的是，可以根据读取的一致性要求选择读取同一个 group 下的 Chunk Server Master 或者 Chunk Server Slave。

在数据模型方面，提供类似 Bigtable 的 schema 支持，并保证单行操作的事务性。所有

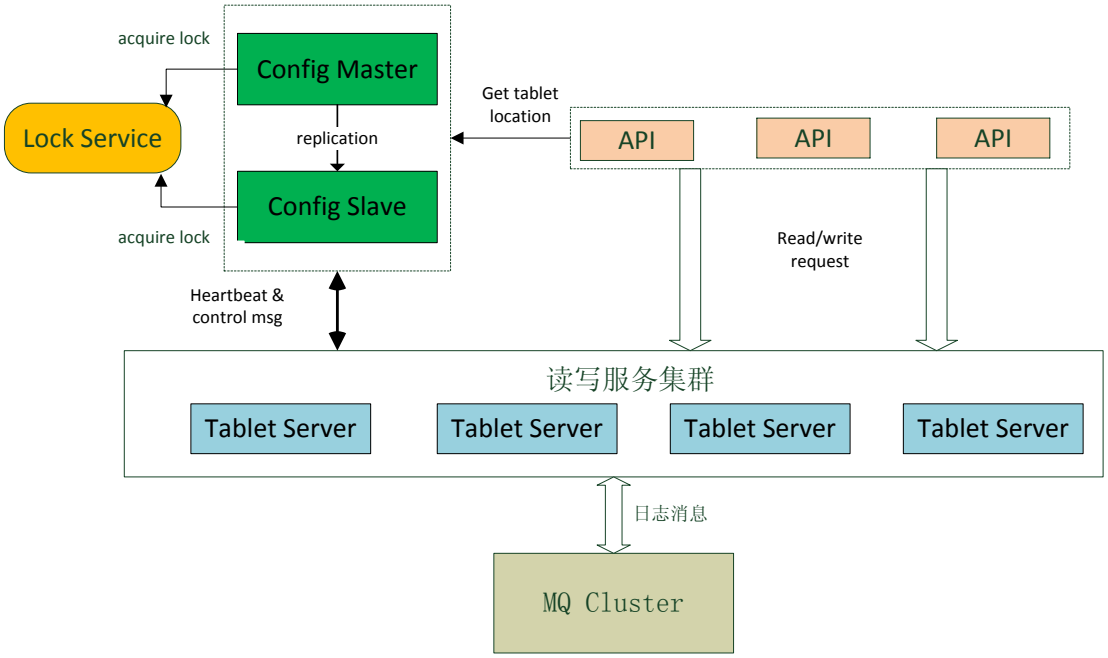
schema 的更新都在 Master 上执行，Master 更新完成后推送到 Chunk Server。客户端向 Master 查询每一行所在的 Chunk Server 位置信息时，Master 会返回 schema 的版本信息，后续客户端查询 Chunk Server 时如果发现 Chunk Server 的版本低，Chunk Server 需要向 Master 请求最新的 schema 信息。

整个系统设计的数据规模控制在 200TB 左右，集群规模在两百台左右，磁盘大致选择 6 * 600GB 的 SAS 盘，内存大致为 16GB，磁盘内存比大致为 100 ~ 200 : 1，数据切分为大小 100MB ~ 200MB 左右的子表 tablet。当子表太大时，需要大致平均地分裂为两个子表，当出现负载不均衡时，也需要 Master 指导子表迁移工作。Master 和 Chunk Server 通过定时的 heartbeat 通信，Chunk Server 将负载相关的信息汇报给 Master。分裂和迁移的做法可以参考 3.3 和 3.4 的描述。

为了同时支持随机读取和顺序读取操作，单机存储引擎选择实现 5.1.2 中提到的 Merge-dump 存储引擎。

系统工程实现时的几个比较难的问题为：Chunk Server Master 与 Chunk Server Slave 同步，子表分裂和子表迁移，schema 管理需要考虑的细节比较多，有些啰嗦。

细心的读取可能会发现，如果 group 中某一台机器宕机并且出现磁盘故障，需要在 group 中新增一台机器，假设机器中有 1TB 的数据，线上拷贝数据限速 20MB/s，那么拷贝数据时间为 1TB/20MB = 1MB/20 = 50000s，也就是十几个小时，因此需要调整架构使得宕机后数据和服务能够迁移到整个集群而不是某一台机器。可扩展的分布式系统设计的难点就在于稳定可靠的地方存储操作日志，我们可以采用类似 PNUTS 中的方法使用消息中间件存储更新数据，大致的架构如下：



如上图，数据划分为 tablet，Tablet Server 提供 tablet 读写服务，每个时刻对同一个 tablet 有一台 Tablet Server 提供写服务，称为 Tablet Server Master，其它几个副本提供保证最终一致性的读服务，写操作写入 Tablet Server Master 和消息中间件后成功返回，其它的副本通过订阅消息中间件的消息回放操作日志，提供读取服务。如果 Tablet Server Master 宕机，可以选择一个副本回放完操作日志后成为新的 Tablet Server Master。Tablet 分裂也可以通过将分裂操作日志写入消息中间件的方式保证多个副本之间的一致性。任何一台机器宕机都只会停止十几秒的部分数据的写服务，读服务不受影响，满足线上服务的需求。

5.4 线上弱一致性系统

线上弱一致性系统以 Dynamo 和 Cassandra 为代表。这类系统有两个好处：一个是机器宕机不需要停写服务，另一个是由于只需要保证弱一致性可以采用 P2P 的方法实现，大大地减少了对工程师个人能力的依赖，减少了系统整体风险。Dynamo 内部使用了很多 P2P 中的技术，这些技术都可以单独应用到其它系统的开发过程中。一个分布式系统在设计时必须想好这个系统将来如何测试，Dynamo 和 Cassandra 这样的弱一致性系统的主要难点就在于如何做系统测试。初步想到的测试工作如下：

1, 模块级别的测试需要做得比较细，比如 gossip 协议测试，DHT 分布测试，机器上下线模块测试，存储引擎测试，冲突处理测试；

2, 专门为测试提供额外的信息，比如每次写操作返回写入的机器编号，写入时刻的 timestamp 等额外信息，如果对同一个 key 多次写入不同的机器，可以在客户端模拟合并过程，将合并结果与读取到的数据进行对比；

3, 模拟一次两台机器宕机的情形，两台机器中的节点某些在 DHT 环相邻，某些不相邻；模拟网络分区的情况；

Dynamo 实现时需要使用到如下技术点：

- **DHT**

DHT 全称 Distributed Hash Table，使用一致性 Hash (consistent hashing)思想：给每台机器赋予固定数量的 token，这些 token 构成一个分布式 Hash 环。执行数据存放操作时，先计算 key 的 hash 值，然后存放到第一个包含大于或者等于该 Hash 值的 token 的节点。这种方法的好处就在于机器上下线时只会影响到在 DHT 中相邻的 token。

外部的数据可能首先传输至集群中的任意一台机器，为了找到数据所属机器，要求每台机器维护一定的集群机器信息用于定位。最直观的想法当然是每台机器分别维护它的前一台及后一台机器的信息，机器的编号可以为机器 IP 的 Hash 值，定位机器最坏情况下复杂度为 O(N)。可以采用平衡化思想来优化（如同平衡二叉树优化数组/链表），使每一台机器存储 O(logN)的集群机器信息，定位机器最坏情况下复杂度为 O(logN)。在 Dynamo 系统中，每台机器尽量维护整个集群的信息，客户端也缓存整个集群的信息，从而大多数请求都能直接发送到目标机器。

Dynamo 中通过 gossip 协议发现机器上下线信息从而更新每台机器缓存的 DHT 环。集群中某台机器上下线时，部分机器先发现，部分机器后发现，从而集群中每台机器缓存的 DHT 环处于不一致的状态。比如系统中初始有四台机器包含的 token 分别为 A, C, D, E，现在上线一台机器 token 为 B，假设 A 发现了 B 上线但是 C, D, E 没有发现，这时原本属于 B 的数据可能写入 B, C, D（假设存储三份），也可能写入 C,D,E，不过没有关系，以后 E 将发现 B，或者说是 B 发现 E，从而触发操作将错误写入到 E 的数据同步到 B，由 B 来进行数据合并，合并的过程中可能需要处理冲突。

