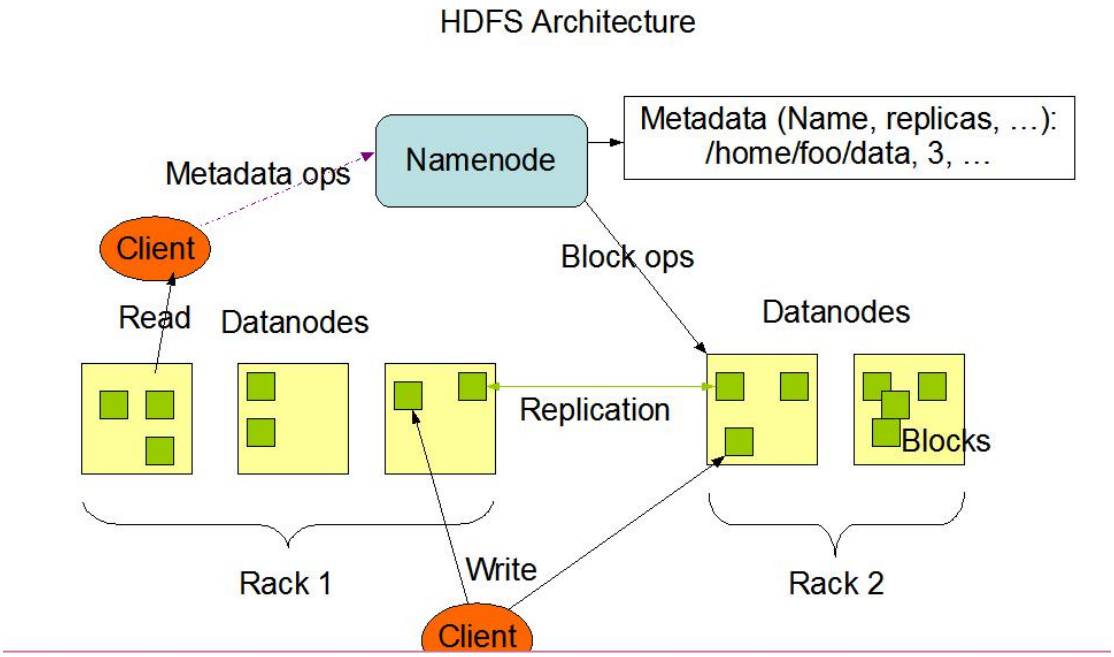
一、HDFS 简介

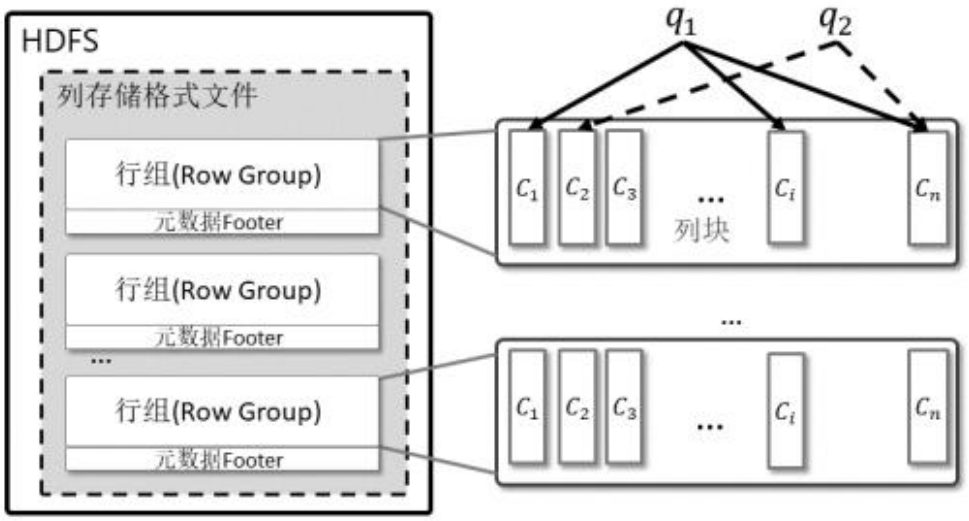
Hadoop 分布式文件系统(HDFS)被设计成适合运行在通用硬件(commodity hardware)上的分布式文件系统。它和现有的分布式文件系统有很多共同点。但同时，它和其他的分布式文件系统的区别也是很明显的。它是一个高度容错性的系统，支持多异构机器，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。同时，HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。HDFS 在最开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发的。HDFS 是 Apache Hadoop Core 项目的一部分。但是现在可以作为单独的应用使用。HDFS 包含 Java 和 C++等多种语言接口。HDFS 具体包含的优点如下:

1. 高度容错性。由于 HDFS 适合大文件存储，并且 HDFS 可能由成百上千的服务器所构成，每个服务器上存储着文件系统的部分数据,因此机器失效的情况不可避免，进而此错误检测和快速、自动的恢复是 HDFS 最核心的架构目标。
2. 支持流式数据访问，高吞吐量。
3. 支持大规模数据集。一个部署在 HDFS 上的文件单位可能是 G 甚至是 T,因此 HDFS 限定最小的结点单位为 64M,HDFS 文件大小总是最小节点单元的整数倍。一个节点不可跨机器。
4. 支持简单的一致性模型。在文件创作之后，不再改变。因此要支持一次写入，多次读取的特性。
5. 移动计算优于移动数据。优于数据拷贝巨大，因此将计算程序分布于不同机器。
6. 异构性。对机器要求低，跨机器忽略硬件，能够搭建逻辑集群。

二、HDFS 架构



Namenode 和 Datanode 被设计成可以在普通的商用机器上运行。这些机器一般运行着 GNU/Linux 操作系统(OS)。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，而集群中的其它机器分别运行一个 Datanode 实例。这种架构并不排斥在一台机器上运行多个 Datanode。HDFS 是标准的列存储结构，列存储结构的示意图如下：

