第5章

大数据存储

随着结构化数据量和非结构化数据量的不断增长以及分析数据来源的多样化，之前的存储系统设计已无法满足大数据应用的需求。对于大数据的存储，存在以下几个不容忽视的问题。

（1）容量：大数据时代存在的第一个问题就是“大容量”。“大容量”通常指可达PB级的数据规模，因此海量数据存储系统的扩展能力也要得到相应等级的提升，同时其扩展还必须渐变，为此，通过增加磁盘柜或模块增加存储容量，这样可以不需要停机。

（2）延迟：大数据应用不可避免地存在实时性的问题，大数据应用环境通常需要较高的IOPS性能。为了迎接这些挑战，小到简单的在服务器内用作高速缓存的产品，大到全固态介质可扩展存储系统，各种模式的固态存储设备应运而生。

（3）安全：大数据分析往往需要对多种数据混合访问，这就催生出了一些新的、需要重新考虑的安全性问题。

（4）成本：成本控制是企业的关键问题之一，只有让每一台设备都实现更高的“效率”，才能控制住成本。目前进入存储市场的重复数据删除和多数据类型处理等技术都可为大数据存储带来更大的价值，提升存储效率。

（5）灵活性：通常大数据存储系统的基础设施规模都很大，为了保证存储系统的灵活性，使其能够随时扩容及扩展，必须经过详细的设计。

由于传统关系型数据库的局限性，传统的数据库已经不能很好地解决这些问题。在这种情况下，一些主要针对非结构化数据的管理系统开始出现。这些系统为了保障系统的可用性和并发性，通常采用多副本的方式进行数据存储。为了在保证低延时的用户响应时间的同时维持副本之间的一致状态，采用较弱的一致性模型，而且这些系统也普遍提供了良好的负载平衡策略和容错机制。

5.1 大数据存储技术发展

20世纪50年代中期以前，计算机主要用于科学计算，这个时候存储的数据规模不大，数据管理采用的是人工管理的方式；20世纪50年代后期至60年代后期，为了更加方便管理和操作数据，出现了文件系统；从20世纪60年代后期开始，出现了大量结构化数据，数据库技术蓬勃发展，开始出现了各种数据库，其中关系型数据库备受人们喜爱。

在科学研究过程中，为了存储大量的科学计算，有Beowulf集群的并行文件系统PVFS做数据存储，在超级计算机上有Lustre并行文件系统存储大量数据，IBM公司在分布式文件系统领域研制了GPFS分布式文件系统，这些都是针对高端计算采用的分布式存储系统。

进入21世纪以后，互联网技术不断发展，其中以互联网为代表企业产生大量数据，为了解决这些存储问题，互联网公司针对业务需求和成本开始设计自己的存储系统，典型代表是Google公司于2003年发表的论文[2]，其建立在廉价的机器上，提供了高可靠、容错的功能。为了适应Google的业务发展，Google推出了一种NoSQL非关系型数据库系统——BigTable，用于存储海量网页数据，数据存储格式为行、列簇、列和值的方式；与此同时亚马逊公司公布了他们开发的另外一种NoSQL系统——DynamoDB。后续大量的NoSQL系统不断涌现，为了满足互联网中大规模网络数据的存储需求，Facebook结合BigTable和DynamoDB的优点推出了Cassandra非关系型数据库系统。

开源社区对于大数据存储技术的发展更是贡献重大，其中包括底层的操作系统层面的存储技术，例如文件系统Btrfs和XFS等。为了适应当前大数据技术的发展，支持高并发、多核以及动态扩展等，Linux开源社区针对技术发展需求开发了下一代操作系统的文件系统Btrfs，该文件系统在不断完善；同时也包括分布式系统存储技术，功不可没的是Apache开源社区，其贡献和发展了HDFS、HBase等大数据存储系统。

总体来讲，结合公司的业务需求以及开源社区的蓬勃发展，当前大数据存储系统不断涌现。

5.2 海量数据存储的关键技术

大数据处理面临的首要问题是如何有效地存储规模巨大的数据。无论是从容量还是从数据传输速度，依靠集中式的物理服务器保存数据是不现实的，即使存在一台设备可以存储所有的信息，用户在一台服务器上进行数据的索引查询也会使处理器变得不堪重负，因此分布式成为这种情况下很好的解决方案。要实现大数据存储，需要使用几十台、几百台甚至更多的分布式服务器节点。为保证高可用、高可靠和经济性，海量数据多采用分布式存储的方式存储数据，采用冗余存储的方式保证存储数据的可靠性，即为同一份数据存储多个副本。

数据分片与数据复制的关系如图5-1所示。

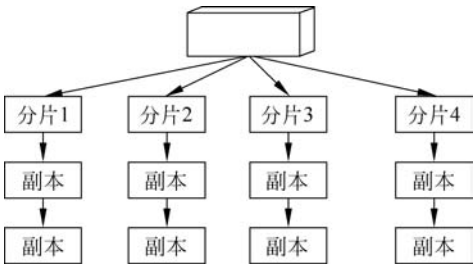

图 5-1 数据分片与数据复制

5.2.1 数据分片与路由

传统数据库采用纵向扩展方式,通过改善单机硬件资源配置解决问题;主流大数据存储与计算系统采用横向扩展方式,支持系统可扩展性,即通过增加机器获得水平扩展能力。

对于海量数据,将数据进行切分并分配到各个机器中的过程叫分片(Shared/Partition),即将不同数据存放在不同节点。数据分片后,找到某条记录的存储位置称为数据路由(Routing)。数据分片与路由的抽象模型如图 5-2 所示。

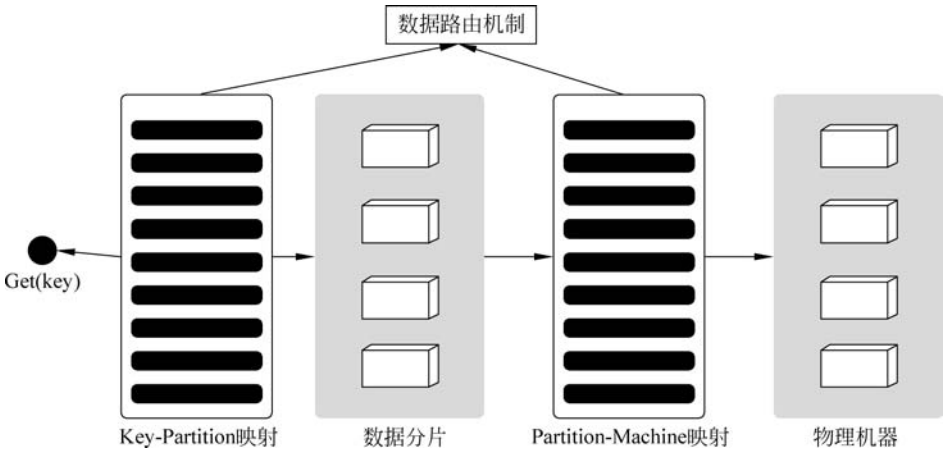

图 5-2 数据分片与路由的抽象模型

首先要介绍的是数据分片。

一般来说,数据库的繁忙体现在不同用户需要访问数据集中的不同部分。在这种情况下,把数据的各个部分存放在不同的服务器/节点中,每个服务器/节点负责自身数据的读取与写入操作,以此实现横向扩展,这种技术称为分片。

用户必须考虑以下两点。

一是如何存放数据。解决了这一点就可以实现用户从一个逻辑节点(实际多个物理节点的方式)获取数据,并且不用担心数据的存放位置。面向聚合的数据库可以很容易地解决这个问题。聚合结构指把经常需要同时访问的数据存放在一起,因此可以把聚合作

为分布数据的单元。

二是如何保证负载平衡，即如何把聚合数据均匀地分布在各个节点中，让它们需要处理的负载量相等。负载分布情况可能会随着时间变化，因此需要一些领域特定的规则。例如有的需要按字典顺序，有的需要按逆域名序列等。