- **Gossip 协议**

Gossip 协议用于 P2P 系统中自治的节点协调对整个集群的认识，比如集群的节点状态，负载情况。我们先看看两个节点 A 和 B 是如何交换对世界的认识的。

  - A 告诉 B 其管理的所有节点的版本（包括 Down 状态和 Up 状态的节点）
  - B 告诉 A 哪些版本他比较旧了，哪些版本他有最新的，然后把最新的那些节点发给 A（处于 Down 状态的节点由于版本没有发生更新所以不会被关注）

- A 将 B 中比较旧的节点发送给 B，同时将 B 发送来的最新节点信息做本地更新；
- B 收到 A 发来的最新节点信息后，对本地缓存的比较旧的节点做更新；

由于种子节点的存在，新节点加入可以做得比较简单。新节点加入时首先与种子节点交换世界观，从而对集群有了认识。DHT 环中原有的其它节点也会定期和种子节点交换世界观，从而发现新节点的加入。

世界不断变化，可能随时有机器下线，因此，每个节点还需要定期通过 gossip 协议同其它节点交换世界观。如果发现某个节点很长时间状态都没有更新，比如距离上次更新的时间间隔超过一定的阀值，则认为该节点已经宕机了。

- **Replication**

为了处理节点失效的情况（DHT 环中删除节点），需要对节点的数据进行 replication。思路如下：假设数据存储 N 份，DHT 定位到的数据所属节点为 K，则数据存储在节点 K, K+1, ..., K+N-1 上。如果第 K + i (0 <= i <= N-1) 台机器宕机，则往后找一台机器 K+N 临时替代。如果第 K+i 台机器重启，临时替代的机器 K+N 能够通过 gossip 协议发现，它会将这些临时数据归还 N+i，这个过程在 Dynamo 中叫做 Hinted handoff。机器 K+i 宕机的这段时间内，所有的读写均落入到机器[K, K+i-1]和[K+i+1, K+N]中。如果机器 K+i 永久失效，机器 K+N 需要进行数据同步操作。一般来说，从机器 K+i 宕机开始到被认定为永久失效的时间不会太长，积累的写操作也不会太多，可以利用 Merkle Tree 对机器的数据文件进行快速同步。

由于数据需要存储 N 份，机器宕机会影响到 DHT 环中后面的 N 个 token 所在的机器。

- **Quorum NWR**

NWR 是 Dynamo 中的一个亮点，其中 N 表示复制的备份数，R 指成功读操作的最少节点数，W 指成功写操作的最少节点数。只要满足 W + R > N，就可以保证在存在不超过一台机器故障的时候，至少能够读到一份有效的数据。如果应用重视读效率，可以设置 W = N, R = 1；如果应用需要在读/写之间权衡，一般可设置 W = 2, R = 2，N = 3，当然也可以选择 W = 1, R = 1, N = 3, 如果丢失最后的一些更新也不会有影响的话。

NWR 看似很完美，其实不对。在 Dynamo 这样的 P2P 集群中，由于每个节点对集群的认识不同，可能出现同一个 key 被多台机器同时操作的情况，在多台机器上执行的顺序是无法保证的，需要依赖基于 vector clock 的冲突合并方法解决冲突，默认的解决方法是"last write wins"，而这时依赖于两台机器之间的时钟同步的。所以，**Dynamo 中即使 N + W > R，这里所谓的"最终一致性"还是依赖冲突合并算法，最后读取的结果和客户期望的结果可能不一致**。关于这一点，我们在 2.4 中已经将最终一致性模型与 Dynamo 中的一致性模型区分开来了。这个不一致问题需要注意，因为影响到了应用程序的设计和对整个系统的测试工作。

- **读写流程**

客户端的读/写请求首先传输到缓存的一台机器，根据预先配置的 N、W 和 R 值，对于写请求，根据 DHT 算法计算出数据所属的节点后直接写入后续的 N 个节点，等到 W 个节点返回成功时返回客户端，如果写请求失败将加入 retry_list 不断重试。如果某台机器发生了临时性异常，将数据写入后续的备用机器并在备用机器中记录临时异常的机器信息。对于读请求，根据 DHT 算法计算出数据所属节点后根据负载策略选择 R 个节点，从中读取 R 份数据，如果数据一致，直接返回客户端；如果数据不一致，采用 vector clock 的方法解决冲突。Dynamo 系统默认的策略是选择最新的数据，当然用户也可以自定义冲突处理方法。读取过程中如果发现副本中有一些数据版本太旧，Dynamo 内部异步发起一次 read-repair 操作，更新过期的数据。

- **异常处理**

Dynamo 中把异常分为两种类型，临时性的异常和永久性异常。有一些异常是临时性的，比如机器假死，其它异常如硬盘报修或机器报废等由于其持续时间太长，称之为永久性的。

Hinted handoff：在 Dynamo 设计中，一份数据被写到 K, K+1, ... K+N-1 这 N 台机器上，如果机器 K+i (0 <= i <= N-1)宕机，原本写入该机器的数据转移到机器 K+N，如果在指定的时间 T 内 N+i 重新提供服务，机器 K+N 将通过 gossip 协议发现，并将启动传输任务将暂存的数据发送给机器 N+i；

Merkle Tree 同步：如果超过了时间 T 机器 N+i 还是处于宕机状态，这种异常被认为是永久性的，这时需要借助 Merkle Tree 机制从其它副本进行数据同步。Merkle Tree 同步的原理很简单，每个非叶子节点对应多个文件，为其所有子节点值组合以后的 Hash 值，叶子节点对应单个数据文件，为文件内容的 Hash 值。这样，任何一个数据文件不匹配都将导致从该文件对应的叶子节点到根节点的所有节点值不同。每台机器对每一段范围的数据维护一颗 Merkle Tree，机器同步时首先传输 Merkle Tree 信息，并且只需要同步从根到叶子的所有节点值均不相同的文件。

Read repair：假设 N=3, W=2, R=2，机器 K 宕机，可能有部分写操作已经返回客户端成功了但是没有完全同步到所有的副本，如果机器 K 出现永久性异常，比如磁盘故障，三个副本之间的数据一直都不一致。客户端的读取操作如果发现了某些副本版本太老，则启动异步的 Read repair 任务，更新过期的数据从而使得副本之间保持一致。

- **负载平衡**

Dynamo 的负载平衡取决于如何给每台机器分配虚拟节点号，即 token。由于集群环境的异构性，每台物理机器包含多个虚拟节点。一般有如下两种分配节点号的方法：

1. 随机分配。每台物理节点加入时根据其配置情况随机分配 S 个 Token(节点号)。这种方法的负载平衡效果还是不错的，因为自然界的数据大致是比较随机的，虽然可能出现某段范围的数据特别多的情况（如 baidu, sina 等域名下的网页特别多），但是只要切分足够细，即 S 足够大，负载还是比较均衡的。这个方法的问题是可控性较差，新节点加入/离开系统时，集群中的原有节点都需要扫描所有的数据从而找出属于新节点的数据，Merkle Tree 也需要全部更新；另外，增量归档/备份变得几乎不可能。

2. 数据范围等分+随机分配。为了解决方法 1 的问题，首先将数据的 Hash 空间等分为 Q = N * S 份 (N=机器个数，S=每台机器的虚拟节点数)，然后每台机器随机选择 S 个分割点作为 Token。和方法 1 一样，这种方法的负载也比较均衡，且每台机器都可以对属于每个范围的数据维护一个逻辑上的 Merkle Tree，新节点加入/离开时只需扫描部分数据进行同步，并更新这部分数据对应的逻辑 Merkle Tree，增量归档也变得简单。

另外，Dynamo 对单机的前后台任务资源分配也做了一些工作。Dynamo 中同步操作、写操作重试等后台任务较多，为了不影响正常的读写服务，需要对后台任务能够使用的资源做出限制。Dynamo 中维护一个资源授权系统。该系统将整个机器的资源切分成多个片，监控 60s 内的磁盘读写响应时间，事务超时时间及锁冲突情况，根据监控信息算出机器负载从而动态调整分配给后台任务的资源片个数。

- **单机存储引擎**

Dynamo 设计支持可插拔的存储引擎，比如 Berkerly db, Mysql Innodb 等。Cassandra 中实现了一个 5.1.2 中提到的 Merge-dump 存储引擎，对于需要同时支持随机读取和顺序读取，

不仅仅通用而且效率高。

5.5 半线上及线下系统

半线上及线下系统指 GFS + Bigtable，开源的实现有 HDFS + HBase/Hypertable，不过开源的实现远没有达到 GFS + Bigtable 应有的高度。这里之所以将 GFS + Bigtable 划分为半线上及线下系统，不是因为这套解决方案不能支持线上应用，而是因为这套应用在并发量，机器异常处理等都做得很极限，如果支持线上的低延迟，永远不停读服务，整个系统过于复杂。我们姑且认为 GFS + Bigtable 这套服务能设计成支持半线上及线下应用，如增量建网页库，已经是非常大的挑战了。

