**海量存储课程报告**

—— Hadoop Distributed File System

姓名： 张淳

学院： 计算机学院

年级： 2017

学号：2017282110234

**摘要**

Hadoop分布式文件系统（HDFS）旨在可靠地存储非常大的数据集，并将这些数据集以高带宽流式传输到用户应用程序。 在大型集群中，数千台服务器都直接连接存储并执行用户应用程序任务。 通过在多个服务器之间分配存储和计算，资源可以随需求增长，同时保持每个尺寸的经济性。 我们描述了HDFS的体系结构，并报告了未来的方向。

**一. 介绍和相关工作**

Hadoop [1] [16] [19]提供了一个分布式文件系统和一个框架，用于使用MapReduce [3]范式分析和转换非常大的数据集。 Hadoop的一个重要特性是跨多台（千台）主机对数据和计算进行分区，并且在并行执行应用程序计算的同时接近其数据。 Hadoop集群通过简单地添加商品服务器来扩展计算容量，存储容量和IO带宽。 Hadoop集群在雅虎！ 跨越25 000个服务器，并存储25 PB的应用程序数据，最大的集群是3500个服务器。 世界各地的其他一百个组织报告使用Hadoop。

|  |  |
| --- | --- |
| HDFS | 分布式文件系统 本文主题 |
| MapReduce | 分布式计算框架 |
| HBase | 面向列的表服务 |
| Pig | 数据流语言和并行执行框架 |
| Hive | 数据仓库基础架构 |
| ZooKeeper | 分布式协调服务 |
| Chukwa | 收集管理数据的系统 |
| Avro | 数据序列化系统 |

表1. Hadoop项目组件

Hadoop是一个Apache项目; 所有组件都可以通过Apache开源许可证获得。雅虎 已经开发并贡献了80％的Hadoop核心（HDFS和MapReduce）。 HBase最初是在Powerset开发的，现在则是微软的部门。 Hive [15]起源于Facebook并由其开发。 Pig，ZooKeeper和Chukwa则是由Yahoo!建立和开发的。 Avro起源于Yahoo! ，并正在被Cloudera共同开发。

HDFS是Hadoop的文件系统组件。 虽然HDFS的接口是在UNIX文件系统之后形成的，但为提高应用程序的性能牺牲了与标准的符合度。

HDFS分别存储文件系统元数据和应用程序数据。 与其他分布式文件系统一样，HDFS将元数据存储在专用服务器（称为NameNode）上，如PVFS [2] [14]，Lustre [7]和GFS [5] [8]。 应用程序数据存储在名为DataNode的其他服务器上。 所有服务器都完全连接，并使用基于TCP的协议相互通信。

我们可以将NameNode理解为管理者，而DataNode则是服从者。这两者之间的地位是不平等的，后者受到前者的控制。

与Lustre和PVFS不同，HDFS中的DataNode不使用RAID等数据保护机制来保证数据的持久性。 相反，像GFS一样，文件内容也被复制到多个DataNode上以保证可靠性。 在确保数据持久性的同时，该策略还具有数据传输带宽倍增的优点，并且在有更多机会定位在所需数据附近的计算。

几个分布式文件系统已经或正在探究真正的分布式命名空间的实现。 Ceph [17]拥有一个命名空间服务器（MDS）集群，并使用动态子树分区算法将命名空间树均匀映射到MDS。 GFS也正在演变成一个分布式命名空间实现[8]。 新的GFS将有数百个命名空间服务器（Master），每个Master有1亿个文件。 Lustre [7]在其Lustre 2.2发行版的路线图上实现了集群名称空间。 目的是在多个元数据服务器（MDS）上分割一个目录，每个元数据服务器都包含一个不相交的名称空间部分。 一个文件会被分派到在文件名上使用哈希函数的元数据服务器上。