下面介绍的是分片类型。

1. 哈希分片

采用哈希函数建立 Key-Partition 映射，其只支持点查询，不支持范围查询，主要有 Round Robin、虚拟桶和一致性哈希 3 种算法。

1) Round Robin

其俗称哈希取模算法，这是最常用的数据分片方法。若有 k 台机器，分片算法如下：

```
1    H(key) = hash(key) mod k
```

对物理机进行编号(0～k－1)，根据以上哈希函数，对于以 key 为主键的某个记录，H(key)的数值即是物理机在集群中的放置位置(编号)。这种算法的优点是实现简单。其缺点是缺乏灵活性，若有新机器加入，之前所有数据与机器之间的映射关系都被打乱，需要重新计算。

2) 虚拟桶

虚拟桶算法在 Round Robin 的基础上加入一个虚拟桶层形成两级映射。具体以 Membase 为例，如图 5-3 所示。

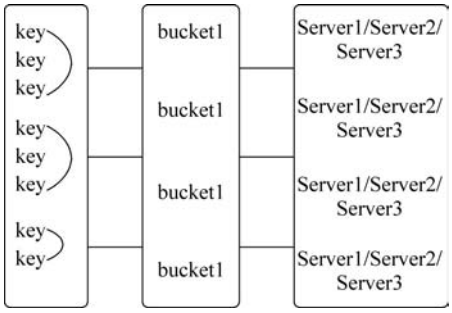

图 5-3 Membase 虚拟桶的运行

Membase 在待存储记录的物理机之间引入了虚拟桶层，所有记录首先通过哈希函数映射到对应的虚拟桶，记录和虚拟桶是多对一的关系，即一个虚拟桶包含多条记录信息；第二层映射是虚拟桶和物理机之间的映射关系，同样也是多对一映射，一个物理机可以容纳多个虚拟桶，具体是通过查找表实现的，即 Membase 通过内存表管理这些映射关系。

对照抽象模型可以看出，Membase 的虚拟桶层对应数据分片层，一个虚拟桶就是一个数据分片。Key-Partition 映射采用映射函数。

与 Round Robin 相比，Membase 引入了虚拟桶层，这样将原先由记录直接到物理机的单层映射解耦成两级映射。当新加入机器时，将某些虚拟桶从原先分配的机器重新分

配到各机器，只需要修改 Partition-Machine 映射表中受影响的个别条目就能实现扩展。这种做法增加了系统扩展的灵活性，但实现相对麻烦。

3）一致性哈希

一致性哈希是分布式哈希表的一种实现算法，将哈希数值空间按照大小组成一个首尾相接的环状序列。对于每台机器，可以根据 IP 和端口号经过哈希函数映射到哈希数值空间内，通过有向环顺序或路由表查找。对于一致性哈希可能造成的各个节点负载不均衡的情况，可以采用虚拟节点的方式解决。一个物理机节点虚拟成若干虚拟节点，映射到环状结构的不同位置。图 5-4 为哈希空间长度为 5 的二进制数值($m=5$)的一致性哈希算法示意图。

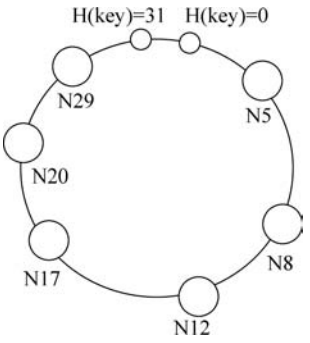

图 5-4　一致性哈希算法

在哈希空间可容纳长度为 32 的二进制数值($m=32$)空间内，每个机器根据 IP 地址或者端口号经过哈希函数映射到环内(图 5-4 中 6 个大圆代表机器，后面的数字代表哈希值，即根据 IP 地址或者端口号经过哈希函数计算得出在环状空间内的具体位置)，而这台机器负责存储落在一段有序哈希空间内的数据，例如 N12 节点存储哈希值在 9～12 的数据，而 N5 负责存储哈希值落在 30～31 和 0～5 的数据。同时，每台机器还记录着自己的前驱和后继节点，成为一个真正意义上的有向环。

2. 范围分片

范围分片首先将所有记录的主键进行排序，然后在排好序的主键空间内将记录划分成数据分片，每个数据分片存储有序的主键空间片段内的所有记录。

支持范围查询即给定记录主键的范围而一次读取多条记录，范围分片既支持点查询，也支持范围查询。

分片可以极大地提高读取性能，但对于频繁写的应用帮助不大。同时，分片也可减少故障范围，只有访问故障节点的用户才会受影响，访问其他节点的用户不会受到故障节点的影响。

那么如何根据收到的请求找到存储的值呢？这就涉及路由的知识。下面介绍 3 种路由的方法。

1）直接查找法

如果哈希值落在自身管辖的范围内，则在此节点上查询，否则继续往后找，一直找到节点 Nx，x 是大于或等于待查节点值的最小编号，这样一圈下来肯定能找到结果。

以图 5-4 为例，若有一个请求向 N5 查询的主键为 H(key)＝6，因为此哈希值落在 N5 和 N8 之间，所以该请求的值存储在 N8 的节点上，即如果哈希值落在自身管辖的范围内，则在此节点上查询，否则继续往后找，一直找到节点 Nx。

2）路由表法

直接查找法缺乏效率，为了加快查找速度，可以在每个机器节点配置路由表，路由表

存储每个节点到每个除自身节点的距离，具体示例见表 5-1。

表 5-1　机器节点的路由表

距离	1	2	4	8	16
机器节点	N17	N17	N17	N20	N29

在表 5-1 中，第 3 项代表与 N12 的节点距离为 4 的哈希值(12＋4＝16)落在 N17 节点上，同理第 5 项代表与 N12 的节点距离为 16 的哈希值落在 N29 节点上，这样找起来非常快速。

3) 一致性哈希路由算法

同样如图 5-4 所示，若请求 N5 节点查询，则 N5 节点的路由表如表 5-2 所示。

表 5-2　N5 节点的路由表

距离	1	2	4	8	16
机器节点	N8	N8	N12	N17	N29

假如请求的主键哈希值为 H(key)＝24，首先查询是否在 N5 的后继节点上，发现后继节点 N8 小于主键哈希值，则根据 N5 节点的路由表查询，发现大于 24 的最小节点为 N29(只有 29，因为 5＋16＝21＜24)，因此哈希值落在 N29 节点上。

5.2.2　数据复制与一致性

将同一份数据放置到多个节点的过程称为复制，例如主从(Master-Slave)复制和对等(Peer-to-Peer)复制，数据复制可以保证数据的高可用性。

1. 主从复制

主从复制中有一个 Master 节点用于存放重要数据，通常负责数据的更新，其余节点都叫 Slave 节点，复制操作就是让 Slave 节点的数据与 Master 节点的数据同步。其优点有两点：①在频繁读取的情况下有助于提升数据的访问(读取 Slave 节点分担压力)，还可以增加多个 Slave 节点进行水平扩展，同时处理更多的读取请求。②可以增强读取操作的故障恢复能力。一个 Slave 节点出故障，还有其他 Slave 节点保证访问的正常进行。它的缺点是如果数据更新没有通知全部的 Slave 节点，则会导致数据不一致。

2. 对等复制

主从复制有助于增强读取操作的故障恢复能力，对写操作频繁的应用没有帮助。它所提供的故障恢复能力只有在 Slave 节点出错时才能体现出来，Master 节点仍然是系统的瓶颈。对等复制是指两个节点相互为各自的副本，没有主从的概念。其优点是丢失其中一个节点不影响整个数据库的访问。但因为同时接受写入请求，容易出现数据不一致问题。在实际使用中，通常只有一个节点接受写入请求，另一个 Master 节点作为候补，只有当对等的 Master 节点出故障时才会自动承担写操作请求。