5.5.1 两层结构

GFS 和 Bigtable 采用两层结构，GFS 关注文件存储文件，虽然对外提供文件接口，但是保证文件可靠性，Bigtable 关注用户接口问题，在 GFS 之上提供支持简单 schema 的 NOSQL 访问接口。

GFS 和 Bigtable 两层结构非常巧妙地解决了一致性的问题，底层的 GFS 提供弱一致性，写操作可能出现重复记录，Bigtable 通过一种类似 B+树的方式索引写入到 GFS 中的记录，很好地解决了一致性问题。Bigtable 中同一个 tablet 同一时刻只能被同一个 Tablet Server 工作机服务，不过由于底层可靠的 GFS 存储，机器宕机时只需要在其它机器中将 GFS 中持久化的内容加载到内存中即可，大大地减少了宕机恢复需要的时间。GFS 相当于是 Bigtable 的可靠的共享存储，由于 GFS 只需要提供文件接口，所以内存操作简单，出现代码错误的可能性很小，并且，GFS 可以提供类似 Snapshot 的功能；Bigtable 内部逻辑和内存操作非常复杂，有了 GFS 的 Snapshot 功能，Bigtable 可以用来备份数据，从而 Bigtable 出现 bug 时，可以修改 bug 后回滚回去重新测试。

前面提到的线上最终一致性系统采用单层结构的设计，把分裂和迁移看成是一种不是特别常见的情况，所以不是特别注重分裂和迁移的效率，且为了解决一致性引入了 Chunk Server group 的概念。而 GFS + Bigtable 的设计中，所有的工作机，即 Chunk Server 和 Tablet Server 都是完全对等的，一台 Tablet Server 机器宕机后，整个集群中的所有的机器都可以加载这个机器上服务的子表，扩展性基本做到了极限。集群的规模越大，GFS + Bigtable 的解决方案越有优势，很适合平台化，唯一的缺点就是过于复杂。

采用两层设计，底层的 GFS 可以简化很多，只需要考虑批量的读写，提供简单的 Append 模型。上层的 Bigtable 负责将写请求聚合成大块，随机读取操作的缓存等。

GFS + Bigtable 解决方案最适合的场景是半线上及线下应用，比如搜索类及数据仓库类应用，系统中的磁盘和内存比大致为 256 ~ 1000 : 1，采用廉价的 SATA 盘作为底层存储，一般单机 12 * 1T SATA 盘配置。由于系统在软件层面具有无可匹敌的容错性，可以极大地降低运维成本，节省服务器开销。

5.5.2 GFS

整个 GFS 的架构如下，

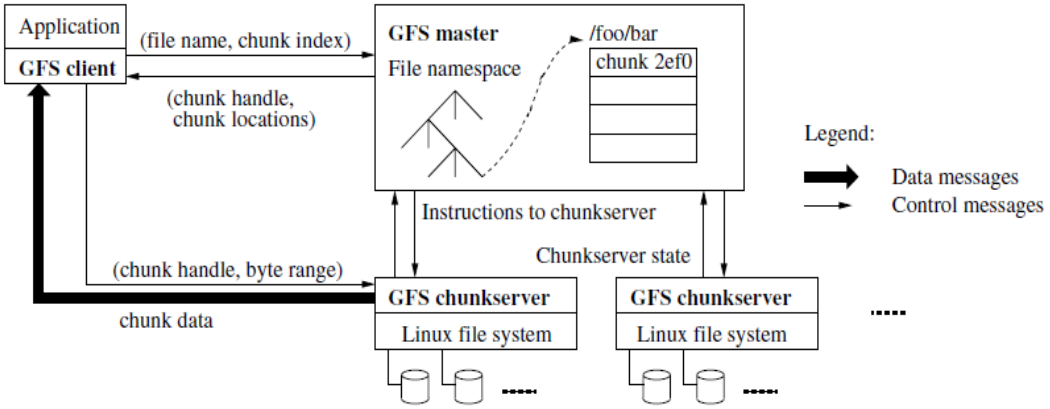


Figure 1: GFS Architecture

如上图，GFS 系统包含三个模块，API，Master，ChunkServer。为了对 GFS 系统进行运维操作，还需要一个 utility 模块提供实用工具。用户通过 API 模块向 Master 发请求或者从 API 缓存中获取 Chunk Server 信息，以后直接与 Chunk Server 打交道，将数据追加到 Chunk Server 机器中。

GFS 提供两种操作模型，Append 追加模型与 Overwrite 模型。由于 GFS 的用户可以认为是 MapReduce 计算系统和 Bigtable 系统，而这两个系统都只需要用到 Append 模型，并通过内部建索引来处理 GFS 的重复记录。

作为分布式文件系统，GFS 最难的问题就是一致性模型，也就是 GFS 的 Append 操作实现。对同一个 chunk，在 GFS 中通过 Lease 机制来保证同一时刻只能有一台机器提供追加写服务，这台 Chunk Server 称为 Primary Chunk Server，用户的数据先发送到 Primary Chunk Server，由它确定写入顺序，然后通过 pipeline 的方式同步到其它的 Chunk Server 副本，等到所有副本都写成功时返回成功，否则，客户端可能重试写操作，从而多次写入同一个记录，即重复记录。如果写入的过程中 Primary Chunk Server 出现异常，chunk 上未完成的写操作全部失败，等到 Lease 过期后，Master 将写权限授给其它服务该 chunk 的 Chunk Server，客户端也将重试写操作。

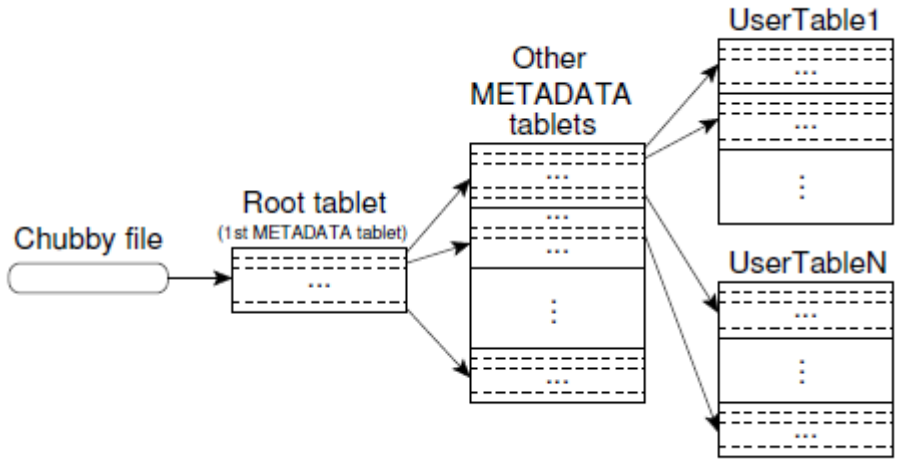
GFS Master 管理的数据包括文件元数据，文件到 chunk 的映射，chunk 元数据，如 chunk 版本号、chunk 所在的 chunk server 编号，chunk server 数据。GFS 的设计中，每个 chunk 大小为 64MB，因此存储 1PB 数据的 chunk 个数为 1024 * 1024 * 1024 * 3 / 64 = 48MB，即接近 5000 万的 chunk 数，在 Bigtable 的设计中，每个子表的大小为 100MB ~ 200MB，而每个子表至少占用一个 GFS 的文件，因此存储 1PB 数据需要的文件个数为 1024 * 1024 * 1024 / 100 = 10MB，也是千万级别。所以，GFS Master 元数据管理数据结构需要高效地支持千万级别的数据插入、查找、删除操作，且必须支持 Snapshot 功能，比较好的数据结构就是支持 Copy-on-write 的 B+树。GFS Master 需要实现 replication 将操作强同步到 Slave 中，从而宕机后可以很快地恢复。另外，GFS Master 还需要能够定期将内存状态生成一份 checkpoint 文件存储到磁盘中，从而减少宕机恢复重放的日志量。

GFS Master 还有一些全局性功能模块，比如 chunk 复制，chunk rebalance，garbage

collection, lease 管理，heartbeat 等等，另外，为了支持大规模集群的运维，可以支持从机器资源池选取机器，发送二进制程序并启动 chunk server 的功能。总之，GFS Master 的功能不仅复杂，而且啰嗦。