**二. 架构**

*A. NameNode*

HDFS命名空间是文件和目录的层次结构。文件和目录通过inode表示在NameNode上，它们记录权限，修改和访问时间，命名空间和磁盘空间配额等属性。 文件内容被分割成多个大块（通常为128MB，但是用户可选择逐个文件），并且文件的每个块在多个DataNode（通常是三个，但是用户可选的逐个文件）被独立地存储。NameNode会维护命名空间树和文件块到DataNode（文件数据的物理位置）的映射。 想要读取文件的HDFS客户端首先与NameNode通信，以获取构成文件的数据块的位置，然后从离客户端最近的DataNode读取文件块的内容。 在写入数据时，客户端请求NameNode选出一组三个DataNode来承载块副本。 客户端然后以流水线方式将数据写入DataNode。 在当前的设计下，每个集群都有一个NameNode， DataNode则可以有数千个，HDFS客户端更多，可以有成千上万个。这是因为每个DataNode可以同时执行多个应用程序任务。

HDFS将整个名称空间保留在RAM中。 inode数据和属于每个文件的块的列表包括称为图像的名称系统的元数据。 存储在本地主机本地文件系统中的图像的持久记录称为检查点。 NameNode还在本地主机的本地文件系统中存储称为日志的映像的修改日志。 为了提高耐用性，可以在其他服务器上创建检查点和日志的冗余副本。 在重新启动期间，NameNode通过读取命名空间并重播日志来恢复命名空间。 块副本的位置可能会随时间而改变，而不是持久性检查点的一部分。

*B. DataNodes*

DataNode上的每个块副本由本地主机的本机文件系统中的两个文件表示。 第一个文件包含数据本身，第二个文件是块的元数据，包括块数据和块生成标记的校验和。 数据文件的大小等于块的实际长度，并且不需要额外的空间就可以像传统文件系统中那样将其四舍五入到标称块大小。 因此，如果一个块是半满的，它只需要本地完整块的一半空间

驾驶。

在启动期间，每个DataNode连接到NameNode并执行握手。 握手的目的是验证DataNode的命名空间ID和软件版本。 如果任一个与NameNode不匹配，则DataNode自动关闭。

命名空间ID在格式化时被分配给文件系统实例。 命名空间ID永久存储在群集的所有节点上。 具有不同命名空间ID的节点将无法加入群集，从而保持文件系统的完整性。

软件版本的一致性非常重要，因为不兼容的版本可能会导致数据损坏或丢失。在数千台机器的大型集群上，很容易忽略在软件升级之前没有正确关闭的节点，或者在升级过程中不可用的节点。

新初始化并且没有任何命名空间ID的DataNode被允许加入集群并接收集群的名称空间ID。 握手后，DataNode向NameNode注册。 DataNode持久存储其唯一的存储ID。 存储ID是DataNode的内部标识符，即使使用不同的IP地址或端口重新启动，也可以识别该ID。当第一次向NameNode注册时，存储ID将被分配给DataNode，并且在此之后从不改变。

DataNode通过发送块报告来识别NameNode拥有的块副本。块报告包含块ID，生成标记和服务器主机的每个块副本的长度。第一个块报告在DataNode注册后立即发送。随后的块报告每小时发送一次，并向NameNode提供块副本位于群集上的最新视图。

在正常操作期间，DataNodes将心跳发送到NameNode以确认DataNode正在运行，并且它所托管的块副本可用。默认的心跳间隔是三秒钟。如果NameNode在十分钟内没有收到来自DataNode的心跳信号，NameNode将DataNode视为不在服务中，并且由该DataNode承载的块副本不可用。然后，NameNode计划在其他DataNode上创建这些块的新副本。

来自DataNode的心跳信息还包含有关总存储容量，正在使用的存储部分以及当前正在进行的数据传输的信息。这些统计信息用于NameNode的空间分配和负载平衡决策。。

NameNode不直接调用DataNode。 它使用对心跳的回复将指令发送到DataNode。 说明包括命令：

•将块复制到其他节点;

•删除本地块副本;

•重新注册或关闭节点;

•发送立即的阻止报告。

这些命令对于维护整个系统的完整性非常重要，因此即使在大型集群上保持心跳频繁也是至关重要的。 NameNode每秒可以处理数千次的心跳，而不会影响其他NameNode操作。

*C. HDFS 客户端*

用户应用程序使用HDFS客户端（一个导出HDFS文件系统接口的代码库）访问文件系统。

与大多数传统文件系统类似，HDFS支持读取，写入和删除文件的操作，以及创建和删除目录的操作。用户通过命名空间中的路径引用文件和目录。用户应用程序通常不需要知道文件系统元数据和存储位于不同的服务器上，或者该块具有多个副本。