3. 数据一致性

分布式存储系统的一致性问题总随着数据复制而产生,一致性模型的定义如下。

(1) 强一致性。按照某一顺序串行执行存储对象的 I/O 操作,更新存储对象之后,后续访问总是读取最新值。假如进程 A 先更新了存储对象,存储系统保证后续 A、B、C 的读取操作都将返回最新值。

(2) 弱一致性。更新存储对象之后,后续访问可能无法读取最新值。假如进程 A 先更新了存储对象,存储系统不能保证后续 A、B、C 的读取操作能读取最新值。从更新成功这一刻开始算起,到所有访问者都能读取修改后的对象为止,这段时间称为“不一致性窗口”,在该窗口内访问存储时无法保证一致性。

(3) 最终一致性。最终一致性是弱一致性的特例,存储系统保证所有访问将最终读取对象的最新值。例如,进程 A 写入一个存储对象,如果存储对象后续没有更新操作,那么最终 A、B、C 的读取操作都会读取 A 写入的值。“不一致性窗口”的大小依赖交互延迟、系统的负载以及副本个数等。

5.3 重要数据结构和算法

分布式存储系统中存储大量数据,同时需要支持大量上层 I/O 操作,为了实现高吞吐量,设计和实现一个良好的数据结构能起到重要作用。典型的如 LSM 树结构,为 NoSQL 系统对外实现高吞吐量提供了更大的可能。在大规模分布式系统中需要查找到具体的数据,设计一个良好的数据结构,以支持快速的数据查找,如 MemC3 中的 Cuckoo Hash,为 MemC3 在读多写少的负载情况下极大地减少了访问延迟; HBase 中的 Bloom Filter 结构,用于在海量数据中快速确定数据是否存在,减少了大量数据访问操作,从而提高了总体数据访问速度。

因此,一个良好的数据结构和算法对于分布式系统来说有着很大的作用。下面讲述当前大数据存储领域中一些比较重要的数据结构。

5.3.1 Bloom Filter

Bloom Filter 用于在海量数据中快速查找给定的数据是否在某个集合内。

如果想判断一个元素是不是在一个集合内,一般想到的是将集合中的所有元素保存起来,然后通过比较确定,链表、树和散列表(又叫哈希表,Hash Table)等数据结构都是这种思路。但是随着集合中元素的增加,需要的存储空间越来越大,同时检索速度也越来越慢,上述 3 种结构的检索时间复杂度分别为 $O(n)$、$O(\log n)$和 $O(n/k)$。

Bloom Filter 的原理是当一个元素被加入集合时,通过 k 个散列函数将这个元素映射成一个位数组中的 k 个点,把它们置为 1。检索时,用户只要观察这些点是不是都是 1 就大约知道集合中有没有被检元素了: 如果这些点有任何一个 0,则被检元素一定不在; 如果都是 1,则被检元素很可能在。

Bloom Filter 的高效是有一定代价的: 在判断一个元素是否属于某个集合时,有可能

会把不属于这个集合的元素误认为属于这个集合。因此，Bloom Filter 不适合那些“零错误”的应用场合。在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

下面具体看 Bloom Filter 是如何用位数组表示集合的。初始状态如图 5-5 所示，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。

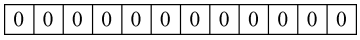

图 5-5　Bloom Filter 初始位数组

为了表达 $S=\{x_1,x_2,\cdots,x_n\}$这样一个有 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数(Hash Function)，它们分别将集合中的每个元素映射到$\{1,2,\cdots,m\}$的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 $h_i(x)$会被置为 1($1\leqslant i\leqslant k$)。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在图 5-6 中，$k=3$，且有两个哈希函数选中同一个位置(从左边数第 5 位，即第 2 个“1”处)。

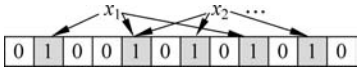

图 5-6　Bloom Filter 哈希函数

在判断 y 是否属于这个集合时，对 y 应用 k 次哈希函数，如果所有 $h_i(y)$的位置都是 1($1\leqslant i\leqslant k$)，那么就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。图 5-7 中的 y_1 就不是集合中的元素(因为 y_1 有一处指向了 0 位)，y_2 属于这个集合或者不属于这个集合。

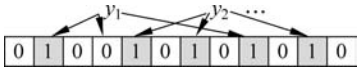

图 5-7　Bloom Filter 查找

这里举一个例子。有 A、B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4GB，试找出 A、B 文件共同的 URL。如果是 3 个或者 n 个文件呢？

根据这个问题计算一下内存的占用，4GB=2^{32}B，大概是 43 亿，乘以 8 大概是 340 亿 b，n=50 亿，如果按出错率为 0.01，则大概需要 650 亿 b。现在可用的是 340 亿 b，相差并不多，这样可能会使出错率上升一些。另外，如果这些 URL 和 IP 是一一对应的，就可以转换成 IP，这样就简单许多了。

5.3.2　LSM Tree

LSM Tree 存储引擎和 B Tree 存储引擎一样，同样支持增、删、读、改和顺序扫描操作，而且可通过批量存储技术规避磁盘随机写入问题。与 B+Tree 相比，LSM Tree 牺牲了部分读性能以换取写性能的大幅度提高。

LSM Tree 的原理是把一棵大树拆分成 n 棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会通过 FFlush 方式写入磁盘中，磁盘中的树定期可以做 Merge 操

作,合并成一棵大树,以优化读性能。

对于最简单的二层 LSM Tree 而言,内存中的数据和磁盘中的数据做 Merge 操作,如图 5-8 所示。

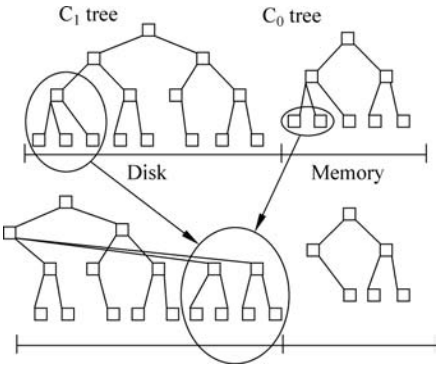

图 5-8 二层 LSM Tree

之前存在于磁盘的叶子节点被合并后,旧的数据并不会被删除,这些数据会复制一份和内存中的数据一起顺序写入磁盘中。这样操作会有一些空间的浪费,但是 LSM Tree 提供了一些机制回收这些空间。磁盘中的树的非叶子节点数据也被缓存在内存中。数据查找会首先查找内存中的树,如果没有查到结果,则会转而查找磁盘中的树。

为什么 LSM Tree 插入数据的速度比较快呢? 首先插入操作会作用于内存,由于内存中的树不会很大,因此速度快。同时,合并操作会顺序写入一个或多个磁盘页,比随机写入快得多。

5.3.3 Merkle Tree

Merkle Tree 是由计算机科学家 Ralph Merkle 提出的,并以他本人的名字来命名。因为 Merkle Tree 的所有节点都是 Hash 值,所以又被称为 Hash Tree。本书将从数据"完整性校验"(检查数据是否有损坏)的角度介绍 Merkle Tree。

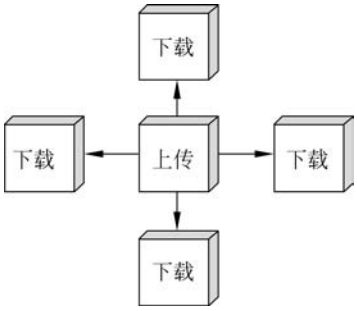

图 5-9 哈希拓扑

1. 哈希(Hash)

要实现完整性校验,最简单的方法就是对要校验的整个数据文件做哈希运算,将得到的哈希值发布在网上,当把数据下载后再次运算一下哈希值,如果运算结果相等,就表示下载过程中文件没有任何损坏。因为哈希的最大特点是,如果输入数据稍微变了一点,那么经过哈希运算,得到的哈希值将会变得完全不一样。构成的哈希拓扑结构如图 5-9 所示。

如果从一个稳定的服务器上进行下载,那么采用单个哈希进行校验的形式是可以接受的。

2. 哈希列表(Hash List)

在点对点网络中进行数据传输时,如图 5-10 所示,我们会同时从多个机器上下载数据,而其中很多机器可以认为是不稳定或者是不可信的,这时需要有更加巧妙的做法。在实际中,点对点网络在传输数据时都是把一个比较大的文件切成小数据块。这样的好处是如果有一小块数据在传输过程中损坏了,只要重新下载这个数据块,不用重新下载整个文件。当然,这要求每个数据块都拥有自己的哈希值。这样,系统在执行下载任务时,会先下载一个哈希列表,之后完成真实数据的下载。这时有一个问题出现了,如此多的哈希,我们怎么保证它们本身都是正确的呢?

