第8章　附录 英文文献翻译

Hadoop 分布式文件系统

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler

Yahoo!

Sunnyvale, California USA

{Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com

摘要

Hadoop 分布式文件系统(HDFS)设计用于为大规模数据集提供可靠性的存储，同时能够将数据集以高带宽的传输速率推送给用户应用程序。在一个大规模集群上，将会有数千台的服务器同时负责数据存储及执行用户应用级的计算任务。通过将存储和计算分布到很多个服务器上，使得存储和计算资源可以在保持低成本的情况下根据数据规模按需增长。在本文中，我们会描述下 HDFS 的架构，以及我们在 Yahoo!使用 HDFS 来管理 25PB 的企业数据的相关经验。

1.简介及相关工作

Hadoop[1][16][19]提供了一个分布式文件系统及一个使用 MapReduce[3]范式进行大规模数据集分析和转换的框架。Hadoop 的一个重要特点是，将数据和计算划分在很多(数千台)主机上，同时直接在这些数据附近并行执行计算应用{!即存储数据的跟执行计算的是同一个节点集合，这就可以很容易地将计算移动到数据附近执行}。一个 Hadoop 集群可以简单地通过增加服务器来对计算能力、存储能力及 IO 带宽进行扩展。Yahoo!的 Hadoop 集群目前已包含 25000 台服务器，存储了 25PB 的应用数据，最大的集群目前包含 3500 台服务器。目前世界上已有上百个组织宣布他们采用了 Hadoop。

HDFS	Distributed file system Subject of this paper!
MapReduce	Distributed computation framework
HBase	Column-oriented table service
Pig	Dataflow language and parallel execution framework
Hive	Data warehouse infrastructure
Zookeeper	Distributed coordination service

Chukwa	System for collecting management data
Avro	Data serialization system

Table 1. Hadoop project components

Hadoop 是一个 Apache 项目；所有的组件都遵循 Apache 开源许可证。在 Hadoop 核心组件(HDFS 和 MapReduce)中，其中 80%都是由 Yahoo!开发和贡献的。HBase 最初是在 Powerset 开发的，现在它已经是微软的一个部门。Hive[15]最初由 Facebook 开发。Pig[4]，ZooKeeper[6]，及 Chukwa 都是由 Yahoo!发起并开发的。Avro 也是源自 Yahoo!，目前 Cloudera 也在参与它的开发。

HDFS 是 Hadoop 的文件系统组件。它的接口类似于 Unix 文件系统，但是为了提高应用程序性能，它并没有严格遵从标准。

HDFS 将文件系统元数据和应用程序数据分开存储。像其他的一些分布式系统比如[2][14], Lustre [7]及 GFS [5][8]一样，HDFS 将元数据存放在一个被称作 NameNode 的专门的服务器上。应用数据则被存储在称作 DataNode 的其他服务器上。所有的服务器都是相通的，相互之间通过基于 TCP 的协议进行通信。

与 PVFS 和 Lustre 不同，HDFS 的 DataNodes 没有使用像 ACID 这样的数据保护机制来保证数据的持久性。而是像 GFS 那样，通过将文件内容复制到多个 DataNodes 上来保证可靠性。在保证数据持久性的同时，这种方式也带来了一些额外的好处，比如数据传输带宽变成了原来的几倍，同时提高了将计算移动到数据附近的可能性(locality)。

一些分布式文件系统目前也在探索一些名字空间的真正的分布式实现方式。比如 Ceph 使用一个具有多个名字空间服务器(MDS-MetaDataServer)的集群，同时使用一个动态的子树划分算法来将名字空间树均匀地映射到 MDSs 上。GFS 也已经演化成一个分布式名字空间[8]的实现。新一代的 GFS 将会具有数百个名字空间服务器(masters)，其中的每个都能管理 100 million 的文件。Lustre[7]在 2.2 版中，已经具有一个集群化的名字空间实现。目的就是为了将一个目录划分到多个元数据服务器(MDS)，让每个服务器负责名字空间的一部分。文件会通过对文件名称使用一个 hash 函数来分配到特定的 MDS 上。

2.架构

2.1NameNode

HDFS 名字空间是一个由文件和目录组成的层次性结构。在 NameNode 上，文件和目录通过 inode 标识，每个 inode 会记录像访问权限、修改信息、访问时间、名字空间及磁盘空间 qutas 这样的一些属性。文件内容会被切分成很多大的 blocks(通常是 128MB，用户可以为每个文件设定自己的 block 大小)同时组成文件的每个 block 会被复制到多个 DataNodes 上(通常是 3，用户也可以为每个文件设定自己的副本数)。NameNode 维护一个名字空间树及文件 blocks 到 DataNodes 的映射信息(即文件数据的物理位置)。当 HDFS Client 想读取文件时，必须与 NameNode 联系以获取组成该文件的 blocks 的位置信息，然后选择一个离它最近的 DataNode 去读取 block 内容。在写数据时，client 向 NameNode 发送请求，让它指定应该由哪三个 DataNodes 来保存该 block 的三个副本。之后客户端就会以 pipeline 的模式将数据写入到 DataNodes。当前的设计中，每个集群中只有一个 NameNode。但是每个集群可以有数千个 DataNodes 及数万个 HDFS clients，因为每个 DataNode 可能会同时执行多个应用程序任务{!所以 HDFS clients 的数目可能比 DataNodes 多个数量级}。

HDFS 会将整个名字空间保存在内存中。由 inode 数据及每个文件包含的所有 blocks 列表组成的名字系统元数据叫做 image。保存在本机本地文件系统中的该 image 的一个持久化记录称为一个 checkpoint。NameNode 也会将称为 journal 的针对该 image 的修改日志保存到本机本地文件系统中。为了提高持久性，可以在其他服务器上保存 checkpoint 和 journal 的多个副本。在重启的时候，NameNode 会通过读取名字空间 checkpoint 及重放 journal 来恢复名字空间。Block 副本位置信息可能会随着时间而改变，同时它们也不是持久化的 checkpoint 的组成部分。

2.2DataNode

DataNode 中的每个 block 副本由本机本地文件系统中的两个文件组成。第一个文件包含数据本身，第二个文件是该 block 的元数据包括该 block 数据的校验和及该 block 的世代戳(generation stamp)。数据文件大小等于该 block 的实际长度，同时不需要补上额外的空间以达到标准的块大小{!比如该 block 只有 10MB，那么本地文件系统中的数据文件大小就是 10MB，而无需在额外补足让它变成标准的 128MB}。因此，如果一个 block 只有标准大小的一半，那么本地磁盘也只需要半个标准 block 所需的空间。

在每个 DataNode 启动时，它会连接到 NameNode 执行一个握手。握手的目的是为了验证名字空间 ID 及 DataNode 的软件版本。如果其中只要有一个无法与 NameNode 匹配，那么 DataNode 会自动关闭。

名字空间 ID 是在文件系统创建时分配给它的实例编号。名字空间 ID 会持久化存储在集群的所有节点中。具有不同名字空间 ID 的节点无法加入到集群中，这就保护了文件系统的数据完整性。

软件版本的一致性是非常重要的，因为不兼容的版本可能会导致数据损坏或丢失，同时在一个具有数千个节点的大规模集群上，很容易会在升级期间忽略掉某些节点，比如它没有在升级之前正确的关闭或者在升级时处于不可用的状态。

允许一个新初始化的并且没有任何名字空间 ID 的 DataNode 加入到集群中，它会接受集群的名字空间 ID。{!这是因为很多情况下我们需要对集群进行扩容，因此 HDFS 应该允许我们往集群中添加新机器}