5.5.3 Bigtable

整个 Bigtable 的结构如下：



如上图，Bigtable 其实就是在 GFS 之上增加一层索引层，并对外提供带有 schema 的 NOSQL 用户接口。Bigtable 有三种类型的表：用户表格存储用户实际数据，METADATA 表格存储用户表格的元数据，如位置信息，SSTable 及操作日志文件编号，日志回放点等，Root table 表格存储 METADATA 表格的元数据。Bigtable 系统将每个表切分为大小在 100MB ~ 200MB 之间的子表 tablet。

整个系统中有三种角色：Bigtable Master，Tablet Server 和 API。其中，Bigtable Master 负责管理 Tablet Server，并提供一些全局性服务，比如负载均衡，用户和 schema 管理，Tablet Server 宕机恢复等；Tablet Server 是工作机，将每一个子表加载到内存中对外提供服务，它是内存操作最为复杂的模块，出现的 bug 也最多；API 用于找到子表所在的 Tablet Server 并请求服务。在 Bigtable 系统中，用户表在进行某些操作，比如子表分裂的时候需要修改 METADATA 表，METADATA 表的某些操作需要修改 Root 表，因此，Tablet Server 也需要使用 API 模块，它以动态库的方式链接到 Tablet Server 可执行程序中。

第一个需要解决的问题还是一致性。Bigtable 系统保证强一致性，同一个时刻同一个 tablet 只能被一台 Tablet Server 服务，也就是说，Master 将 tablet 分配给某个 Tablet Server 服务时需要确认没有其它的 tablet 正在服务这个 tablet。可以通过 Lease 的机制实现，每个 Tablet Server 有 Lease 才能提供服务，当 Lease 快要到期时，Tablet Server 会找 Master 重新请求延长 Lease 期限，如果 Master 发现 Tablet Server 的 Lease 已经过期了，可以安全地将它服务的 tablet 分配给其它 Tablet Server，同样，Tablet Server 也必须自觉，当它发现自己没有 Lease 时主动下线，不要影响全局一致性。

第二个需要解决的是宕机恢复问题。Tablet Server 执行一个写操作是，需要先写操作日志然后应用到 MemTable，当 MemTable 中的数据达到一定大小或者超过一定时间会写成 SSTable 存储到 GFS 文件系统中。这里我们看到，如果 Tablet Server 宕机，需要将原来服务的 tablet 迁移到其它机器，而部分内存中的数据没有序列化成 SSTable，加载时需要回放操

作日志的数据。为了更好地利用 GFS 文件系统，同一个 Tablet Server 的操作日志写到了同一个 GFS 文件。由于多个 tablet 的日志混在一起，需要对日志文件排序从而每个 tablet 加载时可以连续读取自己需要的操作日志。Master 需要有一个模块专门用于调度 Tablet Server 对操作日志文件进行排序。

第三个需要解决的问题是子表分裂和迁移问题。由于同一个 tablet 同一时刻只被一个 Tablet Server 服务，子表分裂简化很多。子表分裂之前先将 MemTable 中的数据序列化到 SSTable 中，然后通过 API 往表格的 Metadata 中加入一行表示新生成的 tablet：如果是用户表格，需要修改 METADATA 表格；如果是 METADATA 表格，需要修改 Root 表格。子表迁移也需要先将 MemTable 中的数据序列化到 SSTable，然后停止服务，Master 会把它分配给其它 Tablet Server 加载，因为此时 MemTable 中的数据都持久化到 SSTable 中，不需要回放日志。

第四个要解决的问题是存储引擎问题。采用 5.1.2 节中的 Merge-dump 存储引擎，Merge-dump 存储引擎每次操作都是大块读写，非常适合 GFS。

第五个需要解决的就是缓存和内存管理问题。由于 Bigtable 的内存操作非常复杂，要求每个模块都从全局统一的内存池中申请，从而管理整个系统对内存的使用。如果是随机读操作，Tablet Server 还需要进行缓存。和普通的文件系统的 page cache 原理类似，需要缓存从 GFS 中读取的 SSTable，由于 SSTable 是以块（大小一般为 64KB）为单位，所以，有一个块缓存；另外，还需要对每一个 SSTable 文件缓存随机读操作的读取结果。

第六个需要考虑的问题是压缩和解压缩。Bigtable 支持的应用比如网页库，数据仓库的数据重复读很高，且由于 Bigtable 存储历史版本数据，Google 里面使用了 Zippy 和 BMDiff 压缩算法，两个算法的特点都是压缩和解压缩速度快。

6 通用计算系统分类

互联网公司大致需要如下几类通用计算系统：

1, MapReduce Offline；

2, Online 计算；

线下计算，挖掘类应用 MapReduce 基本上可以包打天下了。MapReduce 的批处理能力、可扩展性以及容错能力极大地迎合了线下计算应用的需求，MapReduce 模型本身简单高效，使得普通的工程师也能写出运行在成百上千台集群的分布式程序。

Online 实时计算与线下批处理有本质的区别，线下批处理计算处理静态数据，发展出了通用的 MapReduce 模型，但是线上实时计算一直都没有统一的解决方案。后续将举一些例子说明常见的线上计算场景，如下：

1, 流式计算；

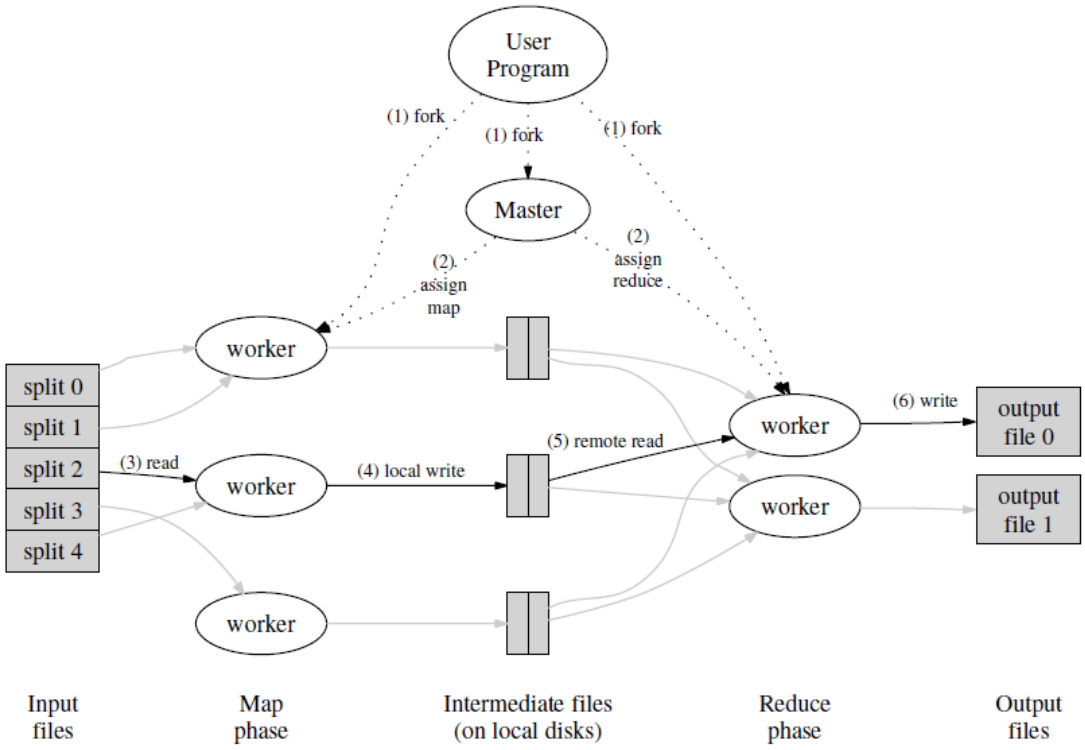
2, 并行数据库的 SQL 查询；

3, 数据仓库复杂查询；

7 典型计算系统工程实现

7.1 MapReduce Offline

MapReduce 的架构如下：



MapReduce 执行顺序如下：

1, 首先从用户提交的程序 fork 出 Master 进程，Master 进程启动后将切分任务并根据输入文件所在的位置和集群信息选择机器 fork 出 Map 或者 Reduce Worker；用户提交的程序可以根据不同的命令行参数执行不同的行为；

2, Master 将切分好的任务分配给 Map Worker 和 Reduce Worker 执行，任务切分和任务分配可以并行执行；

3, Map Worker 执行 Map 任务：读取相应的输入文件，根据指定的输入格式不断地读取<key, value>对并对每一个<key, value>对执行用户自定义的 Map 函数；