当一个应用程序读取一个文件时，HDFS客户端首先向NameNode询问承载该文件块的副本的DataNode列表。然后它直接联系DataNode并请求传输所需的块。当客户端写入时，它首先要求NameNode选择DataNode来承载文件的第一个块的副本。客户端组织从节点到节点的管道并发送数据。当第一个块被填满时，客户端请求选择新的DataNode来承载下一个块的副本。这时一个新的管道会被组织起来，客户端继续发送文件的更多字节。 每个DataNode的选择可能会有所不同。 客户端，NameNode和DataNode之间的交互如图1所示。

与传统的文件系统不同，HDFS提供了一个暴露文件块位置的API。 这允许MapReduce框架等应用程序将任务安排到数据所在的位置，从而提高了读取性能。 它还允许应用程序设置文件的复制因子。 默认情况下，文件的复制因子是三。 对于经常访问的关键文件或文件，具有更高的复制因子可提高其对错误的容忍度并增加其读取带宽。

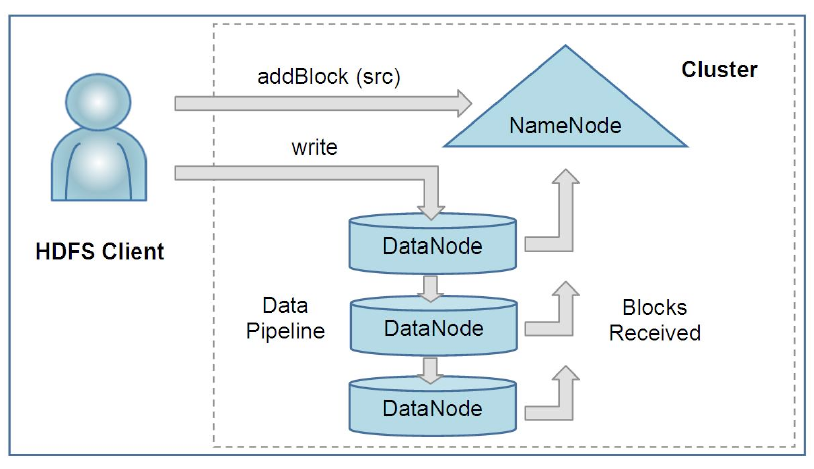


图1. HDFS客户端通过给NameNode提供路径来创建一个新文件。 对于文件的每个块，NameNode返回一个DataNode列表来托管其副本。 然后，客户端将数据传递到选定的DataNode，最终确认将块副本创建到NameNode。

*D. 映像和日志*

名称空间映像是描述应用程序数据组织为目录和文件的文件系统元数据。写入磁盘的映像的持久记录称为检查点。日志是一个预先提交的提交日志，用于记录文件系统的更改，必须持久存在。对于每个由客户端启动的事务，更改都记录在日志中，日志文件在更改提交给HDFS客户端之前刷新并同步。 NameNode永远不会改变检查点文件;在重新启动期间创建新的检查点时，管理员请求时或在下一节中介绍的CheckpointNode时，将全部替换它。在启动过程中，NameNode会从检查点初始化名称空间映像，然后从日志中重放更改，直到映像与文件系统的最后一个状态保持一致。 NameNode开始为客户端提供服务之前，将新的检查点和空日志写回存储目录。

如果检查点或日志丢失或者被破坏，命名空间信息将会部分或全部丢失。 为了保存这个关键信息，HDFS可以配置为将检查点和日志存储在多个存储目录中。 建议的做法是将目录放在不同的卷上，并将一个存储目录放在远程NFS服务器上。 第一个选择是防止单个卷失败的损失，第二个选择防止整个节点的失败。 如果NameNode在将日志写入其中一个存储目录时遇到错误，它会自动从存储目录列表中排除该目录。 如果没有存储目录可用，NameNode会自动关闭。