在握手过程完成之后，DataNode 会与 NameNode 进行注册。DataNodes 会持久化存储它们自己对应的那个唯一的存储 ID。存储 ID 是 DataNode 的内部标识符，可以保证即使是它更换了 IP 地址或者端口也能识别出来。存储 ID 是在 DataNode 第一次向 NameNode 进行注册时分配的，之后它就再也不会改变。

DataNode 会通过向 NameNode 发送一个 block report 来声明它所拥有的 block 副本。一个 block report 包含该 block 的 id，世代戳(generation stamp)以及它所持有的 block 副本长度。当 DataNode 注册完成之后就会立即发送第一次的 block report。之后，会每隔 1 小时就进行一次 block reports 发送，从而为 NameNode 提供关于该集群内的所有 block 副本的最新位置信息。

在正常情况下，DataNode 会向 NameNode 发送心跳信息以证实它自己正在运行以及它所持有的 block 副本是可用的。默认的心跳周期是 3 秒钟。如果 NameNode 在十分钟内收不到来自某个 DataNode 的心跳信息，它会认为该 DataNode 已经不能提供服务，它所持有的 block 副本就变成了不可用状态。NameNode 就会将这些 block 副本在其他 DataNode 上创建出来。

来自 DataNode 的心跳中还会携带一些关于总的存储容量、存储空间使用量及当前正在处理的数据传输量方面的信息。这些统计信息会被用于 NameNode 的空间分配及负载平衡决定中。

NameNode 不会直接联系 DataNode，它会通过对心跳的响应信息来向 DataNodes 发送指令。这些指令包括如下一些命令：

- 复制 blocks 到其他节点
- 删除本地的 block 副本
- 重新注册或者关闭节点
- 发送一个即时 block report

这些命令对于维护整个系统的完整性是十分重要的，因此就算是在大规模的集群中，保持心跳的通畅也是非常重要的。NameNode 每秒可以处理数千个心跳请求而不会影响到其他的 NameNode 操作。

2.3HDFS Client

用户应用程序通过 HDFS Client(一个包含 HDFS 文件系统接口的代码库)来访问文件系统。

类似于大部分的传统文件系统，HDFS 支持文件的读写和删除操作，以及针对目录的创建和删除操作。用户通过名字空间里的路径来访问文件和目录。用户应用程序通常并不需要知道文件系统元数据和数据存储是位于不同的服务器上的，或者是一个 block 是有多个副本的。

当一个应用程序读取一个文件时，HDFS client 首先向 NameNode 询问持有组成该文件的 blocks 的 DataNodes 列表。然后直接联系某个 DataNode 请求对于它所需要的 block 的传输。当 client 进行写的时候，它会首先让 NameNode 选定持有该文件的第一个 block 的那些 DataNodes。客户端会把这些节点组织成一个 pipeline，然后发送数据。当第一个 block 写出后，客户端会继续请求选定持有下一个 block 的新的 DataNodes。新的 pipeline 会被建立起来，客户端开始发送该文件后面的那些数据。每次选定的 DataNodes 可能是不同的。NameNode 和 DataNodes 与客户端的交互如图 1 所示。

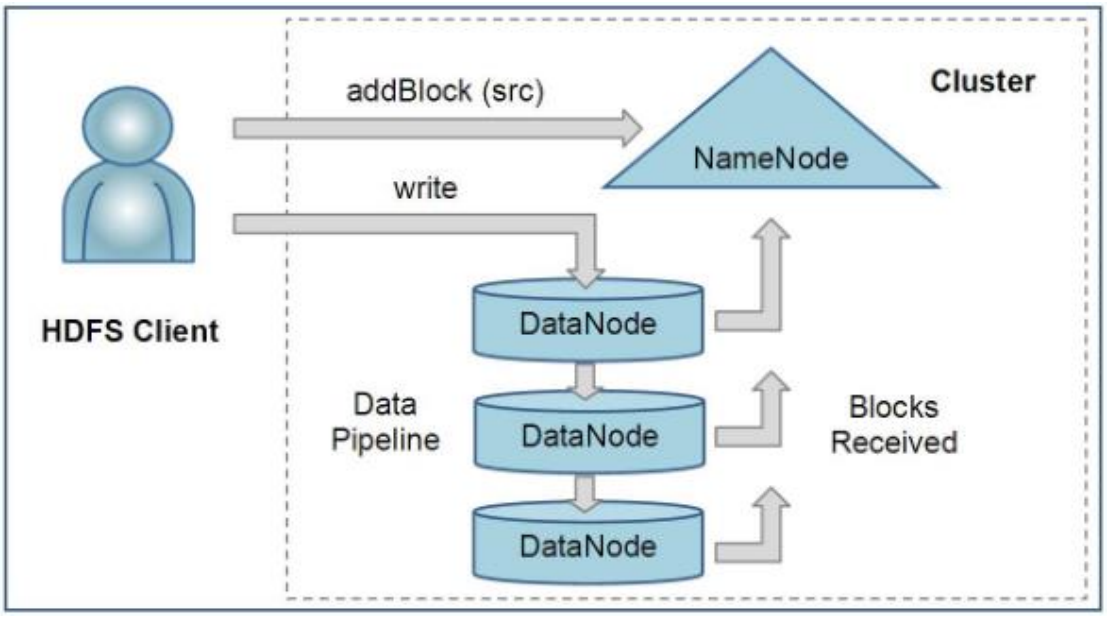

图 1.一个 HDFS 客户端通过向 NameNode 传输 path 创建一个新文件。对于该文件的每个 block，NameNode 返回持有它的副本的那些 DataNodes 列表。客户端然后将数据通过 pipeline 的形式传送给选定的 DataNodes，DataNodes 最终会再联系 NameNode 对 block 各副本的创建情况进行确认。

与传统文件系统不同，HDFS 提供了一个 API 用于提供某个文件的 blocks 的位置信息。这就允许应用程序比如 MapReduce 框架可以将 task 调度到数据所在的节点上，这就提高了读性能。同时它也允许应用程序对文件的副本数进行设置。默认情况下，文件的副本数是 3。对于某些重要文件或者是某些经常被访问的文件，可以增大该参数值以提高容错性及读取带宽。

2.4Image 与 Journal

名字空间 image 是代表应用数据的目录和文件组织方式的文件系统元数据。写入到磁盘中的 image 的持久化记录称为 checkpoint{!image 在内存中，checkpoint 则是在磁盘中}。Journal 是一个记录了那些必须被持久化的文件系统变更的 write-ahead commit 日志。对于每个客户端发起的事务，变更会被记录到 journal 中，在变更提交给 HDFS 客户端之前 journal 文件必须被 flush 及 sync。checkpoint 文件永远不会被 NameNode 修改；当在重启时创建好新的 checkpoint 时，或者在管理员或下一节描述的 CheckpointNode 发出请求时，它会被整个替换掉。在 NameNode 启动时，会根据 checkpoint 初始化名字空间 image，然后重放 journal

中的变更直到 image 更新到文件系统的最终状态。在 NameNode 提供服务之前，一个新的 checkpoint 和空的 journal 会被写回到存储目录下。