4, Map Worker 执行用户定义的 Map 函数，不断地往本地内存缓冲区输出中间<key, value>对结果，等到缓冲区超过一定大小时写入到本地磁盘中。由于 Map 操作的输出结果将根据 partition 函数分配给 R 个 Reduce Worker，所以 Map Worker 的中间结果组织成 R 份。

5, Map 任务执行完成时，Map Worker 通过 heartbeat 定期向 Master 汇报，从而 Master 通知 Reduce Worker。Reduce Worker 向 Map Worker 请求传输生成的中间结果数据。这个过程称为 Shuffle。当 Reduce 获取完所有的 Map 任务生成的中间结果时，需要进行排序操作。如果数据量太大，需要执行外排；

6, Reduce Worker 执行 Reduce 任务：对中间结果的每一个相同的 key 及 value 集合，执行用户自定义的 Reduce 函数。Reduce 函数的输出结果被写入到最终的输出结果，通常是 GFS 或者 Bigtable；

MapReduce 实现时有如下几个关键点：
1, 输入任务划分：MapReduce 的输入和输出一般是 GFS 或者 Bigtable，如果输入为 GFS，可以按照 chunk 来切分任务；如果输入为 Bigtable，可以按照 tablet 来切分任务；
2, Master 状态机：Master 状态机管理任务的状态转化，Map 或者 Reduce Worker 的状态转化，且两个状态机有关联，设计时需要画出状态转化图；
3, 用户代码和框架代码的结合：例如任务划分操作，框架应该需要支持用户自定义任务划分方法，且有默认的任务划分实现，即 GFS 文件和 Bigtable 表格任务划分实现；
4, Reduce Worker 往 Map Worker 获取中间结果的压力应该分散，不能出现某些 Map Worker 压力太大的情况；另外，Map Worker 出现宕机故障时，可能有部分 Map 任务的中间结果已经发送给 Reduce Worker，部分没有发送，由于以后 Map Worker 上的任务将被重做，所以 Reduce Worker 需要支持回滚某些 Map 任务生成的中间结果；
5, 备份任务的支持：大集群环境中，机器的处理能力相差可能非常大，备份任务对整个作业的总体执行时间有很大的优化；
6, 本地化：尽量将任务分配给离输入文件最近的 Map Worker，如同一台机器或者同一个机架；

7.2 Online 计算

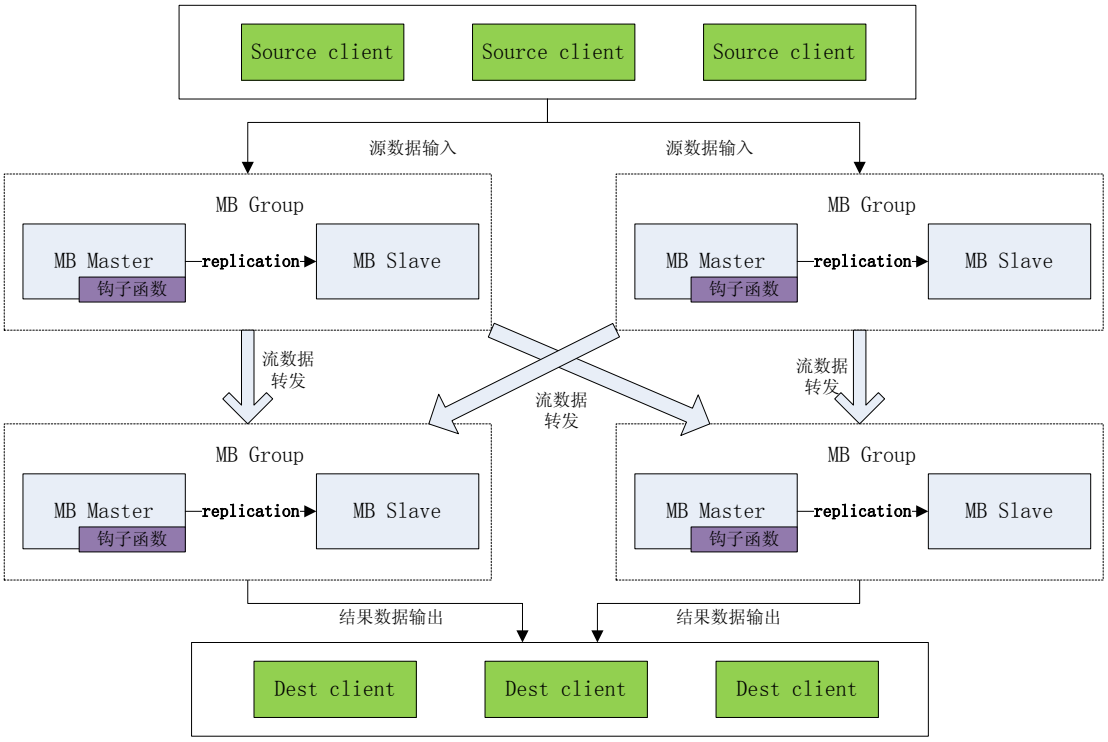
MapReduce 解决了大部分线下批处理计算的问题，SQL 语句基本能够表达所有的计算问题，但是 SQL 语句的某些操作执行效率地下，线上计算系统一般选择性地支持部分 SQL 操作，比如 limit，order by，etc。

7.2.1 流式计算

流式计算，英文名称为 Stream Processing，解决在线聚合（Online Aggregation），在线过滤（Online Filter）等问题，流式计算同时具有存储系统和计算系统的特点，经常应用在一些类似反作弊，交易异常监控等场景。流式计算的操作算子和时间相关，处理最近一段时间窗口内的数据。

如果不考虑机器故障，在线聚合和在线过滤的实现没有什么特别困难之处。如果考虑机器故障，问题变得复杂。上游的机器出现故障时，下游有两种选择：第一种选择是等待上游恢复服务，保证数据一致性；第二种选择是继续处理，优先保证可用性，等到上游恢复后再修复错误的计算结果。

流式计算系统架构如下：



如上图，源数据写入到 Message Broker (MB)处理节点，MB 处理节点通过 Master/Slave 的方式保证可靠性。MB 处理节点内部运行用户定义的钩子函数对输入流进行处理，处理完后根据一定的规则转发给下游的 MB 节点继续处理。典型钩子函数包括：

1，聚合函数：计算最近一段时间窗口内数据的聚合指，如 max, min, avg, sum, count 等；

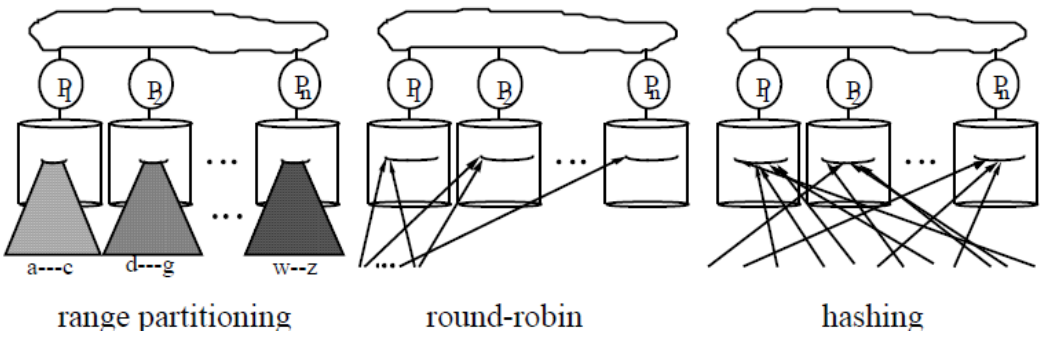
2，过滤函数：过滤最近一段时间窗口内满足某些特性的数据，如过滤 1 秒钟内重复的点击；

消息队列是一种保证数据可靠有序传输的基础设施，典型的实现有 Active MQ，Mule MQ 等。如果在消息传输过程中注入 Stream Processing 算子，可以实现流式计算。