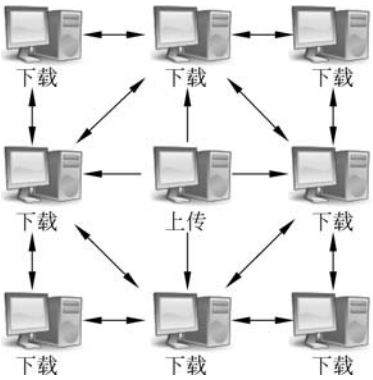

图 5-10　哈希列表

答案是我们需要一个根哈希,如图 5-11 所示,把每个小块的哈希值拼到一起,然后对这个长长的字符串再做一次哈希运算,最终的结果就是哈希列表的根哈希。如果我们能够保证从一个绝对可信的网站拿到一个正确的根哈希,就可以用它校验哈希列表中的每一个哈希是否都是正确的,进而可以保证下载的每一个数据块的正确性。

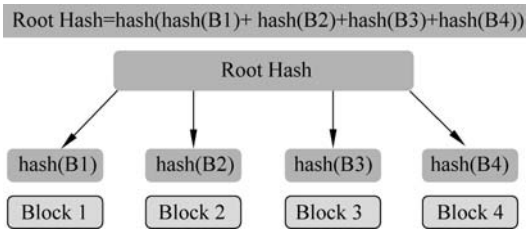

图 5-11　哈希流程

3. Merkle Tree 结构

在最底层,和哈希列表一样,我们把数据分成小数据块,有相应的哈希与它对应。但是往上走,并不是直接运算根哈希,而是把相邻的两个哈希合并成一个字符串,然后运算这个字符串的哈希,这样每两个哈希组合得到了一个“子哈希”。如果最底层的哈希总数是单数,那么到最后必然出现一个单哈希,对于这种情况直接对它进行哈希运算,所以也

能得到它的子哈希。于是往上推,依然是一样的方式,可以得到数目更少的新一级哈希,最终必然形成一棵倒着的树,到了树根的这个位置就剩下一个根哈希了,我们把它称为Merkle Root,Merkle Tree结构如图5-12所示。

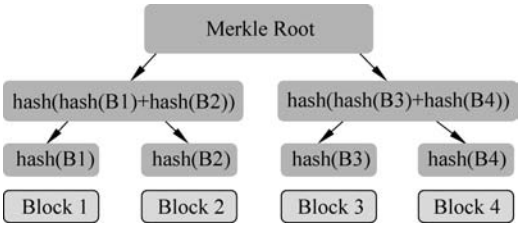

图5-12 Merkle Tree结构

与哈希列表相比,Merkle Tree明显的一个好处是可以单独拿出一个分支对部分数据进行校验,这是哈希列表所不能比拟的方便和高效。

5.3.4 Cuckoo Hash

Cuckoo Hash是一种解决Hash冲突的方法,其目的是使用简易的Hash函数提高Hash Table的利用率,保证$O(1)$的查询时间也能够实现Hash Key的均匀分布。

基本思想是利用两个Hash函数处理碰撞,从而使每个Key都对应到两个位置。

插入操作如下。

(1) 对Key值哈希,生成两个Hash Key值:hash k1和hash k2,如果对应的两个位置上有一个为空,直接把Key值插入即可。

(2) 否则,任选一个位置,把Key值插入,把已经在那个位置的Key值剔除。

(3) 被剔除的Key值需要重新插入,直到没有Key值被剔除为止。

其查找思路与一般哈希一致。

Cuckoo Hash在读多写少的负载情况下能够快速实现数据的查找。

5.4 分布式文件系统

5.4.1 文件存储格式

文件系统最后都需要以一定的格式存储数据文件,常见的文件系统存储布局有行式存储、列式存储以及混合式存储3种,不同的类别各有其优缺点和适用的场景。在目前的大数据分析系统中,列式存储和混合式存储因有诸多优点被广泛采用。

1. 行式存储

在传统关系型数据库中,行式存储被主流关系数据库广泛采用,HDFS文件系统也采用行式存储。在行式存储中,每条记录的各个字段连续地存储在一起,而对于文件中的各个记录也是连续地存储在数据块中,图5-13是HDFS的行式存储布局,每个数据块除了存储一些管理元数据外,每条记录都以行的方式进行数据压缩后连续地存储在一起。

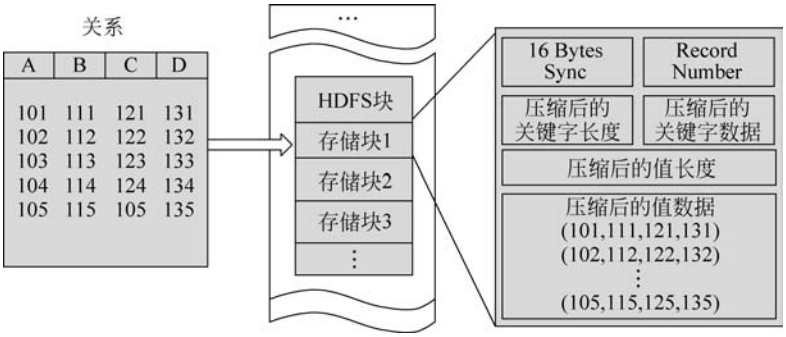

图 5-13　HDFS 的行式存储布局

行式存储对于大数据系统的需求已经不能很好地满足，主要体现在以下几方面。

(1) 快速访问海量数据的能力被束缚。行的值由相应的列的值定位，这种访问模型会影响快速访问的能力，因为在数据访问的过程中引入了耗时的输入/输出。在行式存储中，为了提高数据处理能力，一般通过分区技术减少查询过程中数据输入/输出的次数，从而缩短响应时间。但是这种分区技术对海量数据规模下的性能改善效果并不明显。

(2) 扩展性差。在海量规模下，扩展性差是传统数据存储的一个致命的弱点。一般通过向上扩展(Scale Up)和向外扩展(Scale Out)解决数据库扩展性差的问题。向上扩展是通过升级硬件提升速度，从而缓解压力；向外扩展则是按照一定的规则将海量数据进行划分，再将原来集中存储的数据分散到不同的数据服务器上。但由于数据被表示成关系模型，从而难以被划分到不同的分片中，这种解决方案仍然存在一定的局限性。

2. 列式存储

与行式存储布局对应，列式存储布局实际存储数据时按照列对所有记录进行垂直划分，将同一列的内容连续存放在一起。简单的记录数据格式类似传统数据库的平面型数据结构，一般采取列组(Column Group/Column Family)的方式。列式存储布局有两个好处：①对于上层的大数据分析系统来说，如果查询操作只涉及记录的个别列，则只需读取对应的列内容即可，其他字段不需要进行读取操作；②因为数据按列存储，所以可以针对每列数据采取具有针对性的数据压缩算法，从而提升压缩率。但是列式存储的缺陷也很明显，对于 HDFS 这种按块存储的模式而言，有可能不同列分布在不同的数据块中，所以为了拼合出完整的记录内容，可能需要大量的网络传输，导致效率低下。