如果 checkpoint 或者是 journal 丢失或者损坏了，名字空间信息将会部分地或者整个地丢失。为了对关键信息进行保护，可以将 HDFS 配置成将 checkpoint 和 journal 在多个存储目录下存放。推荐性的做法是将这些目录放在不同的逻辑卷上，或者是在远程 NFS 服务器上的某个存储目录下。第一种做法可以防止单个逻辑卷的损坏造成数据丢失，第二种做法可以应付整个节点失败的情况。如果 NameNode 在将 journal 写入到某个存储目录下的过程中出错，那么它会自动地将该目录从存储目录列表中排除。如果没有存储目录可用，NameNode 会自动地停止运行。

NameNode 是一个多线程系统，可以同时处理来自多个客户端的请求。将事务日志保存到磁盘就成了系统的瓶颈，因为所有的线程都必须等到其中某个线程发起的 flush-and-sync 调用结束。为了优化该处理过程，NameNode 会将由不同客户端产生的多个事务批量进行处理。当其中某个 NameNode 线程发起 flush-and-sync 调用时，堆积在此刻的所有事务会一块进行提交。其他线程只需要检查下它们的事务是否被写入了而不需要再发起一个 flush-and-sync 调用。

2.5CheckpointNode

HDFS 中的 NameNode，除了可以担任客户端请求服务者这一首要角色外，还可以担任其他的一些角色比如 CheckpointNode 或者是 BackupNode。节点可以在启动时设置它的角色。

CheckpointNode 会周期性地合并现有的 checkpoint 和 journal，创建一个新的 checkpoint 和一个空的 journal。CheckpointNode 通常运行在与 NameNode 不同的一个节点上，因为它需要与 NameNode 等同的内存空间。它会从 NameNode 下载当前的 checkpoint 和 journal 文件，然后在本地对它们进行合并，然后将新的 checkpoint 返回给 NameNode。

创建周期性的 checkpoints 是保护文件系统元数据的一种方式。如果名字空间 image 的所有持久化拷贝或者 journal 不可用了，系统就可以从最近的那个

checkpoint 处恢复。

当新的 checkpoint 上传到 NameNode 后，checkpoint 的创建需要 NameNode 在 journal 的尾部进行截断{!即此时若要创建 checkpoint 应该截断当前 journal，而新的修改日志应该写入到新的 journal 中，当前的 journal 会跟旧的 checkpoint 一起用于新 checkpoint 的创建}。HDFS 集群如果长期运行而不重启的话，那么在此期间 journal 会持续增长。如果 journal 变得很大的话，那么 journal 文件发生数据丢失或损坏的概率就会上升。同时，一个很大的 journal 文件也会增加 NameNode 重启所需的时间。对于一个大规模的集群来说，可能会花一个小时来处理一个已存在一周的 journal。因此最好每天都进行 checkpoint 的创建。

2.6BackupNode

BackupNode 是 HDFS 最近引入的一个 feature。与 CheckpointNode 类似，BackupNode 能够创建周期性的 checkpoints，但是除此之外它还在内存中维护了一个文件系统名字空间的最新映像，该映像会一直与 NameNode 状态保持同步。

BackupNode 会接受来自处于活动状态的那个 NameNode 的名字空间事务形成的 journal 流，它会将它们存放到自己的存储目录下，同时将这些事务应用到它自己的内存映像中。NameNode 会像对待存储它的 journal 文件的存储目录那样，将 BackupNode 作为它的一个 journal 存储目标。如果 NameNode 出错了，那么 BackupNode 的内存映像以及磁盘上 checkpoint 都记录了最新的名字空间状态{!即 BackupNode 的内存映像已经是最新的名字空间，checkpoint+journal 也可以用来恢复名字空间状态}。

BackupNode 可以不用从处于活动状态的 NameNode 下载 checkpoint 和 journal 文件就能创建一个 checkpoint，因为它的内存中已经具有了最新的名字空间状态。这使得在 BackupNode 上的 checkpoint 处理更高效，因为它只需要将名字空间保存到本地的存储目录下。

BackupNode 可以看做是一个只读的 NameNode。它包含除 block 位置信息之外的所有文件系统元数据信息。除去那些会引入名字空间改变或者是需要了解 block 位置信息的操作之外，它也可以执行其他所有的常规 NameNode 操作。通

过将名字空间状态的持久化授权让 BackupNode 处理，这样 BackupNode 的使用就提供一种不需要持久化存储及名字空间授权的 NameNode 运行选择{!即 NameNode 运行时可以自己不进行持久化存储了，而让 BackupNode 来负责，这就降低了 NameNode 的负载}。

2.7 升级，文件系统快照

在软件升级期间，由软件 bug 或者人为失误导致的系统崩溃概率会上升。在 HDFS 中创建快照的目的是为了最小化系统升级期间对存储的数据的潜在威胁。

快照机制使得管理员可以将文件系统的当前状态进行持久化保存，这样如果升级导致数据损坏或丢失时，可以对升级进行回滚，使得 HDFS 回到快照创建时的名字空间和存储状态。

快照(只能有一个)可以通过集群管理员配置进行创建，而不管系统是何时启动的。当接受到快照请求后，NameNode 会首先读取 checkpoint 和 journal 文件，然后在内存中合并它们。然后，它会写出一个新的 checkpoint 及空的 journal 到一个新的位置，这样旧的 checkpoint 和 journal 文件就仍然是保持不变的。

在握手期间，NameNode 会向 DataNodes 发出一个创建本地快照的命令。本地快照不能通过简单地对目录下文件进行复制来实现，因为这会导致集群中所有 DataNodes 节点的存储空间加倍。每个 DataNodes 不是真正创建存储目录的一份拷贝，而是为现有的 block 文件创建出硬链接到存储目录下。当 DataNodes 删除一个 block 时，它只是删除了这个硬链接，当 append 操作导致 block 内容改变时会采用 copy-on-write 技术。因此老的目录中的老的 block 文件依然是处于未改变的状态。

集群管理员可以在重启系统时选择让 HDFS 回滚到快照状态。NameNode 会使用快照创建时保存的那个 checkpoint 进行恢复。DataNodes 会恢复之前被重命名的目录，同时启动一个后台线程去删除在快照之后创建的 block 副本。一旦选择了回滚，就不能在退回到之前的状态了。集群管理员也可以通过命令系统丢弃快照来释放由快照所占用的空间，然后完成软件升级。

系统的演化可能会导致 NameNode 的 checkpoint 和 journal 文件格式或者是 DataNodes 上的 block 副本文件的数据表示方式发生变化。Layout version 会被用来标识数据表示格式，它会被持久化地保存到 NameNode 和 DataNodes 的存储目

录中。在启动时，每个节点都会将当前软件的 Layout version 与存储在存储目录下的版本进行比较，并自动地将数据从旧的格式转换为新的。当系统使用新的 Layout version 重启时，该转换会强制性地创建一个快照。{!升级分很多种，对于普通的升级，可以让管理员手动选择是否开启 snapshot，但是对于这种涉及到 Layout version 变更的情况，系统会强制性的进行 snapshot}

HDFS 并没有区分是 NameNode 还是 DataNodes 的 Layout versions{!也就是说无论是 NameNode 还是 DataNodes 发生了 Layout versions 的改变，系统都会认为发生了改变，而进行相同的处理}，因为 snapshot 的创建是整个集群层面的事情而不是单个节点级的事件。如果升级后的 NameNode 因为一个软件 bug 而清除了它的 image，那么如果只是备份了名字空间状态仍然会导致所有数据的丢失，因为 NameNode 无法识别 DataNodes 报告的 blocks，就会发出一个删除命令。在这种情况下回滚虽然恢复了元数据，但是数据本身还是丢失了。