NameNode是一个多线程系统，并同时处理来自多个客户端的请求。 将事务保存到磁盘是一个瓶颈，因为只有多个线程中的一个线程启动的同步刷新和同步过程完成前，所有其他线程都需要等待。 为了优化这个过程，NameNode批量处理由不同客户端发起的多个事务。 当NameNode的线程之一启动一个刷新和同步操作时，所有在此批的事务将一起提交。 剩余的线程只需要检查他们的事务已经被保存了，不需要发起刷新和同步操作。

*E. 检查点节点*

HDFS中的NameNode除了主要处理来自于客户端请求之外，还可以扮演CheckpointNode或BackupNode两个角色中的任意一个角色。 该角色在节点启动时指定。CheckpointNode定期组合现有的检查点和日志来创建一个新的检查点和一个空的日志。 CheckpointNode通常在与NameNode不同的主机上运行，因为它具有与NameNode相同的内存要求。 它从NameNode下载当前的检查点和日志文件，在本地合并它们，并将新的检查点返回给NameNode。

创建定期检查点是保护文件系统元数据的一种方法。 如果名称空间图像或日志的所有其他持久性副本都不可用，则系统可以从最近的检查点开始。

当新检查点上传到NameNode时，创建一个检查点让NameNode截断日志的尾部。 HDFS集群长期运行而不重新开始，日志随之不断增长。 如果日志增长的很大，日志文件丢失或损坏的可能性就会增加。 另外，一个非常大的日志加长了重新启动NameNode所需的时间。 对于一个大集群，处理一个为期一周的日志需要一个小时。 好的做法是创建一个每日检查点。

*F. BackupNode*

最近引入的HDFS功能是BackupNode。像CheckpointNode一样，BackupNode能够创建定期的检查点，另外它还驻留内存，更新文件系统命名空间映像，它总是与NameNode同步。

BackupNode接受来自活动NameNode的名称空间事务的日志流，将它们保存到它自己的存储目录中，并将这些事务应用到内存中它自己的名称空间映像。 NameNode将BackupNode视为日志存储，就像它将日志文件视为其存储目录一样。如果NameNode失败，则内存中的BackupNode映像和磁盘上的检查点是最新命名空间状态的记录。

BackupNode可以创建一个检查点，而无需从活动的NameNode中下载检查点和日志文件，因为它的内存中已经有一个最新的命名空间映像。这使得BackupNode上的检查点进程更加高效，因为它只需要将名称空间保存到其本地存储目录中。

BackupNode可以被视为一个只读的NameNode。它包含除了块位置之外的所有文件系统元数据信息。它可以执行常规NameNode的所有操作，只要不涉及修改名称空间或块位置的知识。BackupNode为人们提供了运行NameNode而不需要永久存储的选项，可以保持命名空间的状态。

*G. 升级，文件系统快照*

在软件升级期间，由于软件错误或人为错误导致系统损坏的可能性增加。在HDFS中创建快照的目的是为了尽量减少在升级过程中对存储在系统中的数据的潜在损害。

通过快照机制，管理员可以持久地保存文件系统的当前状态，以便在升级导致数据丢失或损坏的情况下，可以回滚升级并将HDFS恢复到快照对应时间的命名空间和存储状态。

在系统启动时，快照（只能存在一个）在集群管理员的选项中被创建。如果请求快照，则NameNode首先读取检查点和日志文件并将其合并到内存中。然后将新检查点和空日志写入新位置，以便旧检查点和日志保持不变。在握手期间，NameNode指示DataNode是否创建本地快照。复制数据文件目录来创建DataNode上的本地快照是不行的，因为这需要将群集中每个DataNode的存储容量加倍。采用的做法是每个DataNode创建一个存储目录的副本，并将现有的块文件硬链接到它。当DataNode删除一个块时，它只删除硬链接，并且在附加过程中阻止修改使用copy-on-write技术。因此旧的块复制品在其旧目录中保持不变。

在重新启动系统时，群集管理员可以选择将HDFS回滚到快照状态。 NameNode恢复创建快照时保存的检查点。 DataNodes恢复先前重命名的目录并启动后台进程以删除创建快照后创建的块副本。选择回滚之后，没有任何设备可以继续。集群管理员可以通过命令系统放弃快照来恢复快照占用的存储空间，从而完成软件升级。