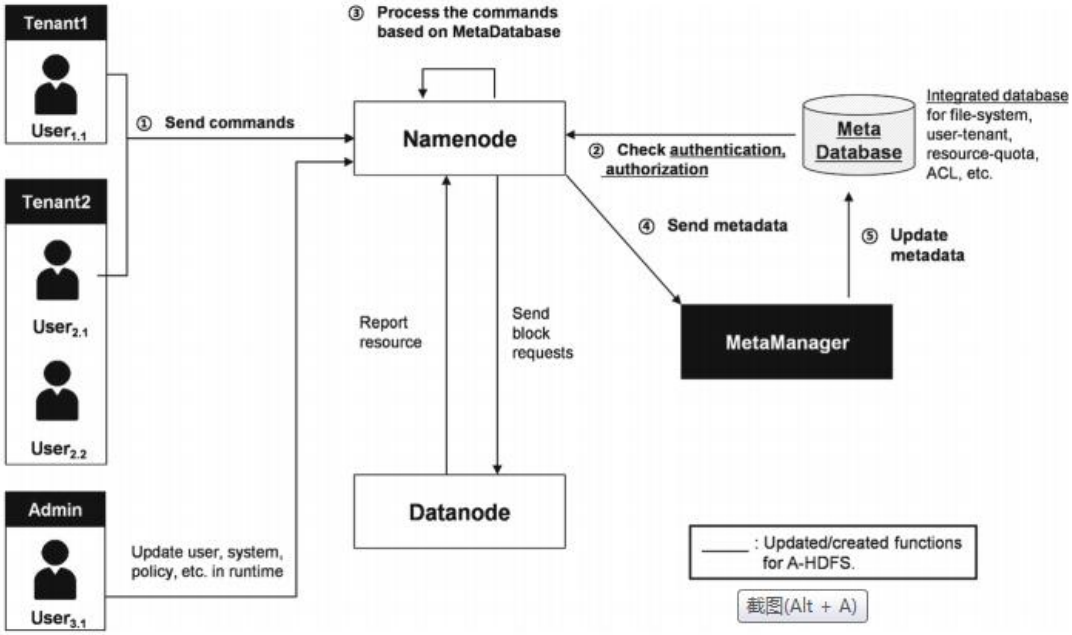


三、HDFS 的压缩算法对比

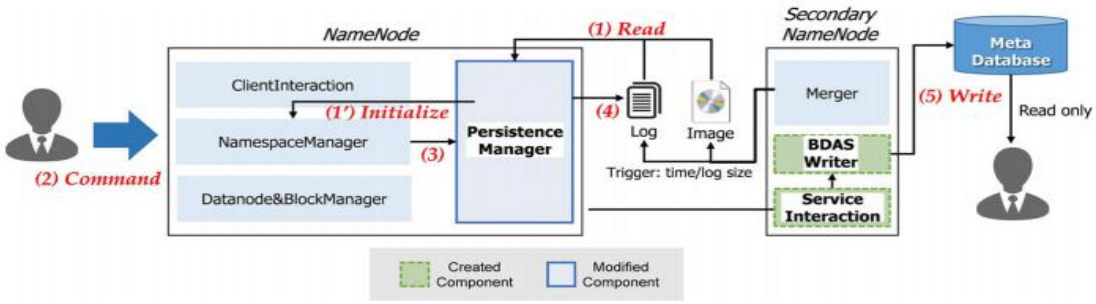
HDFS 中常用的数据压缩算法分为可拆分和不可拆分两类。Snappy 具有高压缩速度和较好的压缩率。它在速度和压缩率之间作了较好的权衡。由于 Snappy 是不可拆分算法,即压缩后的文件不可拆分,它需要在一个特定的文件格式(如 Parquet、ORC)中使用。LZO 和 Snappy 类似,比较注重压缩速度。不同的是 LZO 压缩后的文件是可拆分的.因此相对于 Snappy，LZO 更适合用作一个独立的压缩格式来对 HDFS 上的文本格式的文件进行压缩.Gzip 提供了较高的压缩性能,平均达到 Snappy 的 2.5 倍,但是它的写入性能不如 Snappy.在读性能方面,Gzip 和 Snappy 接近。Gzip 同样是不可拆分算法,因此也需要嵌入在一个文件格式中使用。在部分情况下 Gzip 的压缩效果太好,导致压缩出的数据很小、数据块数很少,所以在执行数据处理任务时的并行度可能会偏低，从而导致数据处理的速度反而降低[38].这个问题可以通过使用较小的数据块来避免.bzip2 提供了非常好的压缩性能,但是其解压性能较差,通常只用于存储空间非常有限的情况。

四、HDFS 的应用、分析及改进

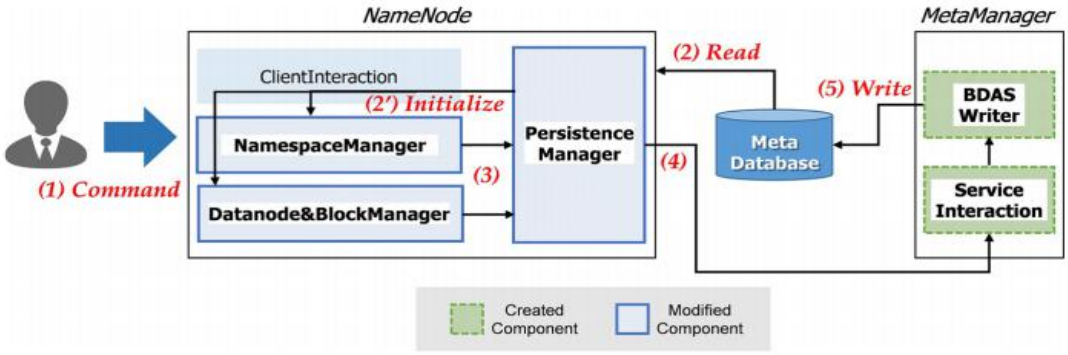
HDFS 改进方案 1：通过改进 HDFS 元数据管理方案，来负责存储和管理大数据。由于当前的 HDFS 基于元数据管理，因此 HDFS 在系统利用率方面还存在许多问题。现有一种基于 RDBMS 的方式改善 HDFS 的功能方面。众所周知，现有的 HDFS 存在五个问题。第一个问题是，因为现有的 HDFS 加载了所有元数据进入 NameNode，因此主内存中的文件（或目录）数可能会受到限制。二是现有的 HDFS 管理的元数据在复杂的映像和日志文件中，这会导致较长的引导时间去查找元数据。三是现有系统只能通过命令来搜索或修改元数据，并且这种基于命令的方案对于分布式的环境具有很高的依赖性。四是由于 Hadoop 管理元数据在单个 NameNode（或在联合模式下的一个或多个单独的 NameNode）中，对 HDFS 的所有访问请求都需要通过相应的 NameNode。这种访问结构严重限制了 Hadoop 的可扩展性。五是所有对文件的查看都需要通过 HDFS，不能用户单独操作文件。综合起来即鲁棒性，依赖性和可扩展性三项问题。现我们考虑一种新型的 HDFS 架构，如图：



MetaDataBase 集成 FSImage 和 EditLog 等多种元数据文件，用户可以通过读取 MetaDataBase 来处理元数据，而不是在通过 HDFS 命令。每一个元数据包含 block,node,和权限等多种信息。我们为 medadata 信息组成的文件构建一个 RDBMS 表。但是对于系统信息数据（例如群集配置和网络设置数据），我们将它们设计为单独的表，因为它们与我们的 HDFS 文件或目录没有的直接关系。我们现在详细说一下 metadata 信息。该 metadata 结构包含:1.集群和节点之间的互联信息，以及安全信息、文件配置和控制访问列表等。2.文件和目录信息：ID，存储位置，所有者和所有者组，用户和组的访问权限，数据块数，块位置，块大小，创建/更新时间等。同时，改进 HDFS 使用 Kerberos 的安全性机制，但是涉及到检索用户信息时，则通过 metadata 检索用户和用户组信息。在改进 HDFS 时，应该循序渐进，反复测试。改进步骤如下：1.在原始元数据文件和 RDBMS 表中都复制了元数据。也就是说，我们仍将元数据保存在 FSImage，EditLog 和 XML 文件中，同时存储相应的元数据。元数据放入 MetaDatabase。先确认数据是否在 RDBMS 中，如果在，则原始元数据与 MetaDatabase 中的数据进行比较，查看是否有损坏或者丢失。步骤如下图：

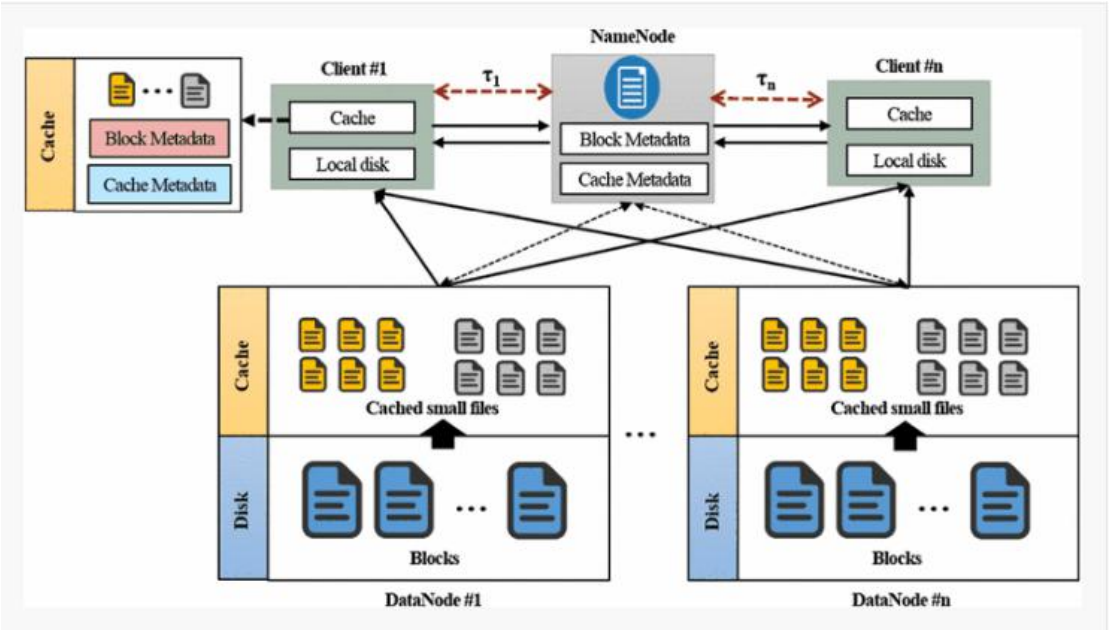


2. 从系统中删除原始元数据文件，改成仅仅使用 metadatabase，即删除 FSImage 和 EditLog 等信息，直接使用中的 MetaDatabase 管理文件。步骤如下图：