3.文件 IO 操作及 Replica 管理

3.1 文件读操作与写操作

应用程序会通过创建新文件然后向文件写入数据来向 HDFS 添加数据。在文件关闭之后，已写入的字节串就不能被改变或删除，但是可以通过重新打开该文件通过 append 操作为该文件增加新数据。HDFS 实现了一个单写者，多读者模型。

HDFS 客户端打开一个文件用于写操作时会被授予该文件的租约；这样其他的客户端就不能再对该文件进行写入。正在进行写入的那个客户端会通过向 NameNode 发送的心跳信息周期性的更新该租约。当该文件被关闭时，租约就会被释放。租约持续时间通过一个 soft limit 和 hard limt 进行限定。在 soft limit 过期之前，写者肯定会独占针对该文件的访问。如果 soft limit 过期了，而客户端没有成功的关闭该文件或者更新该租约，另一个客户端将会优先获取到该租约。如果 hard limt 过期(1 小时)，同时客户端仍未能成功更新该租约，HDFS 会假设该客户端退出了同时会代替该写者自动地关闭该文件，然后释放该租约。写者租约不会阻止其他客户端读取该文件；一个文件可能具有多个并发读者。

一个 HDFS 文件是由多个 blocks 组成。当存在一个新 block 请求时，NameNode 会分配一个具有唯一 block ID 的 block，然后确定用于保存该块的多个副本的 DataNodes 列表。DataNodes 会组成一个 pipeline，它们的排列顺序会尽量的最小

化从客户端到最后一个 DataNode 的总的网络距离。然后数据会以一系列的 packets 的形式推送到该 pipeline 中。应用程序写入的数据会首先缓存在客户端的一个 packet 缓存中。当一个 packet buffer 被填满(默认是 64KB 大小)时，数据就会被推送到 pipeline。在收到前面的 packets 的确认信息之前，下一个 packet 就可以被直接推送到 pipeline 中。处于 outstanding 状态的 packets 数目是通过客户端的一个发送窗口大小限制的。

在数据写入到 HDFS 文件之后，在文件关闭之前，HDFS 不提供任何保证以确保新的读者可以看到该数据。如果一个用户应用程序需要这种可见性保证，它可以显式地调用 hflush 操作。这样当前的 packet 会被立即推送到 pipeline 中，而 hflush 操作会等待直到收到来自 pipeline 中的 DataNodes 关于该包成功传输的确认为止。这样在 hflush 操作之前写入的所有数据对于读取者来说就肯定是可见的了。

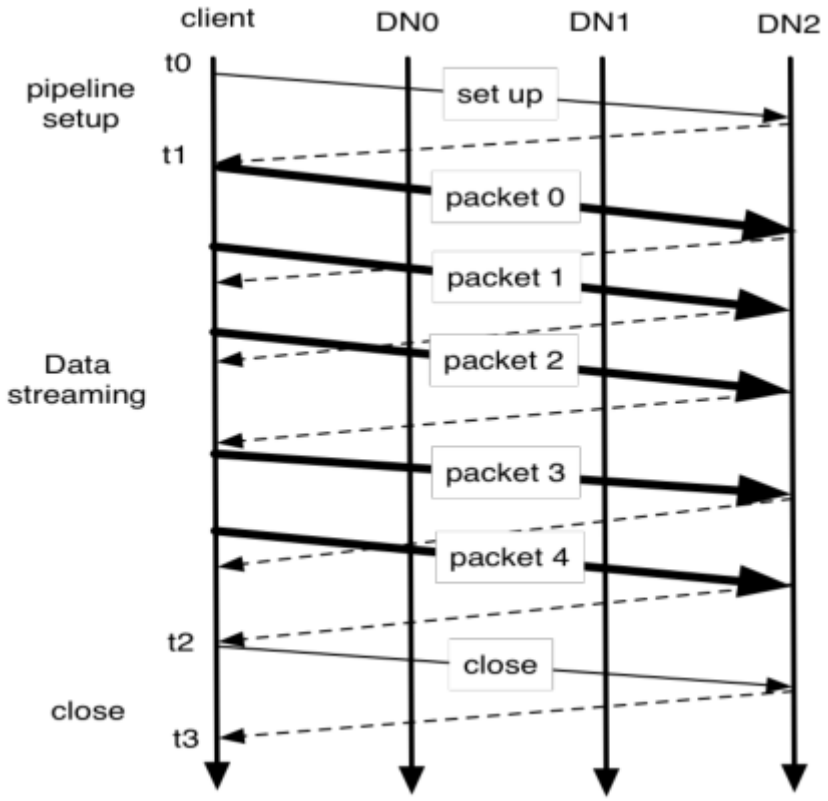

图 2 构造数据块时的管道

如果没有错误发生，块的构建就会经过像图 2 那样的三个阶段。图 2 展示了一个具有三个 DataNodes 的流水线及 5 个 pakctes 的 block。图中，粗线代表了数据包，虚线代表了确认消息，细线代表了用于建立和关闭流水线的控制消息。竖

线代表了客户端及 3 个 DataNodes 的活动，时间流向是自上而下的。从 t0 到 t1 是流水线建立阶段，t1 到 t2 是数据流阶段，t1 代表了第一个数据包被发送的时间点，t2 代表了针对最后一个数据包的确认信息的接收时间点。在这里，在第二个包传输时有一个 hflush 操作。hflush 操作标识是与数据打包在一块的，而不是独立的一个操作。最后，t2 到 t3 是针对该 block 的 pipeline 关闭阶段。

在一个具有数千个节点的集群中，节点失败(通常都是存储系统错误)每天都会发生。因此存储在某个 DataNode 上的副本可能会因为一个内存，磁盘或网络问题而损坏。HDFS 会生成并存储针对 HDFS 文件中每个 block 的校验和。校验和在 HDFS 客户端读取时会进行验证，以检测因客户端，DataNodes 或者是网络导致的损坏。当客户端创建一个 HDFS 文件时，它会为每个 block 计算校验和，然后将它与实际数据一块发送给 DataNodes。DataNode 会将校验和存储在与块数据文件独立的一个元数据文件中。在 HDFS 读取一个文件时，每个 block 的数据和校验和会被传送给客户端。客户端会对接受到的数据计算校验和，并验证它实时计算出的校验和与收到的校验和是否匹配。如果不匹配，客户端会告知 NameNode 该副本损坏了，之后会从另一个 DataNode 上获取该 block 的另一份副本。

当客户端打开文件进行读操作时，它会从 NameNode 获取一个 blocks 列表，及关于每个 block 副本的位置信息。每个 block 的位置信息会根据它们与客户端的距离进行排序。在读取 block 的内容时，客户端会首先尝试从最近的那个副本处进行读取。如果这个读取尝试失败了，客户端会继续尝试从序列中的下一个副本处读取。在目标 DataNode 不可用的情况下读取可能会失败，比如该 DataNode 可能不再持有该 block 的副本了，或者在检查校验和时发现副本是损坏的。

HDFS 允许客户端去读取一个已打开的正在用于写操作的文件。在读取正在被写入的文件时，最后一个 block 因为正在被写入因此对于 NameNode 它的实际大小是未知的。在这种情况下，客户端在开始读取内容前可以询问其中某个副本得到其最新的长度。

HDFS 的 IO 设计是为像 MapReduce 这样需要高顺序读写吞吐率的批处理系统特殊优化过。但是，为了支持像 Scribe 这种实时地向 HDFS 进行数据流导入，或者是像 HBase 这种提供对大表格的随机实时性访问的这些应用，还需要花费很