采用列组方式存储布局可以在一定程度上缓解这个问题，也就是将记录的列进行分组，将经常使用的列分为一组，这样即使是列式存储数据，也可以将经常联合使用的列存储在一个数据块中，避免通过不必要的网络传输获取多列数据，对于某些场景而言会较大地提升系统性能。

在 HDFS 场景下，采用列组方式存储数据如图 5-14 所示，列被分为 3 组，A 和 B 分为一组，C 和 D 各自一组，即将列划分为 3 个列组并存储在不同的数据块中。

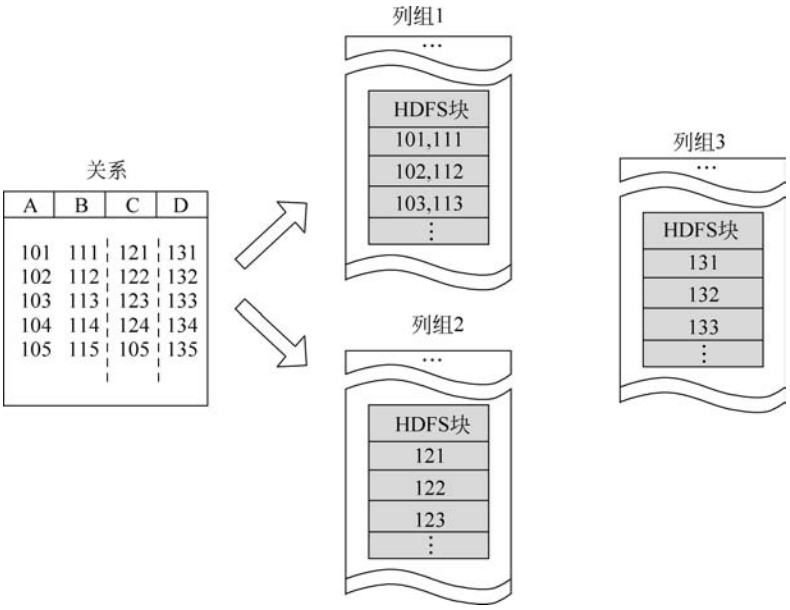

图 5-14　HDFS 列式存储布局

3. 混合式存储

尽管列式存储布局可以在一定程度上缓解上述的记录拼合问题，但是并不能彻底解决。混合式存储布局融合了行式和列式存储布局的优点，能比较有效地解决这一问题。

混合式存储布局首先将记录表按照行进行分组，若干行划分为一组，而对于每组内的所有记录，在实际存储时按照列将同一列内容连续存储在一起。

5.4.2　GFS

GFS(Google File System)是 Google 公司为了存储以百亿计的海量网页信息而专门开发的文件系统。在 Google 的整个大数据存储与处理技术框架中，GFS 是其他相关技术的基石，既提供了海量非结构化数据的存储平台，又提供了数据的冗余备份、成千台服务器的自动负载均衡以及失效服务器检测等各种完备的分布式存储功能。

考虑到 GFS 是在搜索引擎这个应用场景下开发的，在设计之初就定下了几个基本的设计原则。

首先，GFS 采用大量商业 PC 构建存储集群。PC 的稳定性并没有很高的保障，尤其是大规模集群，每天都会发生服务器宕机或者硬盘故障，这是 PC 集群的常态。因此，数据冗余备份、故障自动检测和故障机器自动恢复等都列在 GFS 的设计目标中。

其次，GFS 中存储的文件绝大多数是大文件，文件大小集中在 100MB 到几 GB，所以系统设计应该对大文件的 I/O 操作做出有针对性的优化。

再次，系统中存在大量的“追加”写操作，即在已有文件的末尾追加内容，已经写入的内容不做更改；而很少有“随机”写行为，即在文件的某个特定位置之后写入数据。

最后，对于数据读取操作来说，绝大多数操作都是“顺序”读，少量的操作是“随机”读，即按照数据在文件中的顺序一次读入大量数据，而不是不断在文件中定位到指定位置读取少量数据。

在下面的介绍中可以看到，GFS 的大部分技术思路都是围绕以上几个设计目标提出的。

在了解 GFS 整体架构之前首先了解一下 GFS 中的文件和文件系统。在应用开发者看来，GFS 文件系统类似 Linux 文件系统中的目录和目录下的文件构成的树形结构。这个树形结构在 GFS 中被称为“GFS 命名空间”，同时，GFS 提供了文件的创建、删除、读取和写入等常见的操作接口。

GFS 中大量存储的是大文件，文件大小超过几 GB 是很常见的。虽然文件大小各异，但 GFS 在实际存储时首先将不同大小的文件切割成固定大小的数据块，每一个块称为一个 Chunk。通常一个 Chunk 的大小设定为 64MB，这样每个文件就是由若干个固定大小的 Chunk 构成的。

GFS 以 Chunk 为基本存储单位，同一个文件的不同 Chunk 可能存储在不同的数据块服务器(ChunkServer)上，每个 ChunkServer 可以存储来自不同文件的 Chunk。另外，在 ChunkServer 内部会对 Chunk 进一步切割，将其切割为更小的数据块，每一块被称为一个 Block。Block 是文件读取的基本单位，即每次读取至少读一个 Block。

图 5-15 显示了 GFS 的整体架构，在这个架构中，主节点主要用于管理工作，负责维护 GFS 命名空间和 Chunk 命名空间。在 GFS 系统内部，为了能识别不同的 Chunk，每个 Chunk 都被赋予一个唯一的编号，所有 Chunk 编号构成了 Chunk 命名空间。由于 GFS 文件被切割成了 Chunk，因此主节点还记录了每个 Chunk 存储在哪台 ChunkServer 上以及文件和 Chunk 之间的映射关系。

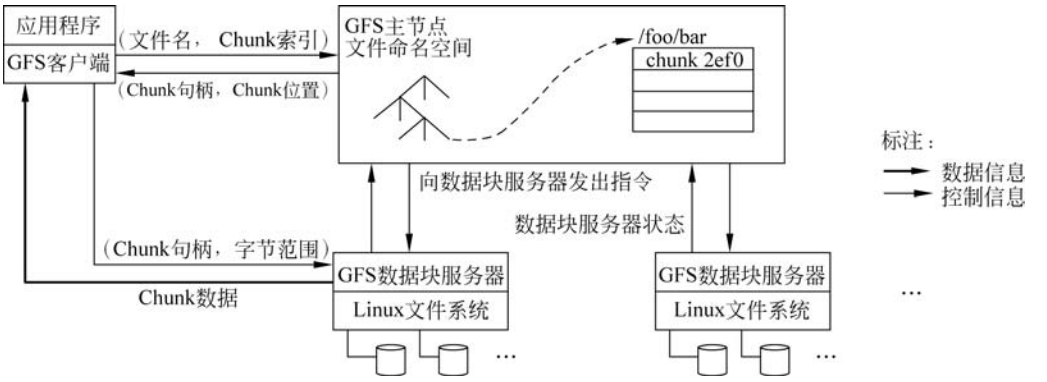

图 5-15　GFS 的整体架构

在 GFS 架构下，下面介绍 GFS 客户端是如何读取数据的。

对于 GFS 客户端来说，应用开发者提交的数据请求是从文件(file)中的位置 P 开始读取大小为 L 的数据。GFS 系统在收到这种请求后会在内部做转换，因为 Chunk 的大小是固定的，所以从位置 P 和大小 L 可以计算出要读的数据位于文件的第几个 Chunk 中，请求被转换为 file，Chunk 序号的形式。随后，这个请求被发送到 GFS 主节点，通过

主服务器可以知道要读的数据在哪台 ChunkServer 上,同时可以将 Chunk 序号转换为系统内唯一的 Chunk 编号,并将这两个信息传回 GFS 客户端。