7.2.2 并行数据库的 SQL 查询

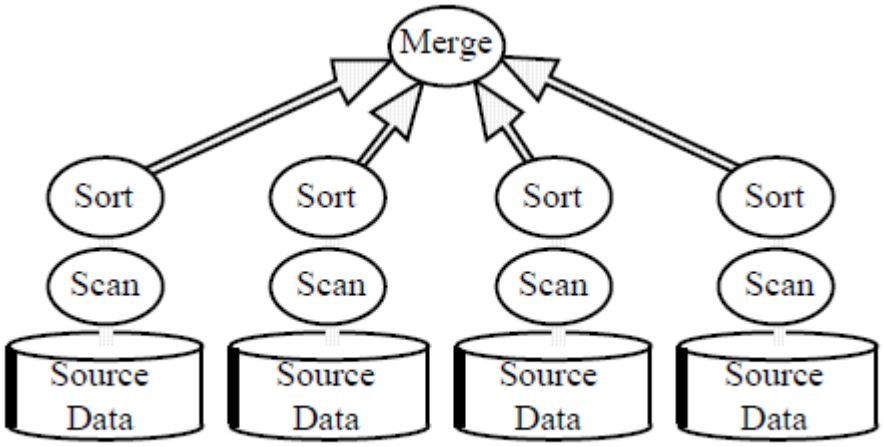
并行数据库与 NOSQL 系统区别主要在于关注点不同，并行数据库需要完全支持 SQL，NOSQL 更加注重扩展性和异常情况处理。并行数据库中 limit, order by, group by, join 等操作的实现对 NOSQL 系统设计具有借鉴作用。

常见的数据分布方式有三种：



1, Range partitioning：按照范围划分数据；
2, Round-robin：将第 i 个元组分配给 i % N 节点；
3, Hashing：根据 hash 函数计算结果将每个元组分配给相应的节点；

Merge 操作符：limit, order by, group by, join 都可以通过 Merge 操作符实现，在系统中增加一个合并节点，发送命令给各个数据分片请求相应的数据，每个数据节点扫描数据，排序后回复合并节点，由合并节点汇总数据并执行 limit, order by, group by, join 操作。这个过程相当于执行一个 Reduce 任务个数为 1 的 MapReduce 作业，不考虑机器出现故障，也不考虑数据分布不均而启动备份任务。



Split 操作符：相当于 MapReduce 中的 partition 函数。由于 Merge 节点处理的数据可能特别大，所以可以通过 Split 操作符将数据分散到多个 Merge 节点，每个节点合并数据并执行相应的 group by, join 操作。比如执行 *"select * from A, B where A.x = B.y"*，可以根据 A.x 的 hash 值将数据节点扫描到的数据分散到不同的合并节点，每个合并节点执行 Join 操作。

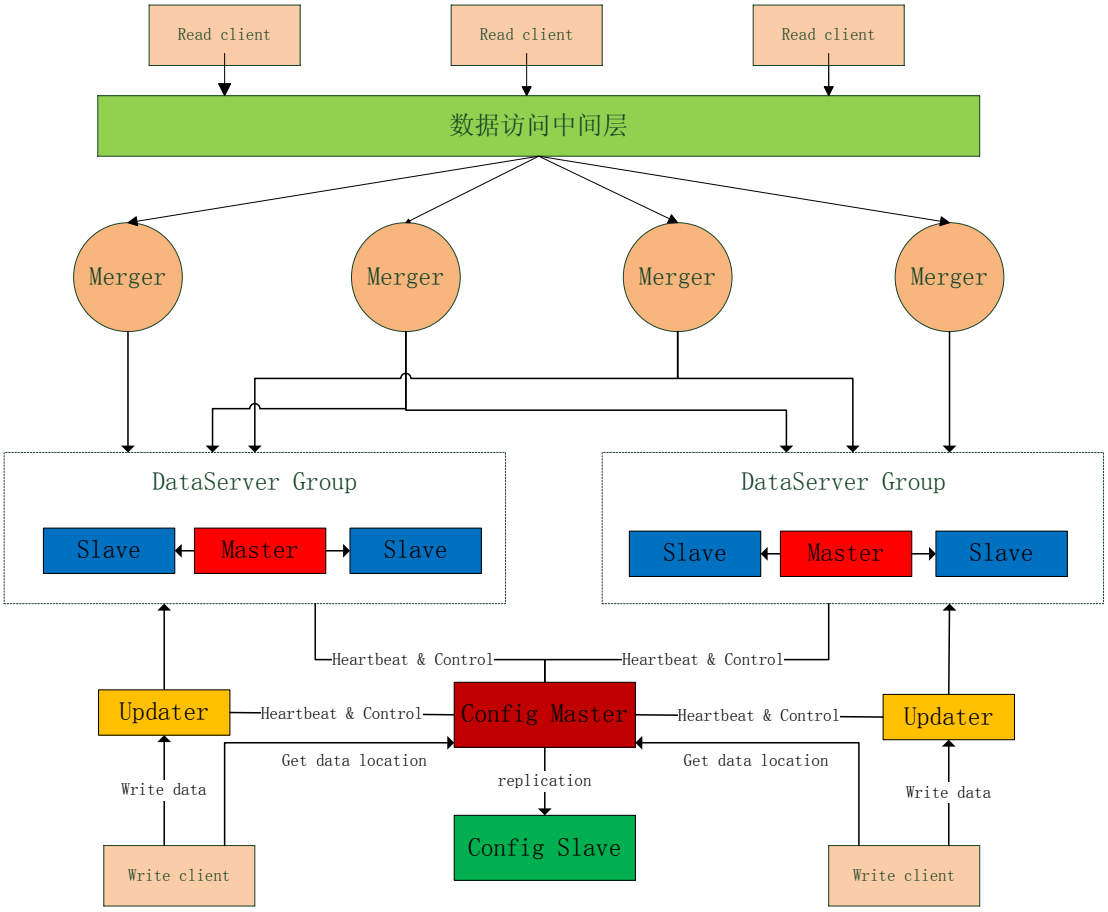
并行数据库的 SQL 查询和 MapReduce 计算有些类似，可以认为 MapReduce 模型是一种更高层次的抽象。由于考虑问题的角度不同，并行数据库处理的 SQL 查询执行时间通常很短，出现异常时整个操作重做即可，不需要像 MapReduce 实现那样引入一个 Master 节点管理计算节点，监控计算节点故障，启动备份任务等。

7.2.3 数据仓库复杂查询

数据仓库线上查询模型需要支持 order by, limit, group by，计算模型和并行数据库类似。另外，它还有几个特点：
1, 使用列式存储：列式存储符合数据仓库的按列访问特性，且增大了数据压缩比；
2, 索引：索引有两种形式：一种为单机层面的索引，另一种为分布式层面的索引。单机层面索引指的是在单机存储引擎之上增加一个索引层，索引和数据绑定，这样做的优点是索引维护成本较低，缺点是执行按索引访问操作需要访问所有的数据分片；分布式层面的索引指

的是建立一张全局的索引表，索引和数据相互独立，这样做的优点是可以根据索引直接定位到主键，缺点是索引维护成本较高。

对于给定主键或者索引列值的查询，直接将请求发送到相应的数据节点；否则，将请求发送到所有的数据节点。与并行数据库类似，由合并节点来生成最终结果。数据仓库存储子系统处理机器故障问题，可以采用 5.4 中的线上最终一致性系统实现。大致的架构如下：



如上图，通过 Updater 节点将数据写入数据节点，数据节点按照 Data Server Group 的形式组织，通过 Master/Slave 备份来保证可靠性，同一个 Data Server Group 中 Master 出现故障后由 Slave 接替其继续提供服务，保证可用性。客户端的查询操作在 Merger 节点上执行，它合并相应 Data Server Group 中的数据分片并进行 limit, order by, group by 等操作。当出现负载不均衡时，Config Master 将指导数据分片从负载高的 Data Server Group 迁移到负载低的 Data Server Group。

8 应用

本章讲述笔者对于一些典型应用的存储问题的理解，这里必须声明：任何一个应用涉及的知识都远超过笔者的能力范畴，后续的内容只是阐述个人很肤浅的理解，漏洞很多，请谅解。

8.1 电子商务类

阿里巴巴引领着电子商务的方向。以淘宝为例，淘宝面临的存储相关问题包括卖家商品，交易信息，用户信息，用户评价，用户收藏，购物车，图片等等，并且淘宝累积存储了不同业务系统收集的海量业务数据，比如访问点击、交易过程、商品类目属性以及呼叫中心客服内容等。

淘宝大多数存储系统的特点是：数据量大，记录条数特别多，单点记录不大，读写比例高，且可能要求事务。由于访问量特别大，以前采用 Oracle + 小型机的解决方案，对于不需要事务的需求，可以通过 Mysql sharding 的方式实现。我们正在做的 Oceanbase 系统巧妙地利用读写比例大且单条记录一般比较小的特点，将动态更新的数据放在单机内存中并通过强同步保证可靠性及可用性，动态数据定期与静态数据合并。