大的精力来提高 HDFS 的读写响应时间。

3.2Block 放置

对于一个大规模集群来说，对所有的节点采用一种平摊的拓扑连接方式可能是不切实际的。通常的做法是将它们分布到多个机柜中。单个机柜中的节点共享一个交换机，机柜之间通过一个或多个核心交换机相连。这样在不同机柜中的节点间的通信需要跨越多个交换机。大多数情况下，相同机柜内节点间的网络带宽要比不同机柜的节点间的网络带宽要高。图 3 描述了一个具有 2 个机柜的集群，每个机柜包含三个节点。

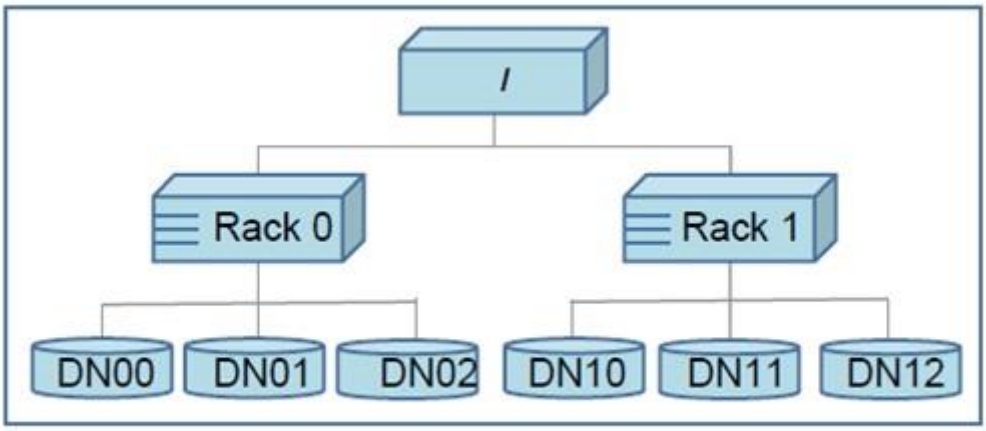

Figure 3. Cluster topology example

HDFS 会根据两个节点间的距离来估算它们的网络带宽。假设从节点到它的父节点间的距离是 1。那么任意两个节点的距离就可以通过将它们到它们的最近公共祖先间的距离求和而得到。距离越短意味着可以用于数据传输的带宽越大。

HDFS 允许管理员安装一个脚本，给定一个节点地址该脚本就可以返回该节点所在的机柜信息。NameNode 会负责解析各个 DataNode 的机柜位置。当 DataNode 向 NameNode 注册时，NameNode 会运行该脚本来确定该 DataNode 属于哪个机柜。如果该脚本没有安装，NameNode 会假设所有 DataNode 都属于默认的同一个机柜中。

副本的放置对于 HDFS 的数据可靠性和读写性能都是至关重要的。一个好的副本放置策略可以提高数据可靠性，可用性及网络带宽利用率。当前的 HDFS 提供一个可配置的块放置策略接口，这样用户和研究人员就可以进行实验测试以为他们的应用选择更好的放置策略。

默认的 HDFS block 放置策略在最小化写开销和最大化数据可靠性、可用性以及总体读取带宽之间进行了一些折中。当一个新的 block 创建时，HDFS 会将第一个副本放置在 writer 本身所在的那个节点上，第二个和第三个副本将会被放到另一个机柜的两个不同节点上，再剩下的就会被随机地放置，但需要保证如下几个条件：一个节点上最多只能放一个副本；如果副本数小于机柜数的 2 倍，那么同一个机柜上最多能放两个副本。我们选择将第二个和第三个副本放到另一个机柜上可以更好的将 blocks 分布到集群上。如果前两个副本被放置在相同的机柜上，那么对于任意文件来说，那么它的三分之二的 blocks 副本都会被放到相同的机柜上{!因为第一个副本已经定了，会被放到 writer 本身所在的那个节点，那么根据这种策略第二个副本也会放到上面，与此同时因为 writer 一直处在该节点上，那么其他 block 也会被这样放置，最后就会导致该文件至少有三分之二的 blocks 副本会被放到 writer 本身所在的那个节点上}。

当所有的目标节点选定之后，这些节点会以它们与第一个副本的接近程度为序组织成一个流水线的形式。对于读取来说，NameNode 会首先判断客户端所在主机是否在集群中，如果是的话，block 的位置信息会以它们与该客户端的接近程度为序返回给客户端。Block 从 DataNodes 中读取时就会参照这个顺序。(这对于那些直接运行在集群内部节点上 MapReduce 很有用，当然了实际上一个主机只要可以连接到 NameNode 和 DataNodes，就可以在它上面运行 HDFS client)

这种策略降低了机柜间及节点间的写流量，提高了写性能。由于单个机柜的失败概率要远低于单个节点的失败概率，这种策略也不会影响数据可靠性和可用性。在三个副本的情况下，它也能降低读取时地总的网络带宽，因为一个 block 仅被放在两个机柜而不是三个上。

默认的 HDFS 副本放置策略可以概述如下：

1.每个 DataNode 最多包含 block 的一个副本

2.在集群具有足够的机柜的情况下，每个机柜最多包含同一个 block 的两个副本

3.3Replication 管理

NameNode 会尽量保证每个 block 总是具有期望的副本数。当来自 DataNode 的 block report 到达时，NameNode 会检测到那些副本数过少(under- replicated)

或过多(over-replicated)的 block。当一个 block 的副本数过多，NameNode 会选择一个副本进行删除。NameNode 首先会尽量不减少持有该副本的机柜数，其次会倾向于从那个具有最少的可用磁盘空间的 DataNode 上进行删除。目标就是尽量平衡 DataNodes 的存储空间使用率，同时又不降低 block 的可用性。

当一个 block 的副本数过少时，它会被放入一个 replication 优先队列。只有一个副本的 block 会具有最高的优先级，那些具有三分之二以上的完好副本数的 blocks 具有最低的优先级。后台线程会周期性地扫描该队列的头部来决定新副本的放置。Block replication 会遵循一个与前面的新副本放置类似的策略。如果现有副本数是一，HDFS 会将下一个副本放置到不同的一个机柜上。在现有副本数是二的情况下，如果现存的两个位于同一个机柜上，那么第三个副本将会被放到另一个机柜上；反之，第三个副本将会被放置到与现有的某个副本相同机柜的不同节点上。这里的目标是为了减少新副本的创建开销。

NameNode 也会保证一个 block 的所有副本不会被放置到同一个机柜上。如果 NameNode 检测到某个 block 的所有副本都处于同一个机柜上，NameNode 会将该 block 当做是副本数过少的情况，然后使用与前面相同的放置策略将 block 复制到另一个机柜上。当 NameNode 收到副本创建成功的通知之后，该 block 就变成了副本数过多的状态。之后 NameNode 会决定删除一个旧的副本，因为针对副本数过多的情况，处理策略是尽量不降低机柜数。

3.4Balancer

HDFS 块放置策略没有考虑 DataNode 的磁盘空间使用状况。主要是为了避免将新数据(更有可能被访问)聚集到个别的 DataNodes 上。因此数据可能并不是总是均匀分布的。同时当有新节点添加到集群中时，集群也会处于 imbalance 状态。