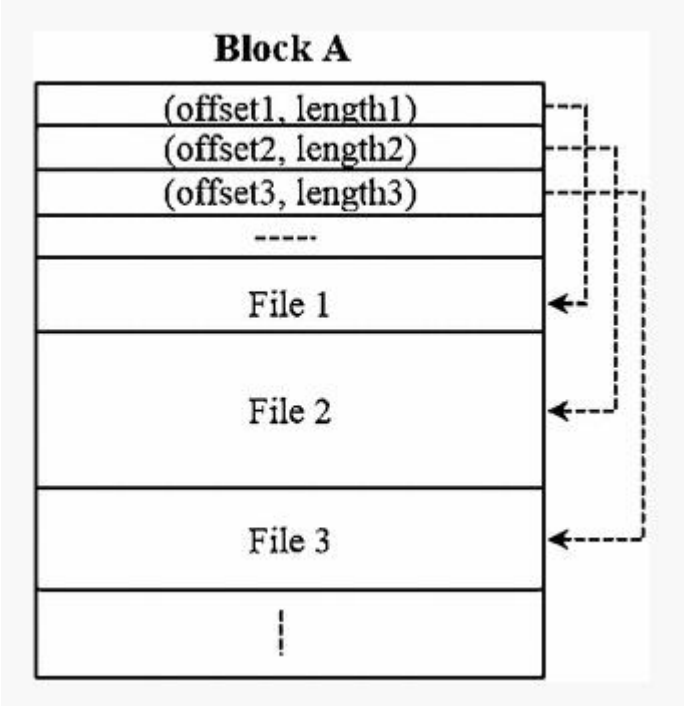


由于改进 HDFS 系统仅加载重要信息来减少 HDFS 启动时间，因此对系统效率的提高还是有显著的影响的。

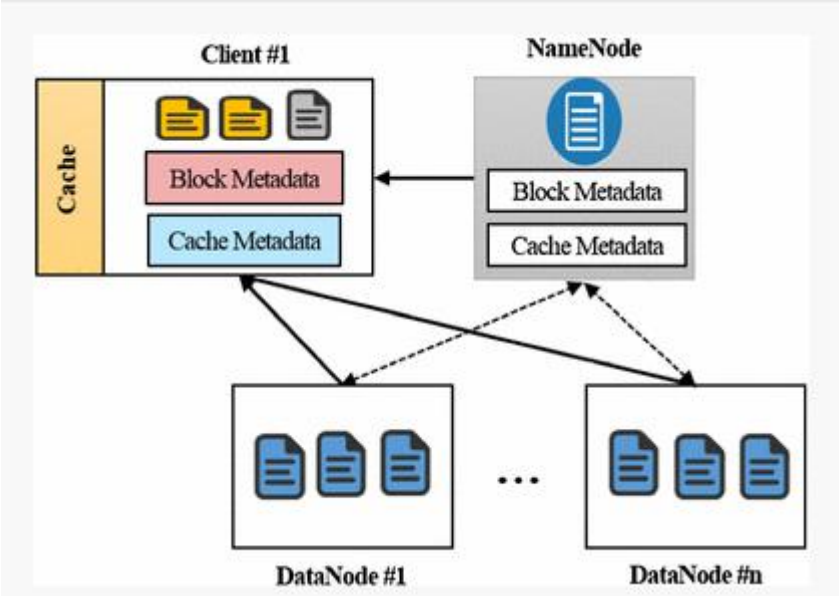
HDFS 改进方案 2：众所周知，HDFS 是针对大文件存储，因此我们提出了一中分布式缓存方案来存储 HDFS 中的小文件。该方案大致方法为，通过在一个块中组合和存储多个小文件的方式来减少要在 NameNode 中管理的元数据的数量。 除此以外，它通过使用客户端缓存和 DataNode 缓存来维护有关请求文件的信息，并且同步客户端缓存的元数据，减少不必要的访问时间。客户端用来缓存维护用户请求的小文件和元数据，每个 DataNode 缓存维护用户经常请求的小文件信息。HDFS 小文件的应用场景为小文件但数量多的情况。现有的 HDFS 对于小文件的读取是使用单个缓存，因此它没有办法维护大量的元数据和小文件。如果我们使用多个 datanode 缓存就可能解决这一问题。这样就形成了一个分布式的缓存系统，它可以缩短小文件的缓存时间。因此，我们可以一下一种方案：应用缓存元数据来提高 HDFS 中访问多媒体小文件性能的分布式缓存方案。还有一种情况，针对 HDFS 存储大文件的特点， 即将小文件合并成大文件，减少 namenode 的负载，并使用预取方法来提高访问性能。现根据以上两种情况，提出了扩展 HDFS 系统，即 EHDFS 系统。它和原有的 HDFS 的区别是它减少了 namenode 主内存中的元数据。所以方式为从单个大文件访问单个文件。如果文件的元数据在客户端存储，就不向 namenode 发送请求信息了。考虑到相同目录的文件依赖性，我们可以将同一个目录的文件进行合并。然后我们将小文件进行缓存。由于访问时间和次数会影响缓存效率，因此开发了一种心得缓存策略：NRU-LFU。 HDFS 的缓存机制 HDCache 建立在 HDFS 的顶部，由一个客户端库和多个缓存服务组成。 HDFS 会构建一个共享内存，它的加载文件会被缓存在共享内存中，以供客户端库直接访问。 加载文件的到期时间可以由客户端自主设置。 如果超过了到期时间，则会根据网络流量，系统工作负荷和文件访问频率来进行处理。NameNode 将元数据存储在 DataNode 的内存中。 当 NameNode 内存已满时，它不会在群集中添加其他节点以满足进一步的存储需求。那么的具体缓存方案是什么呢？即客户端缓存和 datanode 缓存同时进行，减少对 namenode 的通信，对 DataNode 缓存中维护的文件的缓存元数据进行额外管理。 当每个客户端自然地调节与 NameNode 的通信周期并更新缓存元数据时，使用最新信息更新在客户端缓存中维护的缓存元数据。缓存的体系结构由 hadoop 集群和访问该集群的客户端组成，同时包含块的元数据和缓存元数据。每个 DataNode 缓存都维护用户请求的小文件。客户端缓存还维护用户请求的小文件，块元数据和缓存元数据。因为我们需要块元数据来标识包含所请求的小文件的 block， 需要缓存元数据来标识包含小文件的 DataNode。具体的结构如下图所示：