系统更新可能会导致NameNode的检查点和日志文件的格式发生变化，或导致DataNodes上的块副本文件的数据表示形式发生变化。布局版本确定了数据的表示格式，并持久存储在NameNode和DataNode的存储目录中。在启动过程中，每个节点都会将当前软件的布局版本与其存储目录中存储的版本进行比较，并自动将数据从旧格式转换为新格式。当系统重新启动新的软件布局版本时，转换需要强制创建快照。HDFS不会为NameNode和DataNodes分开布局版本，因为快照创建必须是全集群工作而不是节点选择性事件。如果由于软件错误而导致升级的NameNode清除其映像，只副本名称空间状态仍会导致数据全部丢失，因为NameNode不会识别由DataNode报告的块，并将其命令删除。在这种情况下回滚虽然会恢复元数据，但数据本身将丢失。需要同步快照以避免灾难性的破坏

**三. 文件IO操作和副本管理**

*A 文件读写*

应用程序通过创建一个新文件并将数据写入HDFS来向HDFS添加数据。 文件关闭后，写入的字节不能被修改或删除，只能通过重新打开文件添加新的数据添加到文件中。 HDFS实现了一个单写者-多读者模型。

打开文件进行写入的HDFS客户端会被授予该文件的租约; 没有其他客户端可以写入文件。 写入客户端通过向NameNode发送心跳来定期更新租约。 当文件关闭时，租约被吊销。租约期限受到软限制和硬限制的约束。直到软限制到期，作者对文件有一定的访问权限。 如果软限制到期，客户端无法关闭文件或续订租约，则另一客户端可以抢占租约。 如果在硬限制到期（一小时）且客户端未能续约的情况下，HDFS假定客户端已经退出，并且将自动代表写入者关闭文件，并恢复租约。 作者的租约不妨碍其他客户读取文件; 一个文件可能有许多并发读者。

一个HDFS文件由多个块组成。 当需要一个新块时，NameNode分配一个具有唯一块ID的块，并确定一个DataNode列表来存放该块的副本。 DataNode形成一个管道，其顺序使从客户端到最近一个DataNode的总网络距离最小。 字节作为一系列数据包被推送到流水线。 应用程序在客户端写入第一个缓冲区的字节。 数据包缓冲区填满（通常为64 KB）后，数据被推送到管道。 下一个数据包可以在接收到之前数据包的确认之前被推送到流水线。 未完成数据包的数量受限于客户端的未完成数据包窗口大1小。

将数据写入HDFS文件后，HDFS不保证数据对新的读者可见，直到文件关闭。 如果用户应用程序需要可见性保证，则可以显式调用hflush操作。 然后，当前数据包立即被推送到流水线，并且hflush操作将等待，直到流水线中的所有DataNode确认数据包的成功传输。 所有在hflush操作之前写入的数据一定会被读者看到。

如果没有错误发生，则块的构建经历三个阶段，如图2所示，展出出了三个DataNode的流水线和五个包的块。 图片中，粗线表示数据包，虚线表示确认消息，细线表示控制消息以设置和关闭流水线。 垂直线代表客户端和三个DataNode的活动，时间从上到下。 从t0到t1是管道设置阶段。 时间间隔t1到t2是数据流阶段，其中t1是第一个数据包发送的时间，t2是对最后一个数据包的确认收到的时间。 这里hflush操作传输第二个数据包。 hflush指示与分组数据一起传送，不是单独的操作。 最后的时间间隔t2至t3是该块的管道关闭阶段。

在数千个节点的集群中，节点的故障（最常见的是存储故障）是每天发生的。 存储在DataNode上的副本可能由于内存，磁盘或网络故障而损坏。 HDFS为HDFS文件的每个数据块生成和存储校验和。 校验和在读取时由HDFS客户端验证，以帮助检测客户端，DataNode或网络造成的任何损坏。 当客户端创建一个HDFS文件时，它会计算每个块的校验和序列，并将其与数据一起发送到一个DataNode。 DataNode将校验和存储在与块的数据文件分开的元数据文件中。 当HDFS读取一个文件时，每个数据块的数据和校验和