GFS 客户端知道了应该去哪台 ChunkServer 读取数据后会和 ChunkServer 建立连接,并发送要读取的 Chunk 编号以及读取范围,ChunkServer 接收请求后将请求的数据发送给 GFS 客户端,如此就完成了一次数据读取的工作。

5.4.3 HDFS

Hadoop 分布式文件系统(HDFS)被设计成适合运行在商业硬件(Commodity Hardware)上的分布式文件系统。HDFS 和现有的分布式文件系统有很多共同点,但它和其他的分布式文件系统的区别也是很明显的。HDFS 是一个高度容错性的系统,适合部署在廉价的机器上;HDFS 能提供高吞吐量的数据访问,非常适合大规模数据集上的应用;HDFS 在最开始是作为 Apache Nutch 搜索引擎项目的基础架构开发的;HDFS 是 Apache Hadoop Core 项目的一部分。

HDFS 采用 Master/Slave 架构。一个 HDFS 集群由一个 NameNode 和一定数目的 DataNode 组成。NameNode 是一个中心服务器,负责管理文件系统的名字空间(Namespace)以及客户端对文件的访问。集群中的 DataNode 一般是一个服务器,负责管理它所在节点上的存储。HDFS 呈现了文件系统的名字空间,用户能够以文件的形式在上面存储数据。从内部看,一个文件其实被分成一个或多个数据块,这些块存储在一组 DataNode 上。NameNode 执行文件系统的名字空间操作,例如打开、关闭、重命名文件或目录。它也负责确定数据块到具体 DataNode 节点的映射。DataNode 负责处理文件系统客户端的 I/O 请求,在 NameNode 的统一调度下进行数据块的创建、删除和复制。HDFS 架构如图 5-16 所示。

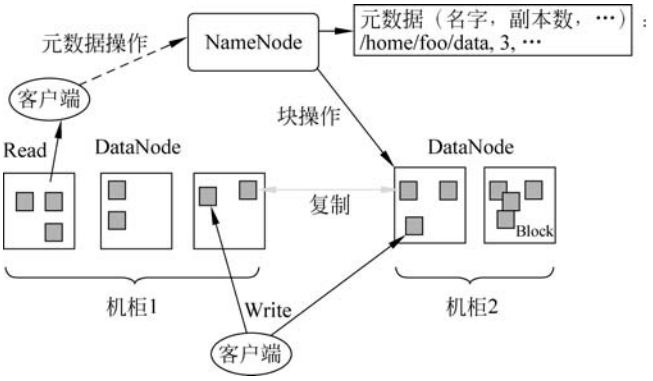

图 5-16 HDFS 架构

NameNode 和 DataNode 被设计成可以在普通的商用机器上运行,这些机器一般运行 GNU/Linux 操作系统。

HDFS 采用 Java 语言开发,因此任何支持 Java 的机器都可以部署 NameNode 或 DataNode。由于采用了可移植性极强的 Java 语言,使得 HDFS 可以部署到多种类型的机器上。一个典型的部署场景是一台机器上只运行一个 NameNode 实例,而集群中的其

他机器分别运行一个 DataNode 实例。这种架构并不排斥在一台机器上运行多个 DataNode,但是这样的情况比较少见。

客户端访问 HDFS 中文件的流程如下。

(1) 从 NameNode 获得组成这个文件的数据块位置列表。

(2) 根据位置列表得到储存数据块的 DataNode。

(3) 访问 DataNode 获取数据。

HDFS 保证数据存储可靠性的机理如下。

(1) 冗余副本策略:所有数据都有副本,对于副本的数目可以在 hdfs-site.xml 中设置相应的副本因子。

(2) 机架策略:采用一种"机架感知"相关策略,一般在本机架存放一个副本,在其他机架再存放别的副本,这样可以防止机架失效时丢失数据,也可以提高带宽利用率。

(3) 心跳机制:NameNode 周期性地从 DataNode 接收心跳信号和块报告,没有按时发送心跳的 DataNode 会被标记为宕机,不会再给任何 I/O 请求,若是 DataNode 失效造成副本数量下降,并且低于预先设置的阈值,NameNode 会检测出这些数据块,并在合适的时机进行重新复制。

(4) 安全模式:NameNode 启动时会先经过一个"安全模式"阶段。

(5) 校验和:客户端获取数据通过检查校验和发现数据块是否损坏,从而确定是否要读取副本。

(6) 回收站:删除文件会先到回收站,其里面的文件可以快速恢复。

(7) 元数据保护:映像文件和事务日志是 NameNode 的核心数据,可以配置为拥有多个副本。

(8) 快照:支持存储某个时间点的映像,需要时可以使数据重返这个时间点的状态。

5.5 分布式数据库 NoSQL

5.5.1 NoSQL 数据库概述

NoSQL 泛指非关系型数据库,相对于传统关系型数据库,NoSQL 有着更复杂的分类,包括 KV 数据库、文档数据库、列式数据库以及图数据库等。这些类型的数据库能够更好地适应复杂类型的海量数据存储。

一个 NoSQL 数据库提供了一种存储和检索数据的方法,该方法不同于传统关系型数据库的表格形式。NoSQL 形式的数据库从 20 世纪 60 年代后期开始出现,直到 21 世纪早期,伴随着 Web 2.0 技术的不断发展,其中以互联网公司为代表,如 Google、Amazon 和 Facebook 等公司,带动了 NoSQL 这个名字的出现。目前 NoSQL 在大数据领域的应用非常广泛,例如实时 Web 应用。

促进 NoSQL 发展的因素如下。

(1) 简单设计原则可以更简单地水平扩展到多机器集群。

(2) 更细粒度地控制有效性。

每一种 NoSQL 数据库的有效性取决于该类型 NoSQL 所能解决的问题。而大多数

NoSQL 数据库系统为了提高系统的有效性、分区容忍性和操作速度都选择降低系统一致性。

较低的系统一致性就意味着标准接口的匮乏和查询语句的原始。这制约了当前 NoSQL 数据库系统的发展。这与传统关系型数据库系统的完整和体系化形成了对比。

目前大多数 NoSQL 提供了最终一致性,也就是数据库的更改最终会传递到所有节点上。表 5-3 是常用的 NoSQL 列表。

表 5-3 常用的 NoSQL 列表

类 型	实 例
Key-Value Cache	Infinispan、Memcached、Repcached、Terracotta、Velocity
Key-Value Store	Flare、Keyspace、RAMCloud、SchemaFree、Hyperdex、Aerospike
Data-Structures Server	Redis
Document Store	Clusterpoint、Couchbase、CouchDB、DocumentDB、Lotus Notes、MarkLogic、MongoDB
Object Database	DB4O、Objectivity/DB、Perst、Shoal、ZopeDB

5.5.2 KV 数据库

KV 数据库是最常见的 NoSQL 数据库形式,其优势是处理速度非常快,缺点是只能通过完全一致的键(Key)查询获取数据。根据数据的保存形式,键值存储可以分为临时性和永久性,下面介绍两者兼具的 KV 数据库 Redis。

Redis 是著名的内存 KV 数据库,在工业界得到了广泛的使用。它不仅支持基本的数据类型,也支持列表和集合等复杂的数据结构,因此拥有较强的表达能力,同时又有非常高的 I/O 效率。Redis 支持主从同步,数据可以从主服务器向任意数量的从服务器上同步,从服务器可以是关联其他从服务器的主服务器,这使得 Redis 可以执行单层树复制。由于完全实现了发布/订阅机制,使得从数据库在任何地方同步树时可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