每个 DataNode 缓存中的小文件的架构，从而管理 DataNode 缓存以及 NameNode 中的文件上的缓存元数据。 每个 DataNode 缓存都维护许多客户端经常请求的小文件。 在这里，NameNode 的块元数据示出了在每个节点中维护的块的位置信息，并且缓存元数据包含了在每个节点的缓存中维护的小文件的位置信息。 该方案减少了与 NameNode 的通信，因为客户端可以使用缓存元数据直接访问相应的 DataNode 缓存。客户端缓存除了保留小文件外，还维护缓存元数据，这样就会阻止客户端经常请求的元数据。 使用这种结构，当客户端请求一个小文件时，系统会在其自己的客户端缓存中搜索所请求的文件。如果没有请求的文件，它将在 DataNode 缓存中查找该文件。 如果 DataNode 缓存中没有请求的文件，则将请求的文件从 DataNode 磁盘块中取出并保存在缓存中。使用此过程，提出的方案克服了针对大量数据进行优化的现有 HDFS 的局限性，并有效地存储和管理小文件。上面的结构只是说明通信量该怎样减少，但是涉及到元数据和小文件，则在下文进行叙述。当需要管理大量小文件时，存储小文件的块数和 NameNode 中要管理的元数据的数量会增加，这会造成小文件数目的瓶颈。上文提到将相同的小文件进行合并，以下是合并结构：



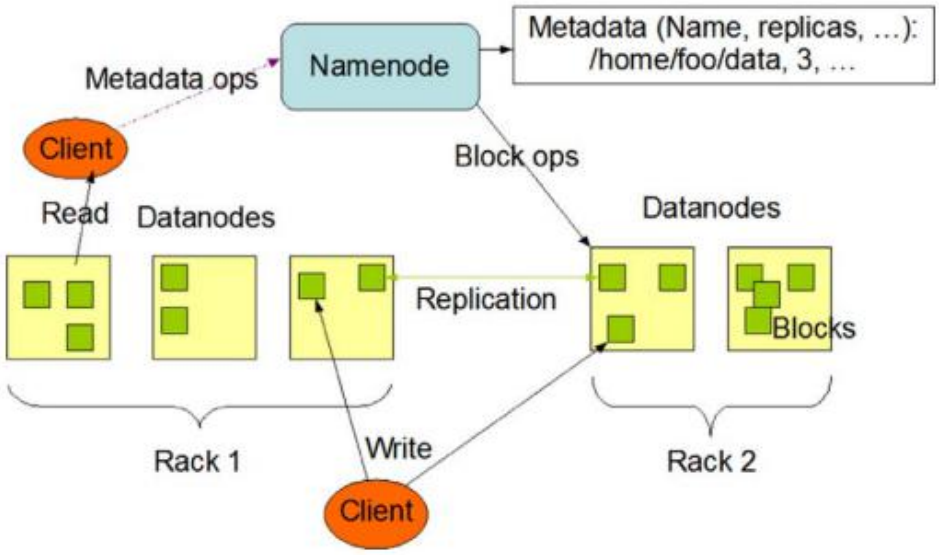
下面解释元数据的具体存放：



如上，显示了将元数据存储在客户端缓存和 NameNode 中的过程。 将小文件组合到一个块中并存储在 DataNode 中后，NameNode 将存储和管理每个块的块元数据。 当用户请求一个小文件时，该方案通过与 NameNode 的通信来检查块元数据，然后访问包含请求的小文件的块，并将该小文件和块元数据保存在客户端缓存中。 此时，包含用户访问的 DataNode 也会将小文件保存在其自己的缓存中，并将保存的缓存元数据与块元数据一起保存在 NameNode 中。 如果用户再次请求的小文件，并且其存储在 DataNode 缓存中，它将通过与 NameNode 的通信检查缓存元数据，然后访问相应的 DataNode 缓存并将请求的小文件和缓存元数据保存在客户端缓存中。该方案减少了 NameNode 中发生的处理负载。并且，随着客户端使用心跳周期更新缓存元数据，小文件访问速度也将提高。综合以上，我们提出了分布式缓存管理方案，该方案缓存元数据同步，以提高小文件访问速度，并且最大程度地降低 HDFS 中 NameNodes 的网络负载。同时，通过在每个 DataNode 缓

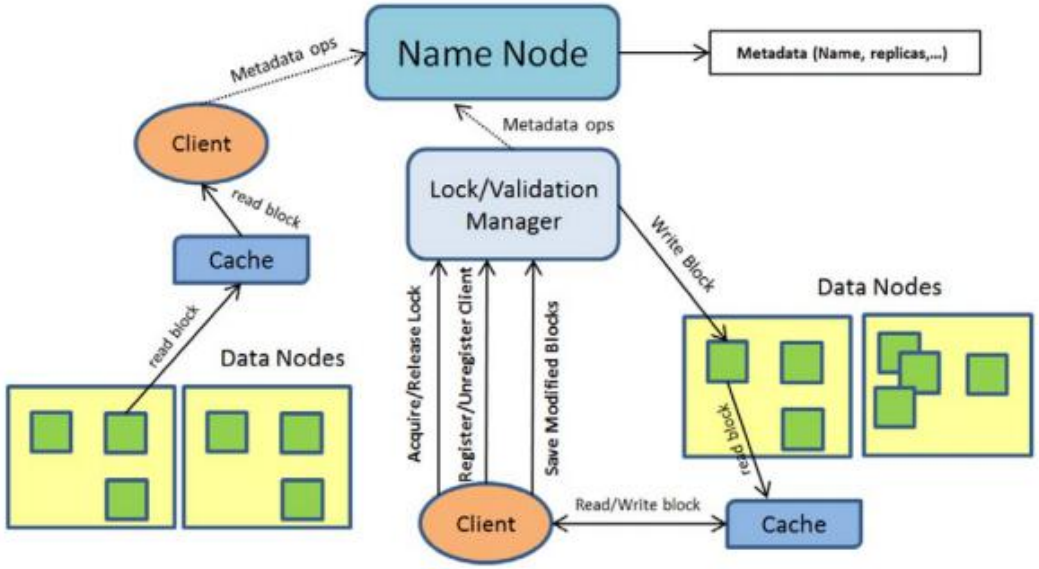
存中维护用户经常使用的小文件并在 NameNode 中管理它们的缓存元数据，因此所提出的方案大大降低了 DataNode 对磁盘访问的频率。此外，该方案通过在客户端缓存中维护块元数据和缓存元数据来最大程度地减少与 NameNode 的通信，并通过应用缓存元数据更新策略来减少不必要的文件访问，在该策略中，每个客户端根据更新自动调节与 NameNode 的通信。这样可以减少小文件的访问时间。性能评估结果表明，该方案在小文件访问方面优于已有的 HDFS 方案。我们还可以在该方案基础之上，研究使用高速缓存中的常用数据重定位的负载平衡方案来近一步探究对小文件存储的影响。

HDFS 改进方案 3：我们知道，HDFS 用户指定用于对给定文件进行分区的块大小，但是它仅发出顺序读取和附加操作。它不允许用户执行随机读取和随机写入操作。我们可以提出了一种增强的 HDFS（REHDFS），它可以探索不同的块放置和块读取策略，还可以实现随机读取和随机写入操作。我们知道，HDFS 配合 hadoop 和 MapReduce 使用。，这些应用程序可以使用低成本的硬件对大数据进行并行处理。传统的基于 MapReduce 的算法包含两个阶段。第一阶段称为 Map 阶段，第二阶段称为 Reduce 阶段。 Map 阶段将键/值对作为输入，并且预计将产生一组中间键/值对。MapReduce 阶段通常会汇总不同映射器生成的输出，我们成为收集。在执行 MapReduce 程序期间，Hadoop 将 Map 和 Reduce 任务分配给集群中的相应服务器。通常，大数据分为多个块，Map 任务在一个或多个服务器上启动，并且每个服务器处理一个或多个块。Reduce 任务的输出经过排序和混洗，然后定向到一个或多个 Reduce 中。 Hadoop 服务器从 Reduce 程序收集结果，并向发起该任务的客户端进行状态返回。HDFS 传统架构如下：