Balancer 是一个用于平衡 HDFS 集群的磁盘空间使用率的工具。它以一个取值范围在(0,1)的阈值作为输入参数。如果对于每个 DataNode 来说，它的磁盘空间使用率(已用空间占节点总的存储空间的比率)与整个集群的使用率(整个集群的已用空间占集群总存储空间的比率)差值不超过该阈值，我们就认为该集群已处于平衡状态。

该工具作为一个可以由集群管理员运行的应用程序部署在集群上。它会不断地将副本从使用率高的 DataNodes 移动到使用率低的 DataNodes 上。对于 Balancer

的一个关键需求就是保持数据的可用性。在选择一个副本的移动目标时，Balancer需要保证此次移动既不能降低副本数也不能降低机柜数。

Balancer 会通过最小化机柜间的数据拷贝来进行优化。如果 Balancer 决定副本 A 需要移到另一个不同的机柜上时，恰好目标机柜上有该 block 的另一个副本 B，那么数据会从 B 处直接进行拷贝而不需要再从 A 处。

还有一个配置参数可以用来限制 rebalancing 操作消耗的带宽。允许它消耗的带宽越高，集群就能越快达到平衡状态，但是也会带来与应用程序进程间更大的资源竞争。

3.5Block Scanner

每个 DataNode 会运行一个 block scanner 周期性地扫描它的 block 副本，验证 block 数据与存储的校验和是否匹配。在每个扫描周期中，block scanner 会调整读取带宽以保证可以在配置的时间周期内完成验证。当客户端读取一个完整的 block 并且检验和验证成功，它会通知 DataNode。DataNode 会将它视为一个对该副本的有效验证{!即因为客户端读取时会进行校验和验证，这样我们就可以直接利用它的验证结果，而不需要 DataNode 的 block scanner 再去验证，这就节省了计算资源}。

每个 block 校验的时间点会存储在一个人工可读的日志文件中。在任意时刻顶层的 DataNode 目录下，都会有两个文件，当前的及前一个日志。新的校验时间会被 append 到当前的文件中。相应地，每个 DataNode 在内存中都保存了一个根据副本校验时间排好序的扫描列表。

无论何时当一个正在读的客户端或者 block scanner 检测到一个损坏的 block 时，都会通知 NameNode。NameNode 会将该副本标记为损坏，但是不会立即对该副本进行删除，而是开始为该 block 复制一个完好的拷贝。只有当好的副本数达到该 block 的正常副本数的情况下，那个损坏的副本才会被删除。该策略旨在尽可能地对数据进行保护。因此即使某个 block 的所有副本都损坏了，该策略还能允许用户从损坏的副本中恢复数据。

3.6Decommissioning(下线)

集群管理员可以通过列出允许进行注册的主机地址和不允许进行注册节点的主机地址，来指定可以加入到集群的节点。管理员可以命令系统重新计算这两

种列表。如果集群中现有的一个节点出现在了排除列表中，就会被标记为Decommissioning。一旦一个 DataNode 被标记为 Decommissioning，它就不会再被选定为副本放置的目标，但是它仍会继续响应读请求。NameNode 会开始将它上面的 blocks 的副本调度到其他 DataNodes 上。一旦 NameNode 检测到该 Decommissioning 节点上的所有 blocks 已复制完成，该节点就会进入 Decommissioned 状态。之后，它就可以安全地从集群中删除而不带来任何数据可用性方面的危害。

3.7 跨集群数据拷贝

在处理大规模数据集时，将数据拷入或拷出 HDFS 集群是很吓人的。HDFS 为大规模的集群内/集群间拷贝提供一个叫做 DistCp 的工具。它是一个 MapReduce job；每个 Map task 会将元数据的一部分拷贝到目标文件系统。MapReduce 框架会自动地处理并行 task 的调度，错误检测和恢复。

4.Practice At Yahoo!

Yahoo!的大规模集群包含大概 3500 个节点。一个典型的集群节点配置如下：

- 2 个 4 核 Xeon 处理器@2.5g 赫兹
- Red Hat Enterprise Linux 服务器 Release 5.1
- Sun Jave JDK 1.6.0_13-b03
- 4 个 SATA 磁盘驱动器(每个 1TB)
- 16G RAM
- gigabit Ethernet

70%的磁盘空间会被分配给 HDFS。剩余的会预留给操作系统，日志，以及 map tasks 产生的中间输出(MapReduce 中间文件没有存储在 HDFS 上)。单个机柜内的 40 个节点共享一个 IP 交换机。机柜上的交换机又会被连接到 8 个核心交换机中的某一个。核心交换机提供了机柜间的以及到外部集群的连通性。对于每个集群来说，NameNode 和 BackupNode 会被特别安置在具有 64GB RAM 的机器上；应用程序 tasks 不会被调度到它们所在的机器上。总共算起来，一个 3500 个节点的集群具有 9.8PB 的可用存储空间，因为 blocks 被存了三份，因此对于应用程序来说只有 3.3PB 的实际存储。大概算下来，1000 个节点代表了 1PB 的存储。在 HDFS 投入使用的这些年里(以及未来的日子里)，组成集群节点的主机性能伴随着

技术的改进也在不断提高。新的集群节点通常具有更高的处理器性能，更大空间的磁盘和内存。慢慢地，那些旧的节点会下线或者用做 Hadoop 的开发测试集群。关于集群节点的选择很大程度上是计算与存储间的考量。HDFS 并没有强制计算与存储之间的比率，或者是对于集群节点的存储空间做出限制。

在一个实际的大规模集群上(3500 节点)，总共有 60 million 个文件。这些文件总共有 63 million 个 blocks。因为每个 block 通常有 3 个副本，这样每个 DataNode 大概有 54000 个 block 副本。用户应用程序每天会在集群上创建 2 million 个新文件。在 Yahoo!的 Hadoop 集群中的 25000 个节点提供了 25PB 的在线数据存储。在 2010 年初，Yahoo!的数据处理规模大概在这样一个水平上，当然还在持续增长中。Yahoo!从 2004 年开始基于分布式文件系统的 MapReduce 的相关研究。Apache Hadoop 项目在 2006 年成立。在那年年底，Yahoo!已经将 Hadoop 投入到内部使用，同时有一个用于开发的 300 个节点的集群。从那时起，HDFS 已经成为 Yahoo!后台架构的不可或缺的一部分。Web Map(作为搜索引擎关键组件的网页索引)产品一直是针对 HDFS 的首要应用，它总共运行 75 个小时，产生 500TB 的 MapReduce 中间数据，300TB 的最终输出。更多的应用正在迁移到 Hadoop，尤其是那些对用户行为进行分析和建模的应用。

Becoming a key component of yahoo's technology suite meant tackling technical problems that are the difference between being a research project and being the custodian of many petabytes of corporate data。最重要的是数据的健壮性和数据的持久性。当然，性能的经济性，用户间的资源共享及对于系统操作者管理的舒适性也都是很重要的。

4.1 数据的持久性

将数据备份三次是为了防止因非关联的节点失效造成数据丢失。通过这种方式，Yahoo!降低了 block 丢失的概率；对于一个大规模集群来说，在一年的时间内丢失一个 block 的概率小于 0.005。需要注意的是每月的节点失效概率是 0.8%。(即使节点最终恢复过来，也不需要再去恢复它曾经持有的数据)。因此，对于我们上面描述的大规模集群来说，每天都会有一两个节点失效。集群大概能在两分钟之内将存放在失效节点上的 54000 个 block 副本重新创建出来。(重备份是很快的，因为它是一个可以随集群规模线性扩展的并行问题)。几个节点同时在两分