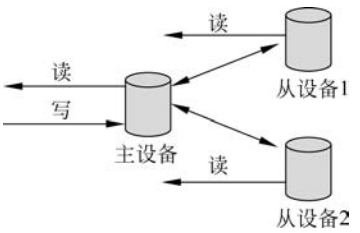

图 5-17 Redis 的副本维护策略

对于内存数据库而言,最为关键的一点是如何保证数据的高可用性,应该说 Redis 在发展过程中更强调系统的 I/O 性能和使用便捷性,在高可用性方面一直不太理想。

如图 5-17 所示,系统中有唯一的主设备(Master)负责数据的 I/O 操作,可以有多个从设备(Slave)保存数据副本,数据副本只能读取不能更新。Slave 初次启动时从 Master 获取数据,在数据复制过程中 Master 是非阻塞的,即可以同时支持 I/O 操作。Master 采取快照结合增量的方式记录即时起新增的数据操作,在 Slave 就绪之后以命令流的形式传给 Slave,Slave 顺序执行命令流,这样就达到 Slave 和 Master 的数据同步。

由于 Redis 采用这种异步的主从复制方式,所以 Master 接收数据更新操作到 Slave

更新数据副本有一个时间差，如果 Master 发生故障可能导致数据丢失。而且 Redis 并未支持主从自动切换，如果 Master 故障，此时系统表现为只读，不能写入。由此可以看出 Redis 的数据可用性保障还是有缺陷的，那么在现版本下如何实现系统的高可用呢？一种常见的思路是使用 Keepalived 结合虚拟 IP 实现 Redis 的高可用（High Availability，HA）方案。Keepalived 是软件路由系统，主要目的是为应用系统提供简洁强壮的负载均衡方案和通用的高可用方案。使用 Keepalived 实现 Redis 的 HA 方案如下。

（1）在两台（或多台）服务器上分别安装 Redis 并设置主从。

（2）Keepalived 配置虚拟 IP 和两台 Redis 服务器的 IP 的映射关系，这样对外统一采用虚拟 IP，而虚拟 IP 和真实 IP 的映射关系及故障切换由 Keepalived 负责。有 3 种情况：当 Redis 服务器都正常时，数据请求由 Master 负责，Slave 只需要从 Master 复制数据；当 Master 发生故障时，Slave 接管数据请求并关闭主从复制功能，以避免 Master 再次启动后 Slave 数据被清掉；当 Master 恢复正常后，首先从 Slave 同步数据以获取最新的数据情况，关闭主从复制并恢复 Master 身份，与此同时 Slave 恢复其 Slave 身份。

5.5.3 列式数据库

列式数据库基于列式存储的文件存储格局，兼具 NoSQL 和传统数据库的一些优点，具有很强的水平扩展能力、极强的容错性以及极高的数据承载能力，同时也有接近传统关系型数据库的数据模型，在数据表达能力上强于简单的 KV 数据库。

下面以 BigTable 和 HBase 为例介绍列式数据库的功能和应用。

BigTable 是 Google 公司设计的分布式数据存储系统，针对海量结构化或半结构化的数据，以 GFS 为基础，建立了数据的结构化解释，其数据模型与应用更贴近。目前 BigTable 已经在超过 60 个 Google 产品和项目中得到应用，其中包括 Google Analysis、Google Finance、Orkut 和 Google Earth 等。

BigTable 的数据模型本质上是一个三维映射表，其最基础的存储单元由行主键、列主键和时间构成的三维主键唯一确定。BigTable 中的列主键包含两级，其中第一级被称为列簇（Column Families），第二级被称为列限定符（Column Qualifier），两者共同构成一个列的主键。

在 BigTable 内可以保留随着时间变化的不同版本的同一信息，这个不同版本由时间戳维度进行区分和表达。

HBase 是一个开源的非关系型分布式数据库，它参考了 Google 的 BigTable 模型，实现的编程语言为 Java。它是 Apache 软件基金会的 Hadoop 项目的一部分，运行在 HDFS 文件系统上，为 Hadoop 提供类似 BigTable 规模的服务。因此，它可以容错地存储海量稀疏的数据。HBase 在列上实现了 BigTable 论文提到的压缩算法、内存操作和布隆过滤器（Bloom Filter）。HBase 的表能够作为 MapReduce 任务的输入和输出，可以通过 Java API 访问数据，也可以通过 REST、Avro 或者 Thrift 的 API 访问。HBase 的整体架构如图 5-18 所示。

HBase 以表的形式存储数据。表由行和列组成，每个列属于某个列簇，由行和列确定的存储单元称为元素，每个元素保存同一份数据的多个版本，由时间戳标识区分，如表 5-4 所示。

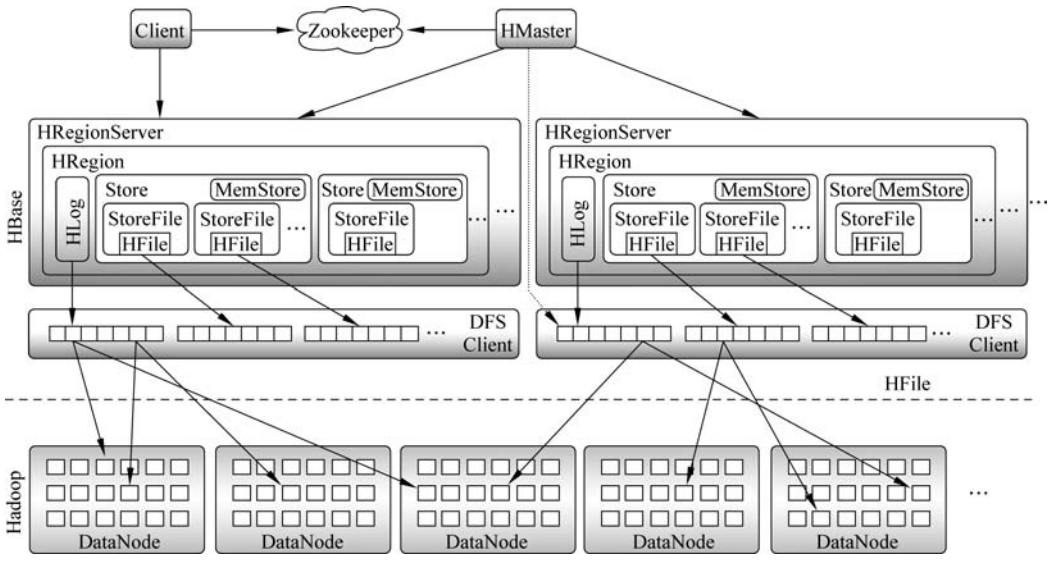

图 5-18 HBase 存储架构图

表 5-4 HBase 存储结构

行键	时间戳	列"contents:"	列"anchor:"		列"mine:"
"com. cnn. www"	t9		"anchor:cnnsi. com"	"CNN"	
	t8		"anchor:my. look. ca"	"CNN. com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

5.6 HBase 数据库搭建与使用

HBase 是分布式 NoSQL 系统中可扩展的列式数据库,支持随机读写和实时访问,能够存储非常大的数据库表(例如表可以有数十亿行和上百万列)。下面简要介绍 HBase 的搭建与使用。

5.6.1 HBase 伪分布式运行

因为 HBase 是运行在 HDFS 的基础上的,所以需要先启动 HDFS 集群。这里首先运行的是 HBase 伪分布式版本,所以 HDFS 也采用伪分布式版本。

1. HDFS 集群启动

HDFS 的核心配置文件 hdfs-site. xml 配置文件如图 5-19 所示。

格式化 HDFS 文件系统,输入如下命令:

```
<configuration>
<property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

图 5-19　hdfs-site.xml 配置文件