从现有的架构中，可以看到 HDFS 体系结构基于主/从体系结构。Hadoop 集群包含一个 NameNode 和一个或多个数据节点。名称节点管理以下所需的元数据；并且对文件系统名称空间进行建模；还可以定位承载文件内容的节点；并实施权限访问控制。namenode 还提供对目录和文件的打开、关闭、重命名和删除等；重定位给定文件的特定段（块）的数据节点。文件的内容分为一个或多个块。一个块存储在多个数据节点上支持容错。datanode 节点管理一个或多个 block。数据节点服务于 HDFS 客户端发出的块读取和块写入请求。datanode 节点还满足创建，删除和复制块的 namenode 节点请求。典型的 Hadoop 部署使用两个或更多机架来容忍机架故障。但是一台机器上也可以部署。 当前的 HDFS 不支持对部署在 HDFS 中的文件进行随机访问和随机写入的操作。REHDFS 研究使用不同块大小的影响，和研究不同块放置策略的优点和缺点。一旦文件块被复制并存储在各各数据节点上，客户端就需要一种透明地访问文件内容的方法，这样才能做到随机访问。REHDFS 评估了几种策略，以实现客户端 API 来访问被布置在 HDFS 中的文件内容。访问透明性是主要设计目标。 HDFS 中存

储的文件只能顺序访问或只能附加。 REHDFS 提供了一种随机访问文件内容的方法，并提出了几种执行随机读写操作的不同机制。REHDFS 的逻辑结构如下：



namenode 管理元数据。 datanode 托管一个或多个 block。 最终用户使用客户端模块提供的 API 来访问 REHDFS 中放置的文件。这与当前的 HDFS 模式大致相同。但是 API 有很大区别。 客户端模块隐藏了如何与其他 REHDFS 组件进行交互的详细过程，并提供了访问透明性。 客户端模块使用缓存模块来缓存从数据节点检索到的块的信息。 如果客户端请求的块存在于高速缓存中，则将从高速缓存中为其提供服务。 否则，则将托管数据节点检索到的块添加到缓存中，然后将块的副本返回给客户端。 当客户端修改文件的特定部分时，与之匹配的缓存块将更新。锁定/验证管理器是添加到 HDFS 体系结构中以支持随机写入操作的新组件。它也是 REHDFS 的亮点。 它支持悲观模型和乐观模型，以实现随机写操作。乐观模型直接提交，将来回滚。悲观模型应对冲突会进行上锁。在悲观模型中，要修改文件，客户端需要从“锁/验证管理器”中获取锁。 在乐观模型中，客户端可以更改文件的缓存块，而无需获取对其的锁定。但是当其他客户端在同一文件上运行时，保存更改的请求可能会失败。综上可知，设计 REHDFS 时必须考虑位置透明性、访问透明性、乐观和悲观模型，以及锁定以支持写操作。针对位置透明性和访问透明性， REHDFS 中存储的文件分为多个块，每个块有多个副本位于不同的数据节点上。名称节点是 Hadoop 集群中一个单独的节点，每个文件的元数据都在此节点中维护。元数据包括详细信息，例如文件系统名称空间，用于决定文件分区的块的大小，和文件的大小以及存储每个块的数据节点位置等。 REHDFS 中存储的每个文件都有一个唯一的文件路径，该路径以众所周知的前缀（例如 dfs：“根/”）开头，以使其与其他文件系统中使用的文件路径区分开。 REHDFS 文件的文件路径隐藏了给定文件的块存储位置的详细信息，并提供了位置透明性。客户端模块联系名称节点以检索与给定文件关联的元数据。最终用户使用客户端模块 API 对给定文件进行操作。客户端模块提供类似于本地文件系统的 API，从而提供访问透明性。现在我们来考虑数据节点上块的组合放置问题。我们知道，大多数 Hadoop 集群部署使用两个或更多机架。 每个块都需要复制到每个机架的至少一个数据节点中，这样能够获得覆盖机架故障的最大故障转移。 当客户端在不属于托管块的任何机架的节点上运行时，该客户端称为远程客户端。 否则，客户端在这里称为本地客户端，客户端运行所在的机架称为主机架。 在每个非主要机架上放置一个块的一个副本后，其余的块副本将放置在主要机架的不同数据节点上。 对于本地客户端方案，为避免机架间切换时间，客户端模块使用主机架的数据节点来检索块，从而减少通信量。 对于远程客户端方案，客户端模块随机选择一个可用机架来检索块。那么实际上在实现上，块的放置决定了整个系统的容错能力。具体算法如下：

Begin
$blockCt = ceiling(N_f/BS_f)$;
$nextDataNode = 1$;
For Each i from 1 to blockCt **do**
    Place one copy of $block_i$ on each non-primary rack;
    $replica = 0$;
    For Each replica of $block_i$ from 0 to $R_f - r$ **do**
        Place $block_i$ on the nextDataNode of primary rack;
        $nextDataNode = (nextDataNode\%M_f) + 1$;
        $replica$++;
    End for
End for
Return

上述算法过程，输入包含文件位置，块大小，复制因素，数据节点数目，主机架和机架数目还有约束条件。其实现思路就是将要放置在 HDFS 中的文件分为一个或多个块。 根据文件大小和块大小计算所需的块数。 每个非主机架上都放置一个块的副本。 其余副本放置在主机架的不同数据节点上。
在检索数据节点的时候，需要将块放入缓存。系统会“前瞻性缓存块”，方法如下：对于访问的每个文件，客户端都保持访问状态，该状态包括当前文件指针位置和从先前请求中检索到的块的缓存。 当用户访问某个块的内容时，很可能用户在不久的将来也可能希望访问以下块中存在的内容。 前瞻性块计数用于指示需要异步将当前块之后的多少个块带入缓存。 这可以由用户配置。这种预读算法如下：

```
Begin
currentPosition=getValue(fdes,“CurrentPosition”);
blockSz=getValue(fdes,“BlockSize”);
lookAheadBlkCt=getValue(fdes,“LookAheadBlockCt”);
endPosition = currentPosition + size − 1;
startingBlock = curPos/blockSz;
endingBlock = (endPosition/blockSz);
For Each i from startingBlock to endingBlock do
    If (block_i is not in Cache) Then
        Download block_i from a Data Node that hosts;
        Add the block_i to the Cache;
    End If
End for
Copy the content from downloaded blocks to buffer;
If(lookAheadBlkCt > 0 ) Then
    Create a thread to download look ahead blocks;
    Add the downloaded blocks to the Cache;
End If
Return
```

如上述算法所示，客户端计算 read 的块号。 从标识该数据块的数据节点检索到的每个已标识的数据块，如果该数据块不在高速缓存中，则将其放置在高速缓存中。 如果指定了先行块计数，则客户端启动一个线程以异步方式检索当前块之前的块，将其加入缓存。既然数据块有多个副本，那么现在需要一种策略使得下载数据块的通信量最小。 REHDFS 支持三种策略，即“第一”，“随机”和“基于负载”三个策略，以从主要机架数据节点集中选择一个数据节点以下载该块。第一模型指的是，选择托管该块的主机架的数据节点列表中存在的第一个数据节点。但是，当多个客户端要访问同一块时，所有客户端仅联系该块的（主机架的）数据节点列表中的第一个数据节点。 即使在许多数据节点上复制了一个块，每个客户端也仅联系列表中的第一个数据节点，众多客户端的并发请求会在在同一数据节点上排队。随机指的是，众多客户端的并发请求会在所有的副本数据节点上排队。基于负载的方法需要事先知道各个数据节点的通信能力。下面我们来看随机访问的实现：从文件的开头查找文件最终的文件指针的位置。 当文件上传到 REHDFS 时，命名节点会为文件的每个块分配一个全局唯一的块 ID。 使用元数据检索与块号相对应的块 ID。 当高速缓存中不存在该块时，将对托管该块的数据节点进行标志。最后，调整文件指针以反映新的查找位置。在悲观模型中，缓存文件的一个或多个块的所有其他活动客户端必须全部同意才会修改文件。成功执行写操作后，将向所有活动客户端通知有关修改的块的信息。在乐观模型中，需要更改文件的客户端检索块并修改其缓存的块。当客户端要保存更改时，将根据其他任何客户端所做的更改来验证更改。如果客户端所做的更改与任何其他客户端的更改冲突，则这些更改将回滚。更改检测的粒度可以在文件级别或块级别。算法在这里不再赘述。以上即实现了随机读写和预读功能。