钟内失效的概率是很低的，因此某个 block 的所有副本都丢失的概率也是很低的。

节点的关联性失效是另一种完全不同的威胁。通常情况下这种失效是因为机柜或者核心交换机的失效造成的。HDFS 可以容忍一个机柜交换机的失效(每个 block 在其他机柜上还会有一份副本)。核心交换机的失效可能会导致集群中的多个机柜的节点无法连通，这种情况下某些 blocks 可能就是不可用的了。在第二种情况下，需要修复核心交换机来将不可用的副本恢复到集群中。另一种关联性的失效是由集群意外或计划中的电力供应中断引起的。如果某些机柜的电力供应中断，那么某些 blocks 就可能会变成不可用的。但是恢复电力供应可能也无法解决问题，因为集群中仍可能有一半到 1%的节点无法通过加电重启恢复过来。统计学上以及实践表明，一个大规模集群将会在加电重启中丢掉一些节点。

除了节点的完全失效之外，存储数据也可能会损坏或丢失。Block scanner 每两星期对一个大规模集群中的 blocks 进行扫描，通常在这个过程中大概会发现 20 个左右的坏副本。

4.2Caring for Commons

伴随着 HDFS 使用的增长，文件系统本身也必须引入一些方式来在一个庞杂的用户群体内共享资源。这样的一个首要 feature 就是类似于 Unix 文件目录权限管理模式的权限框架。在该框架内，文件和目录的访问权限分为针对 owner，关联到该文件和目录的用户组，及所有其他用户的三种类别。与 Unix 不同的是，HDFS 中文件没有执行权限和粘着位(即 t/T 特殊权限)。

在现有的权限框架内，用户认证是很弱的；用户身份是由其登陆身份决定的。在访问 HDFS 的时候，应用程序客户端通过查询操作系统得到用户身份和用户组。一个更强的身份认证模型目前还在开发中。在新的框架中，应用程序客户端必须出示从一个可信任源获取的 name system 证书。可能会使用不同的证书管理方式，初始实现使用了 Kerberos。用户应用程序可以使用同一个框架来确认 name system 也具有一个可信任的身份。同时 name system 也可以询问集群中每个 DataNode 的证书。

总的可用数据存储空间是由 DataNodes 数和每个 DataNode 可以提供的存储空间决定的。HDFS 的早期经验展示了一种针对不同用户群体之间进行资源分配的需求。不仅要保证资源共享的公平性，还要能够防止一个具有数千个数据写入

需求的应用意外地将资源耗尽。对于 HDFS 来说，因为系统元数据总是存在 RAM 中，因此名字空间大小(文件和目录树)也是一种有限的资源。为了对存储和名字空间资源进行管理，每个目录可能会被设置一个 quota 来限制该目录下的存储资源。同时也可以设置另一个 quota 来对文件和目录数进行限制。

虽然 HDFS 架构假定大部分的应用程序会以大规模的数据集为输入，但是 MapReduce 编程框架可能会产生很多小输出文件(每个 reduce task 产生一个)，这会加大对于名字空间资源的占用。为方便起见，一个目录子树可以被合并为一个 Hadoop 归档文件。一个 HAR 文件类似于我们所熟悉的 tar，JAR 或者 Zip 文件，但是文件系统操作必须能够识别出归档文件中的内部文件，一个 HAR 文件应该可以透明地用作一个 MapReduce job 的输入。

4.3Benchmarks

HDFS 的设计目标是为大规模数据集提供高的 IO 带宽。通常有三种针对该目标的度量方式。

- 通过人为的 benchmark 观察带宽是怎样的
- 通过在一个具有多个用户 job 的生产集群里观察带宽是怎样的
- 通过精心构建的大规模用户应用观察带宽是怎样的

这里的统计报告来自于那些至少具有 3500 个节点的集群。在这个规模上，总带宽与节点数成线性关系，因此单节点的带宽是一个很有意义的统计信息。这些 benchmark 本身是 Hadoop 代码的一部分。

DFSIO benchmark 用于测量读写及 append 操作的平均吞吐率。DFSIO 作为一个可用的应用程序，目前是 Hadoop 发布版的一部分。该 MapReduce 程序会从/向文件中读/写/appends 随机数据。Job 内的每个 map task 会在一个不同的文件上执行相同的操作，传输相同大小的数据，同时会将它们的传输速率报告给一个 reduce task。Reduce task 之后会对这些测量信息进行汇总。这项测试在运行时是独占集群的，同时根据集群大小按固定比例来选定 map task 的数目。它只是设计用来测量数据传输性能的，会排除掉任务调度，启动及 reduce task 的开销。

- DFSIO Read：66MB/s per node
- DFSIO Write：40MB/s per node

对于一个生产集群来说，读写的字节数将会被报告给一个 metrics 收集系统。

这些值是几个星期的平均值同时代表着数百个用户的 jobs 的集群使用情况。平均情况下，每个节点上任意时刻会运行这一两个应用程序 tasks(小于可用的处理器核数)。

- Busy Cluster Read：1.02MB/s per node
- Busy Cluster Write：1.09MB/s per node

Bytes	Nodes	Maps	Reduces	Time	HDFS I/O Bytes/s	
					Aggregate(GB)	Per Node(MB)
1	1460	8000	2700	62s	32	22.1
1000	3658	80000	20000	58500s	34.2	9.35

表 2.针对 1TB 和 1PB 数据的 Sort benchmark。每条数据记录有 100 字节，其中 key 有 10 字节。测试程序是一个通用的排序过程而并未针对记录大小进行特殊处理。在 1TB 数据排序中，block 副本数设成了 1，对于一个持续时间比较短的测试来说这是一个合理的设置。在 1PB 数据排序中，block 副本数为 2，这样测试程序就可以在即使有节点失效的情况也可以顺利完成。

在 2009 初，Yahoo!参与了 Gray Sort 比赛，并拿下了冠军。该 task 本身对系统将数据移入移出文件系统的能力要求很高(实际上它的关键并不在于排序) [9]。最后一列的 I/O rate 包含了对 HDFS 的读入及写出。在第二行里，虽然 HDFS 的 rate 有所下降，但是单节点的 I/O 却大概增加了一倍，这是因为对于更大规模(petabyte!)的数据集来说，MapReduce 的中间结果也必须对磁盘进行写入和读取。在小规模的测试里，是不会将 MapReduce 的中间数据溢写(spill)到磁盘的；它们被直接缓存到了 task 的内存中。

大规模集群需要 HDFS NameNode 能够支持与集群规模相对应的大量的 client 操作。NNThroughput benchmark 是一个单机进程，它会启动 NameNode 应用程序，同时在同一个节点上运行大量的客户端线程。每个客户端线程会通过直接调用 NameNode 接口来执行同一个 NameNode 操作。该 benchmark 是用来测量 NameNode 每秒可执行的操作数。为避免由 RPC 连接和序列化引起的开销，该 benchmark 是运行在本地而不是远程的节点上。通过该测试得到了纯 NameNode

的性能上界。

Operation	Throughput(ops/s)
Open file for read	126100
Create file	5600
Rename file	8300
Delete file	20700
DataNode Hearbeat	300000
BolcksReport(blocks/s)	639700

5.工作展望

本节提出一些 Yahoo 的 Hadoop 团队正在考虑中的一些未来的工作计划；Hadoop 作为一个开源项目，意味着很多新的 features 和变更需要由 Hadoop 开发者社区来决定。