```
1    ./bin/hdfs namenode -format
```

启动 HDFS 文件系统，输入如下命令：

```
1    ./sbin/start-dfs.sh
```

通过网页形式查看，若显示页面如图 5-20 所示则表明启动成功。

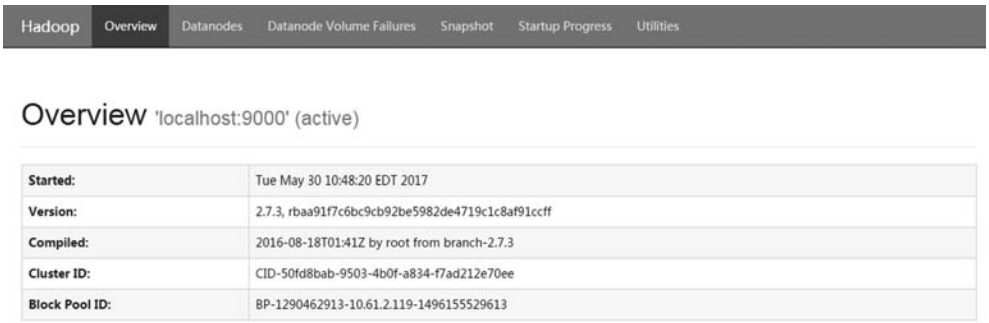

图 5-20　HDFS 集群显示页面

2. Zookeeper 启动

HBase 启动需要 Zookeeper 支持，使用最简单的 Zookeeper 配置，下载 Zookeeper 的运行包，下载地址为 http://www-us.apache.org/dist/zookeeper/zookeeper-3.4.9/。

配置 Zookeeper，执行如下命令：

```
1    cp conf/zoo_sample.cfg conf/zoo.cfg
```

启动 Zookeeper，执行如下命令：

```
1    /bin/zkServer.sh start
```

3. HBase 集群启动

下载 HBase 运行 jar 包，HBase 需要与 Hadoop 兼容，这里 Hadoop 的版本是 2.7.3，HBase 的版本是 1.2.4，下载地址是 http://archive.apache.org/dist/hbase/1.2.4/。

hbase-site.xml 配置文件如图 5-21 所示。

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://localhost:9000/hbase</value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
</configuration>
```

图 5-21　hbase-site.xml 配置文件

启动如下命令,运行 HBase 集群:

```
1    ./bin/start-hbase.sh
```

利用 HBase 的页面显示,查看运行状态,在浏览器中输入服务器 IP 地址加上端口号 16010,显示如图 5-22 所示。

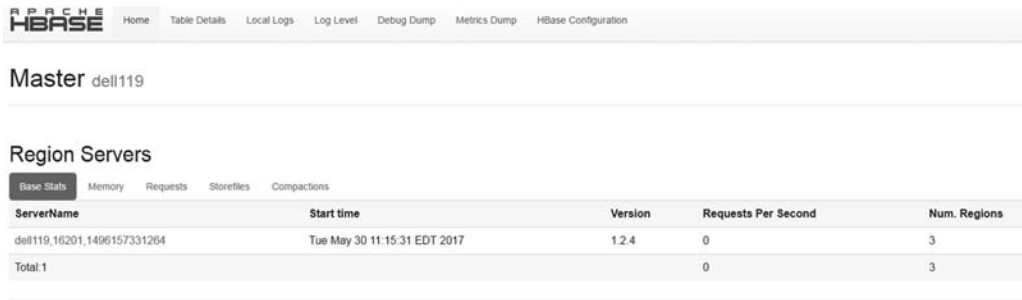

图 5-22　HBase 伪分布式运行状态

5.6.2　HBase 分布式运行

HBase 分布式版本与伪分布式版本配置过程差不多,也是分为 HDFS 集群启动、Zookeeper 启动和 HBase 集群启动三部分。

1. HDFS 集群启动

这里用作 HDFS 集群的机器数目一共为 4 台,1 台当作 NameNode 节点,其他 3 台当作 DataNode 节点。

如图 5-23 所示,是 HDFS 集群的 hdfs-site.xml 配置文件。图中 Hadoop 文件的路径名称由于包含个人信息,所以进行了遮盖。读者应填写个人系统中 Hadoop 文件的存储路径。

启动 HDFS 集群,输入如下命令:

```
1    ./sbin/start-dfs.sh
```

利用 Web 界面查看 HDFS 运行状态,如图 5-24 所示。

```
<configuration>
<property>
        <name>dfs.replication</name>
        <value>3</value>
    </property>
    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>dell118:50090</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/home/[illegible]/opt/hadoop/tmp/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/home/[illegible]/opt/hadoop/tmp/dfs/data</value>
    </property>

</configuration>
```

图 5-23　hdfs-site.xml 配置文件

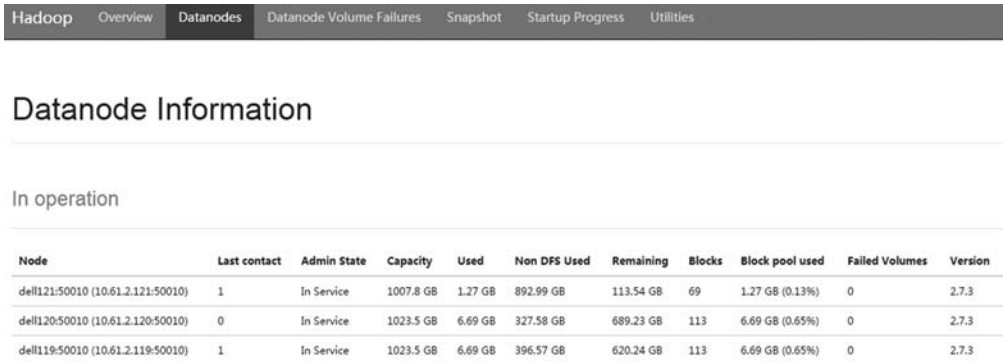

图 5-24　HDFS 集群运行状态

2. Zookeeper 启动

HBase 启动需要 Zookeeper 支持，配置 Zookeeper，修改 zoo.cfg 文件，具体配置如图 5-25 所示。

```
# the port at which the clients will connect
clientPort=2181
server.1=dell118:2888:3888
server.2=dell119:2888:3888
server.3=dell120:2888:3888
```

图 5-25　Zookeeper 集群配置

启动 Zookeeper，执行如下命令：

```
./bin/zkServer.sh start
```

3. HBase 集群启动

hbase-site.xml 配置文件如图 5-26 所示。

启动如下命令，运行 HBase 集群：

```
        <property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>dell118,dell119,dell120</value>
</property>
<property>
    <name>hbase.regionserver.lease.period</name>
    <value>240000</value>
</property>
<property>
    <name>hbase.rpc.timeout</name>
    <value>280000</value>
</property>
<property>
        <name>zookeeper.session.timeout</name>
        <value>120000</value>
</property>
```

图 5-26　hbase-site.xml 配置文件

```
1    /bin/start-hbase.sh
```

利用 HBase 的页面显示,查看运行状态,在浏览器中输入服务器 IP 地址加上端口号 16010,显示如图 5-27 所示。

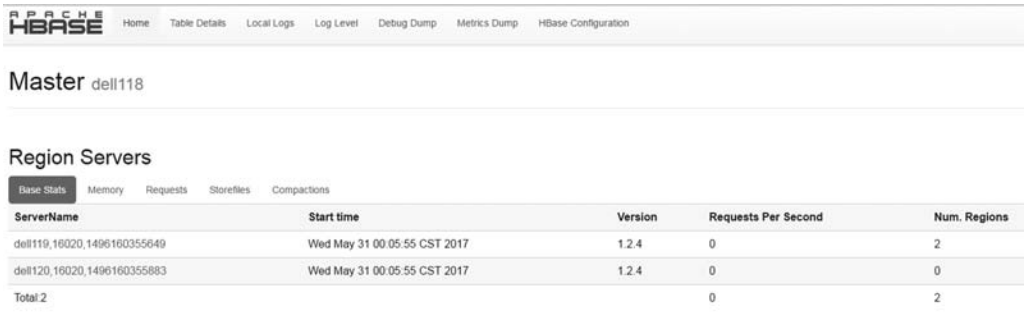

图 5-27　HBase 集群运行状态