HDFS 的改进方案 4：我们考虑 HDFS 多部署在云端，因此，对故障处理和容错处理就要变得十分重要。容错是分布式存储系统（例如 Google File System 和 Hadoop Distributed File System

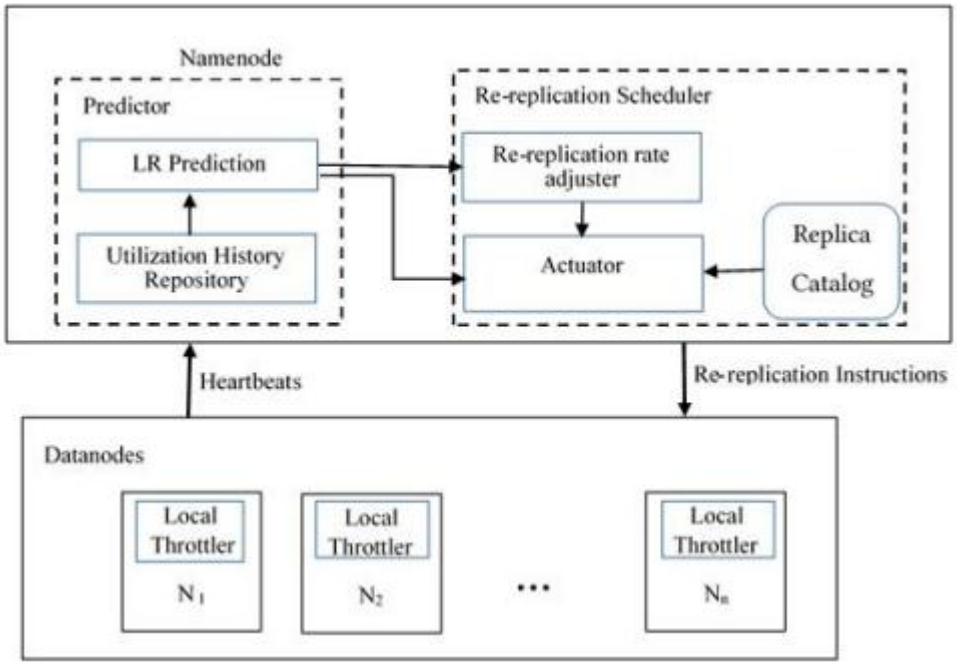
（HDFS））等分布式存储系统的核心方面之一，因此使用三重复制是因为它可以抵抗所有类型的服务器故障。数据恢复困难，主要有以下原因：

1.故障的数据节点保存大量数据，数据恢复过程可能需要花费大量时间。数小时后，如果在成功恢复所有失败的块之前又发生另一个节点故障，则很有可能又发生数据丢失。

2.如果几个数据节点在一次重大故障期间同时发生故障，则将导致恢复过程有不必要的资源浪费。

3.对于具有常规群集作业的资源存在竞争，这些作业具有严格的服务级别目标（SLO），导致它们比平时花费更长的时间来完成。为了避免争夺资源，资源调度程序可以中止作业并将其重新分配给其他节点。但是，HDFS 中的数据复制不仅用于容错，而且还用于数据局部性使用。因此，将用户的作业重新分配到其他数据节点可能会导致丢失数据局部性的风险，从而导致作业性能下降，因为可能会涉及数据远程传输。

在为用户的应用程序工作负荷提供服务的活动集群中触发重新复制时，必须解决两个主要问题，即可靠性和性能。 执行复制时，可靠性和性能之间需要权衡取舍，如果仅考虑性能，则可以以最低处理速率运行复制过程，以最大程度地降低对常规集群作业的性能影响，但这会降低可靠性。 相反，以最高处理速率运行恢复过程会增加对常规作业的性能。我们理解的恢复过程，实际上就是一种数据复制过程。该过程按常规速率赋值，并且无需考虑可靠性和性能问题。而且，原数据节点和目标数据节点如果进行随机选择，则会导致通信量增加，结点工作负载不平衡。该问题的解决方案通过最小化每个复制节点数据传输量之间的差距，达到负载平衡。现有两种新的复制方法，主动复制和延迟复制。每种方法都有其各优缺点。快速复制会消耗大量资源，并在短时间内完成复制过程，这会提高可靠性，但同时也会对常规群集作业产生性能影响。延迟赋值的方法会根据整个群集的平均利用率将复制工作负载转移到将来的一个时间，但是这样会减少可靠性。因此，考虑两种方案的优缺点，现在需要一种其他的方案对以上方案进行取长补短，即积极主动复制法，该方法根据常规群集工作负载的资源利用状况分配了复制工作负载，以确保所有服务器在复制阶段，工作量相对来说比较平衡。此方法使用每个数据节点的预测的资源利用率来确定分配给每个数据节点的副本数目。它使用局部回归的方式来预测未来的资源使用情况。预测方法是：根据可用副本的数量和每个副本的受欢迎程度，将发生故障的数据节点上的数据块分为四个不同的优先级组。哪个副本的受欢迎程度高，哪个副本就应该优先被回复。同时，基于对数据中心服务器在一天中不同时间段内资源使用情况的学习，我们开发了延迟重复复制，该延迟复制基于整个群集的当前资源利用率，转移复制工作负载，并在结点空闲期间执行了复制。如果资源未得到未忙碌，则可以快速执行复制。下面是具体的结构，主要包含两个部分，复制率调整器，用于调整复制节点的数量。每次迭代中要复制的结点和一个执行器，该执行器决定分配给每个数据节点的数据块数目，并将重新复制指令发送到数据节点。



重复率计算公式如下：

$$R_{rate} = Max.R_{rate} - (Predicted\ Avg.Util*(Max.R_{rate} - Min.R_{rate}))/100$$

最小速率是 HDFS 的默认复制率；它以两倍的速度重新复制块系统中的数据节点数量。最小速率以恒定稳定的方式分配副本，其速率可以通过静态控制配置参数来加快或减慢速度，但是在实际复制时不能随资源利用率的变化而自动控制它复制速率。

HDFS 改进方案 2：可以对副本进行改进。由于数据节点的利用程度不同，因此可以根据数据的流行程度更改文件的复制因子，从而可以按需保存更多的副本。首先确定每台计算机的处理能力，可以根据计算机运行的容器数量判定每台计算机的计算处理能力。但是这篇论文我没有看懂决定复制因子的算法过程。

HDFS 应用方案 1：根据重叠数据的要求，我们提出针对 HDFS 和 MapReduce 编程模型的两阶段扩展，称为 XHAMI，可以在图像处理领域进行广泛应用。它大大提高了系统性能并简化了应用程序开发过程。并且，可以在现有 HDFS、Mapreduce 情况下正常工作。该应用的提出背景是大数据的累计会随着时间的推移成为海量数据，处理海量数据是一项艰巨的任务。它可以作为遥感影像的应用， 由于卫星数量的增加和 RS 中技术的进步，其数据量每天都在大幅增加。论文两种扩展包含 XHAMI IO 读写 API 和 MapReduce 分布式过程改进两个部分。在原始方式中，客户端使用 XHAMI I / O 功能进行读取或写入数据。 XHAMI 将客户端请求转换为 create（）或 open（）并将其发送到分布式文件系统中。分布式文件系统查找节点名称来决定块的物理位置。每一个块结点返回结点的读取地址。分布式文件系统也会返回输入输出流给 XHAMI，使其对文件进行读写操作。然后，XHAMI 开始检查文件格式。然后才会进行文件读写操作。XHAMI 读写的方式支持原有 HDFS 的流水线的读写方式，即每个块都有相应的头信息。在读写完成后发送文件关闭请求，并将关闭状态发送给结点。具体的执行过程如下：