淘宝的小文件存储系统 TFS 已经开源了，目前主要是用来存储海量图片文件。淘宝 TFS 处理百亿级别的图片存储，数据量 PB 级别，这个问题属于 5.4 中提到的线上最终一致性系统的范畴，不过通用系统的解决方案过于复杂，性价比不高。于是，淘宝天才的工程师们利用图片应用的特点设计了通用的小文件存储系统 TFS(TFS 开源)。图片存储系统的特点主要有两个：

1, 用户一次性准备好文件所有数据并提交到文件系统，每个文件打开后一次性写入所有数据并关闭；

2, 用户不关心文件的名字，用户不会指定某个文件进行写操作，可以等到文件写成功后生成文件名并由客户端保存。

TFS 利用这两个特点大大地简化了文件系统写流程和元数据管理服务器的设计，而这也正是海量文件系统最为复杂之处。

淘宝是一个开放、共享的数据公司，还通过数据仓库提供各种数据给客户。目前使用了 Oracle RAC 集群提供服务，当然，也通过 Hadoop + HIVE 进行一些线下的预处理。

淘宝的主搜索其实是一个实时搜索，卖家更新的商品信息需要秒级别反映到用户的搜索结果中。淘宝的主搜索是很灵活的，可以根据商品类别，卖家名称，商品属性等进行搜索，因此，主搜索的存储系统需要建立不同维度的索引信息，主搜索使用内部的 iSearch 产品，机器被分成 56 组，每组 14 台，组内机器存储相同的数据。商品更新发生在 Oracle 商品库中，并以异步的方式同步到主搜索索引系统。

8.2 搜索类

搜索类公司的核心竞争力，或者说互联网公司的核心竞争力都可以认为就是数据以及对数据的处理能力，比如商业价值挖掘，用户意图挖掘等。搜索类最成功的当然就是 Google，它能取得现在的成功很大程度上得益于底层的 GFS/MapReduce/Bigtable 等带来的大规模数据处理能力。

搜索流程大致包括：抓取、数据分析、建立索引及索引服务。通过 spider 将网页抓取过来后存储到本地的分布式存储系统，即网页库中。网页库的业务逻辑并不复杂，无非就是对某一个网页或者一批网页，如某个域名下的所有网页的查询，删除一批网页或者更新网页相关的信息，比如权重等。但是网页库的数据量太大，假设需要处理 500 亿网页，每个网页平均存储大小为 50KB，那么，网页库的大小为 50GB * 50KB = 2.5PB，这已经远远超出了关系型数据库的处理能力。网页库应用为半线上应用，采用 GFS 加 Bigtable 的方案最为合适。不过为了规避复杂性，可以简单地将网页库通过 Hash 的方法分布到多台机器组成的分布式集群中，并通过支持 MapReduce 来进行线下挖掘，Rank 调研等。

将网页库的内容进行一系列的处理，比如计算 PageRank，网页去重，最终将生成倒排表用于线上服务。 搜索命令的处理大致分成两步：第一步从倒排表中找出匹配的网页索引信息，第二步根据索引信息从网页库中获取网页内容。倒排表有一个特点就是读取量特别大，要求延迟很小，且倒排表一般是定时生成的，也就是说，倒排表中的数据基本是静态的。倒排表的问题域和存储系统有些差别，这是因为每个关键词对应的网页信息非常多，需要分散到多台机器以便后续的计算。因此，主流的搜索引擎一般将机器分成多个 group，每个 group 可能包含几十台机器，存储相同的数据，每个查询请求都发送到所有的 group ，每个 group 中选择一台机器进行计算，计算完成后合并最终结果。

8.3 社交类

- IM 类

IM 类应用需要存储的数据有两类：用户数据及消息数据。用户数据的存储比较简单，假设每个用户的信息为 10K，有 10 亿用户，用户数据量为 10K * 1GB = 10TB，可以使用 5.4 中的线上最终一致性系统方案或者专用的根据用户 id 进行数据划分的方案。

消息分为两种，在线消息和离线消息，其中，离线消息存储时一个必要的功能，而在线消息是一个 plus，它和离线消息在数据量上有巨大差距，可以选择不在服务器端存储。个人消息和群消息也有一些区别。个人消息处理比较简单，而群的消息处理和 SNS 中订阅好友动态有些类似。SNS 中用户更新动态时，系统会将这个动态更新推送给用户的所有好友，而在 IM 中，用户往群里面发送一条消息，有两种处理方法：第一种方法是推送给群里面的所有用户，第二种方法是直接保存到群中。采用第一种方法群消息的数据量会增大很多倍，采用第二种方法减少了群消息的数据量，不过几十上百个用户同时读取群消息，即使群消息是顺序存储的，最后在磁盘上也变成了随机跳读。另外一种折衷的方案是对在线用户采用第一种方法，离线用户采用第二种方法，即只将群消息推送给在线用户，离线用户上线后主动拉取群离线消息。

- SNS & 微博客

前面已经提到了 SNS &微博客的消息推送功能，这是通过将消息推送给所有的好友实现的，可以开发一个类似 Active MQ 的支持发布/订阅机制的消息队列。另外，微博客支持实时搜索，例如新浪微博中搜索关注人说的话，这和邮件系统的搜索类似，只需要对用户订阅的消息进行字符串匹配。微薄中的一个难点问题是某些用户被关注程度特别高，如果采用推送的方式将对系统产生很大的压力，个人认为可以采用推拉结合的方式。

8.4 邮箱类

邮箱应用的需求有两类，一类是对邮件的顺序读取（如邮件分页显示），随机读取（读取某一封邮件）；另一类是根据关键词，标签的检索功能。

第一类需求的业务逻辑比较简单，难点还是在于邮箱的数据量，假设邮件服务有 1 亿用户，每个用户平均占用空间 100MB，总数据量为 0.1G * 100MB = 10PB。由于某些用户占用的空间很大，比如 10GB，所以不能简单地通过 user_id Hash 划分数据并限定用户的所有数据在一台机器中。

5.4 的线上最终一致性系统设计的容量为百 TB 级别，存储 10PB 的数据还需要做很多工作，比如集群规模可能接近万台，总控 Master 节点需要优化甚至总控节点本身需要分布式实现。5.6 中 GFS + Bigtable 解决方案一般会采用特别廉价的故障率很高的硬件，比如采用单机 12 块，每块 1TB 的 SATA 磁盘，这样，存储三份 10PB 数据需要的机器数为 10 PB * 3 / 12TB = 2500 台，总控 Master 单点优化后能够处理。不过前面已经介绍，Bigtable 的 Tablet Server 宕机时服务的数据范围需要等一段时间才能迁移到其它机器，期间不能进行读写服务，因此，Bigtable 提供线上邮箱服务还需要支持集群异步拷贝功能，写操作同步到备份集群，备份集群可提供读服务。

第二种根据标签和关键词的检索功能实现时需要扫描用户的全部数据，筛选出包含某个标签或者包含某个关键词的邮件，对存储系统而言，这其实是一个顺序扫描并进行字符串匹配的过程。

8.5 图片及视频类

图片和视频应用的特点是数据量和网络流量大且更新很少，系统架构上需要后台的大规模存储系统和靠近用户的 CDN 缓存系统支持。淘宝的核心系统部门在这方面做了非常出色的工作，CDN 通过各种方法，比如优化的全局调度算法，采用 SSD + SAS + SATA 混合存储等技术缩短访问延迟并节约成本，TFS 解决了百亿级淘宝图片存储问题。

视频类应用的特点是视频可能比较大，因此存储的时候需要对视频文件切片，可以将视频文件切片后存储到类似 TFS 的文件系统中，并在关系型数据库中保存视频切片的索引信息。

8.6 数据仓库类

数据仓库的特点是数据量特别大，数据列维度很高，且查询需要用到各种列维度的组合。OLAP 分成两部分：数据 CUBE 预处理和 CUBE 实时查询。

数据 CUBE 预处理涉及的数据量非常庞大，可通过 5.6 的半线上及线下系统解决，目前一般使用开源的 Hadoop 及其上封装的 HIVE 工具。数据 CUBE 预处理以后生成的 CUBE 数据提供线上服务，CUBE 数据可以采用 5.4 中的线上最终一致性系统提供服务，也可以采用并行数据库提供服务。

数据仓库底层一般可以使用列式存储引擎，提高压缩比和按列访问的效率。用户的查询操作发送到多个存储节点执行后再进行结果合并，排序(limit, order by)等。数据仓库的数据有一个特点：数据更新一段时间之后就不再更新了，变成历史数据。实时数据一般比较少，