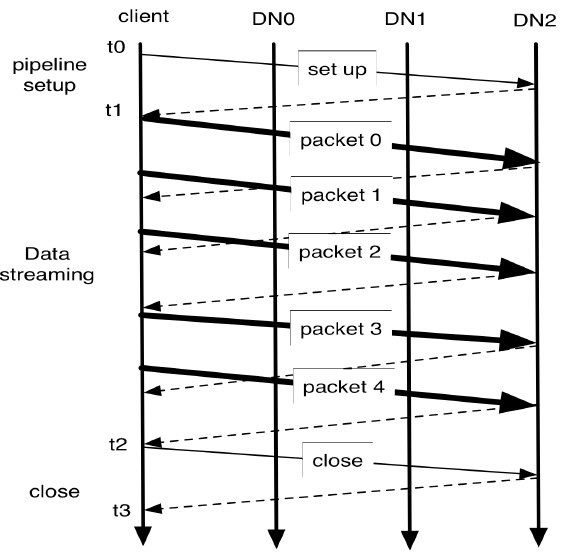


图 2.块构建时的数据管道

被发送到客户端。 客户端计算接收到的数据的校验和，并验证新计算的校验和与它收到的校验和相匹配。 如果不是，则客户端通知NameNode损坏的副本，然后从另一个DataNode中获取该块的不同副本。

当客户端打开要读取的文件时，它会从NameNode中获取块的列表以及每个块副本的位置。 每个区块的位置按照距离阅读器的距离排序。 读取块的内容时，客户端首先尝试最接近的副本。 如果读取尝试失败，客户端将依次尝试下一个副本。 如果目标DataNode不可用，节点不再承载该块的副本，或者在校验和测试时发现副本损坏，则读取可能会失败。

HDFS允许客户读取一个可写入的文件。 读取打开的文件时，NameNode不知道最后一个正在写入的块的长度。 在这种情况下，客户会在开始阅读其内容之前询问其中一个副本的最新长度。

HDFS I / O的设计特别针对批量处理系统进行了优化，如MapReduce，这些系统需要高吞吐量进行顺序读取和写入。 然而，为了支持像Scribe这样的向HDFS提供实时数据流的应用程序，或者提供随机的，实时访问大型表的应用程序，HBase已经做出了许多努力来改进其读/写响应时间。

*B. 块的放置*

对于大型集群，连接平面拓扑中的所有节点可能并不实际。 通常的做法是将节点分布在多个机架上。 机架的节点共享交换机，机架交换机通过一个或多个核心交换机连接。 不同机架中的两个节点之间的通信必须经过多个交换机。 在大多数情况下，同一机架中的节点之间的网络带宽大于不同机架中的节点之间的网络带宽。 图3描述了一个有两个机架的集群，每个机架包含三个节点。

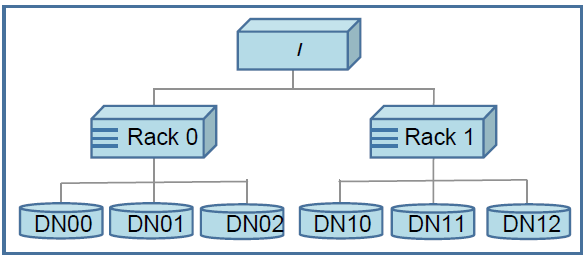


图3.集群拓扑示例

HDFS根据距离估算两个节点之间的网络带宽。 假设从节点到其父节点的距离是1。 两个节点之间的距离可以通过总结它们到它们最近的共同祖先的距离来计算。 两个节点之间的距离越短意味着可以利用更大的带宽来传输数据。

HDFS允许管理员配置脚本，该脚本返回给定节点地址的节点的机架标识。 NameNode是解决每个DataNode的机架位置的中心位置。 当一个DataNode向NameNode注册时，NameNode运行一个配置脚本来决定该节点属于哪个机架。 如果没有配置这样的脚本，则NameNode将假定所有节点都属于默认单个机架。

副本的放置对HDFS数据可靠性和读/写性能至关重要。 一个好的副本放置策略应该可以提高数据的可靠性，可用性和网络带宽利用率。 目前，HDFS提供了一个可配置的块放置策略接口，以便用户和研究人员可以尝试和测试任何最适合其应用的策略。