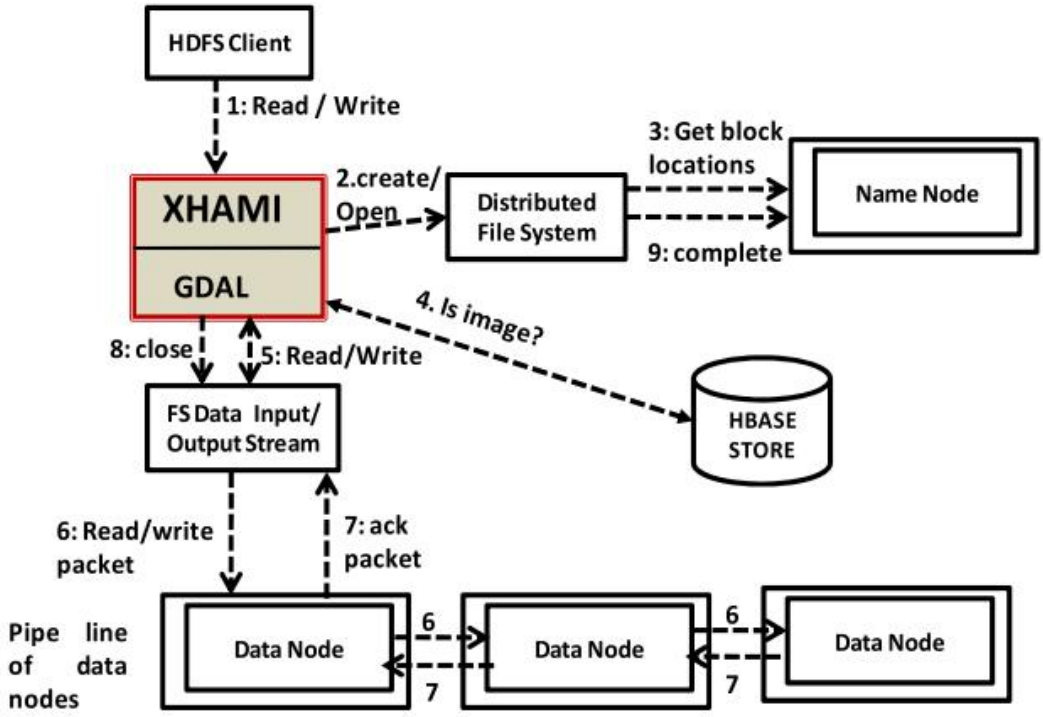


Figure 2. XHAMI for read/write operations.

关于 XHMAI 的文件系统方面，它是基于 Hadoop 的大规模特定于域的数据密集型应用程序设计的软件开发套件。它提供了用于数据组织和用于基于 MR 的处理通过隐藏图像组织和处理的许多底层细节，简化了开发和快速的应用程序设计。XHAMI 从标准 Hadoop 包扩展了 FileInputFormat，并扩展了几种方法的实现，以隐藏用于 HDFS 数据组织的数据低层处理，方法包括，设置重叠，获得重叠，读取数据等，并且它可以屏蔽底层数据细节。并且 XHAMI 也向上支持继承，会提供一些基类给开发人员。它实现了几种图像处理方法，用于设置图像的标题和元数据信息，在图像处理方面，如果开发人员使用基于 Hadoop 的图像处理应用程序，则可以通过设置自己的块信息头实现来扩展图片处理类。在以上的依赖 XHAMI 文件系统将数据读取完成后，就可以进行任务转发，执行 MAP 过程了。下面例举图片处理的 MAPREDUCE 过程。以图片边缘检测为例，边缘检测有两种方法：梯度法和拉普拉斯算子法。 梯度方法通过在图像的一阶导数中寻找最大值和最小值来检测边缘。 拉普拉斯算子方法在图像的二阶导数中搜索零交叉点以找到边缘。 边缘是具有坡道的一维形状，计算图像的导数可以突出显示其位置。 在地图功能中，对于边缘检测，由于不执行组合器和归约功能，所以不需要汇总各个地图功能。我们所认知的原有的 Hadoop 实施与 XHAMI 实施之间的区别如下：在前者中，数据按块进行分段组织，并且没有重叠数据出现。 因此，将难以处理一个块的边缘像素，并且，在将其发送到地图函数进行处理之前，应该获得两个块并计算重叠像素。 而且，将难以防止像素分裂。 但是 XHAMI 会隐藏所有此类低层次的数据组织细节，例如线条或像素重叠，像素内没有分割，将图像以块的块数以及块的标题信息的形式进行组合。这就是 XHMAI 在图片处理应用中的优点了。

有关于验证 HDFS 的性能方面，我们可以使用流程代数建模的方式。我们是用顺序过程 CSP 对 HDFS 进行建模和分析。还可以对心跳机制进行建模，使用模型检查器 PAT 进行模型的构建及验证。该模型通过通信框架 CSP 在 HDFS 中读写文件，同时，我们借助心跳机制，群集中的每个 DataNode 都会定期将心跳信息发送到 NameNode。 最后，我们使用模型检查器过程分析工具包 PAT 来验证所实现的模型是否满足一些重要性能，是否有死锁，是否支持一次性写入等等。我们知道，HDFS 的通信架构包含三部分：namenode,datanode 和 HDFSclient。 NameNode 负责管理文件的元数据，并且管理 HDFS 客户端对文件的访问。 DataNodes 存储文件的实际数据，并与 HDFS 客户端

通信以直接读取文件和写入文件。 除此以外，NameNode 会定期从每个节点接收心跳集群中的 DataNode 并向其发送命令。我们将通信流程抽象为如下：

$$HDFS =_{df} ClientNode \,||\, DataNode \,||\, NameNode$$

Clientnode 又分为 Clientread 和 Clientwrite。客户端的读包含向 NameNode 读和向 DataNode 读两部分。客户端读的内容就是文件信息。我们将文件信息抽象为：

$$fileInfo =_{df} FileName\#OffSet\#Length$$

如果再细分，我们文件实际存储的块的信息，分为块的聚集列表和数据节点集合的信息。BlockList 代表整个文件的块的存储信息，DataNodeInfo 代表存储这些块的数据节点的信息。我们将其抽象为：

$$blockLocation =_{df} blockList\#DataNodeInfo$$

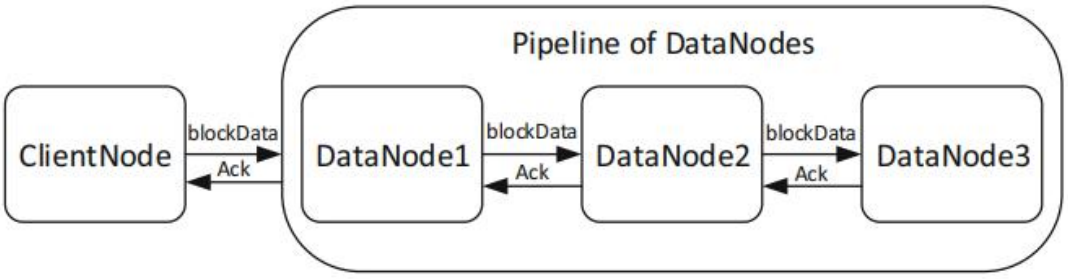
同样的抽象结构体还包含，块信息和块包，和块包里面的块信息等等。

$$blockInfo =_{df} blockId\#GenerationStamp\#Length \\ \#StartOffset\#ClientName$$

blockInfo 通常是由客户端传向数据节点。blockId 是 block 的数据号，Generationstamp 代表时间戳，length 代表数据长度，startoffset 代表该块的起始位置，clientname 代表该块的用户。

$$blockData =_{df} ResponseHeader\#DataPacket$$
$$ResponseHeader =_{df} Checksum.Header\#Offset$$
$$DataPacket =_{df} PacketLen\#Offset\#SequenceNum$$