当 NameNode down 掉的时候 Hadoop 集群实际上就会变成不可用的。由于 Hadoop 主要是作为一种批处理系统使用，重启 NameNode 也是一种可以接受的恢复方式。但是，我们已经开始向着自动化的故障恢复(failover)而迈进。当前情况下，BackupNode 会接受来自 primary NameNode 的所有事务。如果我们同时将 block reports 发送给 primary NameNode 和 BackupNode，这就允许一个故障恢复成为 warm 的或者 hot 的{!与冷启动，热启动中的含义相同，当然也有处于二者之间的暖}。我们的目标是使用 Zookeeper 来构建一个自动化的故障恢复解决方案。

NameNode 的可扩展性[13]已成为一个首要的需要解决的问题。因为 NameNode 将名字空间和 block 位置信息全部保存在内存中，NameNode 的堆空间大小已经限制了文件数及可寻址的 blocks 数。当 NameNode 的内存使用率接近极限时，NameNode 就会变成无响应的有时甚至需要进行重启，这已经成为 NameNode 面临的主要挑战。虽然我们鼓励用户创建更大的文件，但是有时这也是不可行的，因为这需要对应用程序进行比较大的变更。现在我们已经为 HDFS 的使用管理提供了 quota，同时也提供了一个归档工具。然而，这些都没有从根本上解决可扩展性问题。

我们针对可扩展性的近期解决方案是允许使用多个名字空间(及多个NameNodes)来共享集群内的物理资源。我们正在扩展我们的 block IDs，使它可以以一个 block pool 标识符为前缀。Block pools 类似于一个 SAN(StorgeAreaNetwork)存储系统中的 LUNs(LogicUnitNumber)，而具有多个 blocks pool 的名字空间类似于一个文件系统卷(volume)。

这种策略很简单同时对系统的修改也是最小化的。除可扩展性之外，它还提供了其他一些优点：可以将不同的应用程序集隔离在不同的名字空间下，同时可以提高集群的整体可用性。可以将块存储服务进行通用化，这样就允许其他的一些具有不同名字空间结构的服务来使用这个块存储服务。我们也计划探索一些其他的方式来进行扩展，比如只将部分名字空间存在内存中，在未来提供一个 NameNode 的真正的分布式实现。此外，我们关于应用程序只会创建少数大文件的假设也是有问题的。如前所述，改变应用程序行为是很难的。此外，针对 HDFS 的新一类应用程序可能需要存储大量的小文件。

多个独立名字空间的主要缺点是带来的管理开销，尤其是在名字空间数很大的情况下。我们也计划使用以应用程序或者是 job 为中心的 namespaces 而不是以集群为中心的—这类似于 80 年代晚期和 90 年代早期[10][11][12]用于处理分布式系统中远程执行的进程级(per-process)名字空间。

目前我们的集群少于 4000 节点。我们相信通过上面的解决方案，可以将它扩展到更大的规模。然而，我们认为使用多个小集群与使用单个的大集群(比如 3 个 6000 节点集群与一个 18000 节点集群)相比要更明智一些，因为这样具有更好的可用性和隔离性。最后，我们也计划提供更多的集群间协作支持。比如，为跨越多个集群的文件集合缓存被访问的远程文件或者是降低 blocks 副本数。

6.经验

一个非常小的团队构建了 Hadoop 文件系统，同时使得它稳定健壮的运行在产品系统中。这种成功大部分归因于简单的设计：replicated blocks, 周期性地 block reports 以及中央化的元数据服务器。避免了完全的 POSIX 语义也提供了一定的帮助。尽管将整个元数据保存到内存中限制了可扩展性，这也使得 NameNode 非常简单：避免的典型文件系统中那种复杂的锁机制。Hadoop 成功的另一个原因是在 Yahoo!的快速地产品化，这就使得它可以不断地快速地改进。

文件系统十分健壮，NameNode 很少出错；事实上大部分的停机时间都是由升级造成的。只是最近才引入了自动化的故障恢复机制。

很多人可能会很吃惊，在构建这样一个大型系统中选择了 Java 语言。尽管由于 Java 的对象内存和垃圾回收开销给 NameNode 的扩展造成了一些挑战，但是 Java 也带来了系统的健壮性；避免了由指针和内存管理 bugs 造成的危害。

7.致谢

我们需要感谢现在的和过去的所有的 Yahoo! HDFS 团队成员，感谢他们在构建该文件系统中所做的努力。我们也需要感谢所有的 Hadoop committers 和 collaborators(合作者)，感谢他们的宝贵共享。Corinne Chandel 绘制了论文中的插图。

参考文献：

[1] Apache Hadoop. http://hadoop.apache.org/

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. “PVFS: A parallel file system for Linux clusters,” in Proc. of 4th Annual Linux Showcase and Conference, 2000, pp. 317–327.

[3] J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” In Proc. of the 6th Symposium on Operating Systems Design and Implementation, San Francisco CA, Dec. 2004.

[4] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C.Olston, B. Reed, S. Srinivasan, U. Srivastava. “Building a High-Level Dataflow System on top of MapReduce: The Pig Experience,” In Proc. of Very Large Data Bases, vol 2 no. 2, 2009, pp. 1414–1425

[5] S. Ghemawat, H. Gobioff, S. Leung. “The Google file system,” In Proc.of ACM Symposium on Operating Systems Principles, Lake George,NY, Oct 2003, pp 29–43.

[6] F. P. Junqueira, B. C. Reed. “The life and times of a zookeeper,” In Proc. of the 28th ACM Symposium on Principles of Distributed Computing, Calgary, AB, Canada, August 10–12, 2009.

[7] Lustre File System. http://www.lustre.org

[8] M. K. McKusick, S. Quinlan. “GFS: Evolution on Fast-forward,” ACM Queue,

vol. 7, no. 7, New York, NY. August 2009.

[9] O. O'Malley, A. C. Murthy. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. May 2009. http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html

[10] R. Pike, D. Presotto, K. Thompson, H. Trickey, P. Winterbottom, “Use of Name Spaces in Plan9,” Operating Systems Review, 27(2), April 1993, pages 72–76.

[11] S. Radia, "Naming Policies in the spring system," In Proc. of 1st IEEE Workshop on Services in Distributed and Networked Environments, June 1994, pp. 164–171.

[12] S. Radia, J. Pachl, “The Per-Process View of Naming and Remote Execution,” IEEE Parallel and Distributed Technology, vol. 1, no. 3, August 1993, pp. 71–80.

[13] K. V. Shvachko, “HDFS Scalability: The limits to growth,” ;login:.April 2010, pp. 6–16.

[14] W. Tantisiriroj, S. Patil, G. Gibson. “Data-intensive file systems for Internet services: A rose by any other name ...” Technical Report CMU-PDL-08-114, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, October 2008.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu,P. Wyckoff, R. Murthy, “Hive – A Warehousing Solution Over a Map-Reduce Framework,” In Proc. of Very Large Data Bases, vol. 2 no. 2,August 2009, pp. 1626-1629.

[16] J. Venner, Pro Hadoop. Apress, June 22, 2009.

[17] S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn, “Ceph: A Scalable,High-Performance Distributed File System,” In Proc. of the 7 thSymposium on Operating Systems Design and Implementation, Seattle,WA, November 2006.

[18] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J.Zelenka, B. Zhou, “Scalable Performance of the Panasas Parallel file System”, In Proc. of the 6 th USENIX Conference on File and StorageTechnologies, San Jose, CA, February 2008

[19] T. White, Hadoop: The Definitive Guide. O'Reilly Media, Yahoo! Press, June 5, 2009.