默认的HDFS块放置策略提供了在最小化写入成本和最大化数据可靠性，可用性和聚合读取带宽之间的折衷。 当创建一个新块时，HDFS将第一个副本放在写入程序所在的节点上，将第二个和第三个副本放在另一个机架上的两个不同节点上，其余节点放置在随机节点上，但不得超过 一个副本放置在一个节点上，当副本数量少于机架数量的两倍时，同一个机架中不会放置两个副本。 将第二个和第三个副本放置在不同机架上的选择可以更好地为集群中的单个文件分配块副本。 如果前两个副本放在同一个机架上，则对于任何文件，三分之二的副本将位于同一个机架上。

在所有目标节点被选择之后，节点按照它们与第一副本的接近度的顺序被组织为管管道。 数据按此顺序推送到节点。 为了读取，NameNode首先检查客户端的主机是否位于集群中。 如果是的话，块地址以接近读者的顺序返回给客户端。块按照此优先顺序从DataNodes中读取。 （MapReduce应用程序通常在集群节点上运行，但只要主机可以连接到NameNode和DataNode，它就可以执行HDFS客户端。）

该策略减少了机架间和节点间的写入流量，并且通常会提高写入性能。 由于机架故障的机率远远小于节点故障的机率，因此该策略不影响数据可靠性和可用性保证。 在三个副本的通常情况下，它可以减少读取数据时使用的总体网络带宽，因为一个块只能放在两个唯一的机架中，而不是三个。 默认的HDFS副本放置策略可以总结如下：

1.没有DataNode包含任何块的多个副本。

2.没有机架包含同一个块的两个以上副本，只要群集上有足够的机架。

*D: 平衡器*

HDFS块放置策略不考虑DataNode磁盘空间利用率。 这是为了避免在DataNode的一个小子集中放置更多的被引用数据。 因此，数据可能并不总是统一放置在DataNode中。 新节点添加到群集时也会发生不平衡。

平衡器是一个平衡HDFS集群磁盘空间使用情况的工具。 它将一个阈值，一个（0,1）范围内的分数作为输入参数。如果对于每个DataNode，集群是均衡的，则节点的利用率（节点上已用空间与总节点容量的比率）与整个集群的利用率（集群中已用空间与总容量的比率）的差值不超过阈值。

该工具被部署为可由群集管理员运行的应用程序。 它将具有较高利用率的DataNodes中的副本迭代移动到利用率较低的DataNode。 平衡器的一个关键要求是保持数据可用性。 当选择副本来移动并确定其目的地时，平衡器保证决定不会减少副本的数量或机架的数量。

平衡器通过最小化机架间数据复制来优化平衡过程。 如果平衡器决定副本A需要移动到不同的机架上，并且目标机架正好具有相同块的副本B，则将从副本B而不是复制副本A复制数据。

第二个配置参数限制了重新平衡操作所消耗的带宽。 允许的带宽越高，集群达到平衡状态的速度越快，但与应用程序进程竞争更加激烈。

**四. 未来的工作**

NameNode的可伸缩性一直是一个关键的问题。由于NameNode保留了内存中的所有名称空间和块位置，NameNode堆的大小限制了文件的数量，也限制了可寻址的块的数量。NameNode的主要挑战是当其内存使用量接近最大值时，由于Java垃圾收集，NameNode变得无法响应，有时需要重新启动。 虽然创建更大的文件会改善情况，但是这并没有发生，因为它需要改变应用程序行为。增加配额管理使用情况和提供归档工具并不能从根本上解决可伸缩性问题。

近期的可伸缩性解决方案是允许多个命名空间（和NameNode）共享集群内的物理存储。这种方法相当简单，只需要对系统进行微小的更改。 除了可伸缩性之外，它还提供了许多优点：它隔离了不同应用程序集的命名空间，并提高了集群的整体可用性。 它还概括了块存储抽象，以允许其他服务使用块存储服务，可能具有不同的名称空间结构。也可以探索其他的扩展方法，比如在内存中只存储部分名字空间，并在未来真正分布式实现NameNode。

**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: Aparallel 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 CMUPDL- 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 7th Symposium 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 6th USENIX Conference on File and Storage Technologies, San Jose, CA, February 2008

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