同样对于数据的写也是如上过程，下面我们展示用户和 datanode 数据段交互的过程。



下面我们看一下心跳模型。

$$Clock(i) =_{df} (tick \rightarrow Clock(i+1)) \\ \Box\, (time?request \rightarrow time!i \rightarrow Clock(i))$$
$$\text{where } i \geq 0 \ \wedge \ i \in \mathbb{N}.$$

当时间 time 未到既定时间，则不断记时。若到时间，则发送请求信息，这里主要是 datanode 向 namenode 发送自己的信息。如下抽象模型代表发送信息的主要内容：

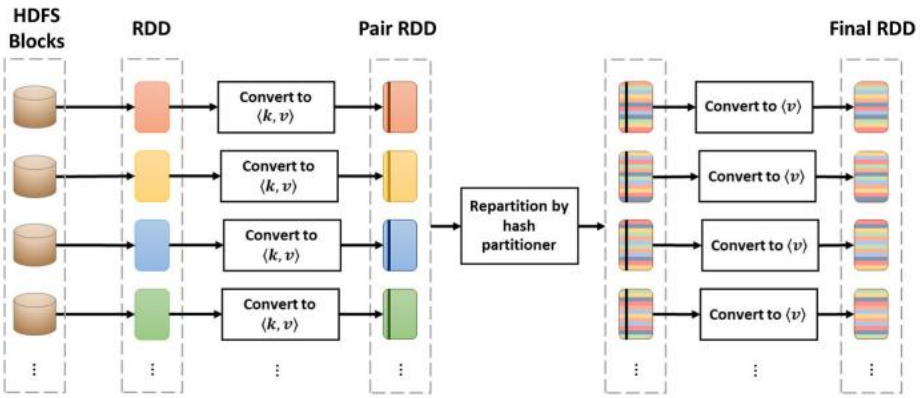
$$heartbeat =_{df} nodeReg\#capacity\#dfsUsed \#remaining\#xmitsInProcess\#xceiverCount$$

如下分别代表数据节点的信息，存储能力，当前已经使用的存储情况，仍然可用的存储情况，当前主要进行块复制的线程数目，当前使用的线程数目。namenode 根据 datanode 发过来的心跳进行解析，并返回一定的指令信息。指令信息的抽象模型包含：

$$command =_{df} dnatransfer\#dnainvalidate \#dnashutdown\#dnaregister \#dnafinalize\#dnarecoverblock$$

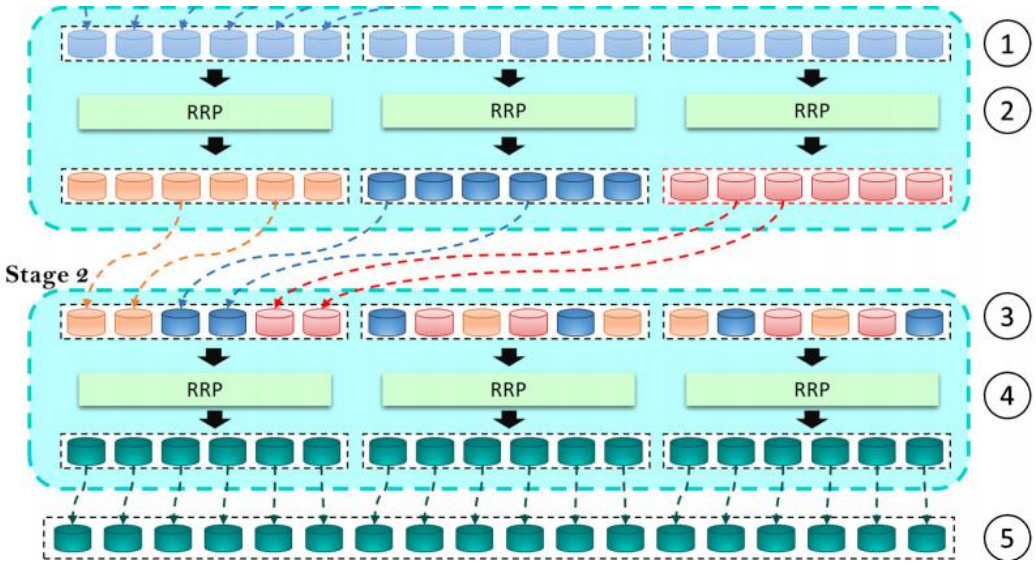
Dnatransfer 代表将数据复制到其他节点。Dnainvalidate 代表删除快。Dnashutdown 代表关闭数据节点。剩下的以此代表重新注册数据节点，升级数据节点，恢复块操作。通过以上流程可以验证，1.HDFS 死锁自由。2.互斥能够实现不能同时写。3.HDFS 能够实现一写多读。4.鲁棒性验证，可以实现即使有故障，也能可靠地存储数据。由此我们可以得出结论，HDFS 支持死锁自由，互斥，一致性读写和可靠存储。但是由上可知，验证分布式系统仍然是一个巨大的挑战。

HDFS 应用方案 2:HDFS 在 spark 中也有很大的应用。它可以选举出代表 HDFS 数据块做为 HDFS 数据集的代表。我们在使用大数据进行分析时，经常面临数据量超出可用计算资源的尴尬情况。这时候，常用的解决方法是对数据采样。由于我们需要对整个数据进行全面扫描，因此通过逐条记录采样数据的传统方法计算量巨大。但是，如果将海量数据划分为一组数据块，而每个块是一个随机样本数据块，则选择一些块作为样本的话，这样的数据计算时间会大大减少。可以设计一个 RRBlib,包含数据生成器，RRP(随机分割器)和大规模 RRP。数据生成器会生成和分类数据集。生成和分类过程如下图：



RDD 是抽象数据集。中间过程代表 RDD 的重映射。在涉及海量数据分区的时候，尤其是数据规模超过内存资源规模的时候，可以使用大规模 RRP。将文件放入 D 个目录下，每一个块包含 P/D 个数据块，每一个目录会产生一个小目录，用于存放该目录下的随机数据块。可经过多重选择，最后将所有随机样本收到 5 这个目录里面。

如下图所示：



经过实验验证，使用数据采样的方法进行与传统方法相比，准确定波动比较小，并且节省时间，是一种可行的方法。

五、综述论文来源

[1]Heesun Won,Minh Chau Nguyen,Myeong-Seon Gil,Yang-Sae Moon,Kyu-Young Whang. Moving metadata from ad hoc files to database tables for robust, highly available, and scalable HDFS[J]. The Journal of Supercomputing,2017,73(6).

[2]XHAMI - extended HDFS and MapReduce interface for Big Data image processing applications in cloud computing environments. Softw., Pract. Exper. 47(3): 455-472 (2017)

[3]Modeling and Verifying HDFS Using Process Algebra. MONET 22(2): 318-331 (2017)

[4]An efficient distributed caching for accessing small files in HDFS. Cluster Computing 20(4): 3579-3592 (2017)

[5]REHDFS: A random read/write enhanced HDFS. J. Network and Computer Applications 103: 85-100 (2018)

[6]Effects of Design Factors of HDFS on I/O Performance. JCS 14(3): 304-309 (2018)

[7]Avoiding Performance Impacts by Re-Replication Workload Shifting in HDFS Based Cloud Storage. IEICE Transactions 101-D(12): 2958-2967 (2018)

[8]RRPlib: A spark library for representing HDFS blocks as a set of random sample data blocks. Sci. Comput. Program. 184 (2019)

[9]HaRD: a heterogeneity-aware replica deletion for HDFS. J. Big Data 6: 94 (2019)

[10]Heesun Won,Minh Chau Nguyen,Myeong-Seon Gil,Yang-Sae Moon,Kyu-Young Whang. Moving metadata from ad hoc files to database tables for robust, highly available, and scalable HDFS[J]. The Journal of Supercomputing,2017,73(6).