可以存放到内存中，历史数据需要建索引，一般的索引方法包括 Btree 索引，Bitmap 索引等。数据库的查询数量可能不多，比如统计应用写多于读，但是每个查询涉及的数据量一般都很大。类似 max, min, count 这样的查询也可以生成一些视图（即冗余数据），提高查询性能。

8.7 云服务类

云服务的问题很多，比如研究人员经常提到的云安全问题。不过，云服务的第一步还是在工程设计和实现一个多应用共享的、可扩展的通用存储系统。我们能够想到的云服务提供商包括 Amazon，Google，Microsoft，其中 Amazon 和 Google 公布了其存储系统设计细节。

Amazon 云服务存储相关的系统 S3 内部的实现与 Dynamo 有些类似。Amazon 系统的优势是没有单点，因此某个模块设计或者编码出现问题都不会对整体系统造成特别大的影响。缺点就是一致性方面始终不够完美。

Google 的系统是贵族式的，在可扩展性及成本上具有无可比拟的优越性，问题就是从硬件选型，文件系统，表格系统的设计到单点服务避免等是一个巨大的系统工程，对架构师和工程师要求都非常高。另外，由于 Google 的设计有单点，某些设计或者编码上的问题可能导致系统整体停机现象，如 Gmail 停机。

9 工程实现注意事项

分布式系统的理论和工程实现完全是两个概念。理论上一个很简单的问题，工程上如果考虑宕机，磁盘故障等各种异常情况，都会变得比较复杂；另外，理论上相互独立的几个问题，比如子表分裂，子表迁移，工程实现时交织在一套系统中使得问题复杂度成平方级甚至指数级增长。

9.1 工程现象

1, 错误必然出现原理。只要是理论上有问题的设计/实现，运行时一定会出现，不管概率有多低。如果没有出现问题，要么是稳定运行时间不够长，要么是压力不够大。

2, 错误的必然复现原则。实践表明，分布式系统测试中发现的错误等到数据规模增大以后必然会复现。分布式系统中出现的多机多线程问题有的非常难于排查，但是，没关系，根据现象推测原因并补调试日志吧，加大数据规模，错误肯定会复现的。

3, 两倍数据规模原则。实践表明，分布式系统最大数据规模翻番时，都会发现以前从来没有出现过的问题。这个原则当然不完全是准确的，不过可以指导我们做开发计划。不管我们的系统多么稳定，不要高兴太早，数据量翻番一定会出现很多意想不到的情况。

4, 用户足够聪明。如果不经意将不希望暴露的 API 暴露给用户，尽管没有 API 说明，总有人能够发现。如果随意修改你认为不可能有人使用的 API，总有用户会抱怨。一个极端的例子是曾经有用户通过在程序中调用 grep 使用 utility 打印到屏幕上的说明信息，修改了说明信息后导致用户程序出错。

5, 怪异的现象的背后总有一个愚蠢的初级 bug。调试过程中有时候会发现一些特别怪异的错

误，比如总线错误，core 的堆栈面目全非，等等，不用太担心，仔细 Review 代码，看看编译连接的库是否版本错误等，特别怪异的现象背后对应的一般是很初级的 bug。

9.2 规范制订

经理和架构师需要参与的内容至少包括：
1, 确定技术方案；
2, 模块划分及接口设计；
3, 代码规范制订；
4, 关键算法和流程的代码 Review；
每个问题总会有多种技术方案，架构师要有能力在整体上从稳定性、性能及工程复杂度明确一种设计方案，而且要求思考得很细，切忌模棱两可，需要明确**工程开发与研究工作完全是两个概念**。

9.3 经验法则

1, 简单性原则。工程上的事情越简单越好，一般来说，复杂的事情都是思考逻辑混乱而不是系统的本来面貌。
2, 精力投入原则。不是什么事情都是做得越完美越好，系统设计时我们需要把大量的精力花在频繁出现的事件上，比如优化占整体时间比例较大或者调用次数较多的例程。
3, 伪代码细化设计。设计过程中出现想不清楚的问题时，先尽量和项目组的同学讨论，如果有大致的做法但是无法确定做法是否可以实现，尝试写出伪代码，而不是等到编码的时候再来思考。
4, 先稳定再优化。系统整体性能的关键在于架构，程序中多一些 double check 等编码规范需要的操作不会对整体性能造成影响。

9.4 质量控制

9.4.1 测试第一

分布式开发有一个经验是：如果一个系统或者一个模块设计时没有想好怎么测试，说明设计做得还不够。比如开发一个类似 Chubby 的系统，如果想好了怎么测试，基本就可以开始开发了。系统服务的数据规模越大，开发人员调试和测试人员测试的时间就越长。项目进展到后期需要依靠测试人员推动，测试人员的素质直接决定项目成败。

9.4.2 代码 Review

代码中的一些 bug，比如多线程 bug，异常情况处理 bug，后期发现并修复的成本很高。我

们经历过系统的数据规模达到 10TB 才会出现 bug 的情况，这样的 bug 需要系统持续运行接近 48 小时，并且我们分析了大量的调试日志才发现了问题所在。前期的代码 Review 很重要，我们没有必要代码 Review 带来的时间浪费，因为编码时间在整个项目周期中只占很少一部分。

9.4.3 服务器程序的资源管理

内存，线程池，socket 连接等都是服务器资源，设计的时候就需要确定资源的分配和使用。比如，对于内存使用，设计的时候需要计算好服务器的服务能力，常驻内存及临时内存的大小，系统能够自发现内存使用异常。一般来说，可以设计一个全局的内存池，管理内存分配和释放，并监控每个模块的内存使用情况。线程池一般在服务器程序启动时静态创建，一般不允许动态创建线程的情况。

10 致谢

11 参考文献

11.1 书籍类

[1] <<Distributed Systems: Principles and Paradigms>>
[2] << High Performance Mysql>>

11.2 论文类

11.2.1 分布式理论

[1] Time, clocks, and the ordering of events in a distributed system.
[2] Impossibility of distributed consensus with one faulty process.
[3] CAP: Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.
[4] Base: An acid alternative.
[5] Life beyond Distributed Transactions: an Apostate’s Opinion
[6] The part-time parliament.
[7] Paxos Made Simple.
[8] Paxos Made Practical.
[9] Paxos made live . An engineering perspective.
[10] Consensus on Transaction Commit.

11.2.2 Google 系列

[1] The Google file system.
[2] MapReduce: Simplified data processing on large clusters.
[3] Bigtable: A Distributed Storage System for Structured Data.
[4] The Chubby lock service for loosely-coupled distributed systems.
[5] The Datacenter as a Computer.
[6] Interpreting the data: Parallel analysis with Sawzall.
[7] Web search for a planet: The Google cluster architecture.
[8] Designs, Lessons and Advice from Building Large Distributed Systems

11.2.3 Dynamo 及 P2P 系列

[1] Dynamo: Amazon's highly available key-value store.
[2] Cassandra: A Decentralized Structured Storage System.
[3] Chord: A scalable peer-to-peer lookup service for Internet applications.
[4] Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.

11.2.4 存储系统

[1] Parallel database systems: The future of high performance database systems.

11.2.5 计算系统

[1] NowSort: High-Performance Sorting on Networks of Workstations.
[2] Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks
[3] The Design of the Borealis Stream Processing Engine
[4] Availability-Consistency Trade-offs in a Fault-Tolerant Stream Processing System

11.2.6 其它

[1] PNUTS: Yahoo!'s Hosted Data Serving Platform.
[2] Boxwood: Abstractions as the foundation for storage infrastructure.
[3] The dangers of replication and a solution.
[4] Niobe: A Practical Replication Protocol.
[5] Data compression using long common strings.
[6] Large-scale Incremental Processing Using Distributed Transactions and Notifications.
[7] SEDA: an architecture for well-conditioned, scalable internet services.
[8] B-trees, Shadowing, and Clones.
[9] Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems.

11.3网页类

11.3.1 个人博客类

[1] http://duartes.org/gustavo/blog/
[2] http://www.allthingsdistributed.com/
[3] http://www.yankay.com/
[4] http://timyang.net/
[5] http://dbanotes.net/

11.3.2 专题类

[1] http://nosql-database.org/
[2] http://highscalability.com/
[3] Hadoop: Open source implementation of MapReduce. http://lucene.apache.org/hadoop/.
[4] http://en.wikipedia.org/wiki/Two-phase_commit_protocol
[5] http://codahale.com/you-cant-sacrifice-partition-tolerance/
[6] http://blog.nosqlfan.com/

11.3.3 其它

[1]
http://www.yankay.com/wp-content/uploads/2010/02/NoSql%20Database%20Note/#NOSQL__0
9502721972778405