一种新型分布式元数据管理策略①

王丽淑[1,2], 张鸿骏[1,2], 芮建武[1], 杨 骁[1,2]

[1](中国科学院软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 高效、可扩展的元数据管理系统是提高分布式存储系统整体性能的关键. 传统的元数据分配策略会导致元数据负载不均衡, 以及在多进程资源抢占的情况下, 会存在响应处理用户请求效率不高, 存储文件数目受限等问题. 上述问题在高并发、低延迟的数据存储需求中尤为突出. 提出了一个基于一致性 Hash 与目录树的元数据管理策略, 并实现了相应的分布式元数据管理系统: 利用负载均衡算法, 对元数据进行迁移, 保证了粗粒度负载信息收集, 细粒度调整的均衡策略. 多项实验的结果表明, 该策略能实现元数据负载均衡, 降低用户请求处理延迟, 提高分布式系统的可扩展性和可用性.

关键词: 元数据; HDFS; Hash; 目录树; 负载均衡

New Distributed Metadata Management Strategy

WANG Li-Shu[1,2], ZHANG Hong-Jun[1,2], RUI Jian-Wu[1], YANG Xiao[1,2]

[1](National Engineering Research Center of Fundamental Software, Institute of Software, CAS, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Efficient and scalable metadata management system is the key to improve the overall performance of distributed storage systems. Traditional metadata allocation strategies result in load imbalance of metadata, and in the case of multi-process resource preemption, there exists low efficiency in response to user requests, limitation on the number of files stored and etc. The system cannot meet the demands of high-concurrency, low-latency. To solve these problems, a distributed metadata management system based on consistent hash and directory tree, is presented in this paper. It introduces a load balance algorithm for metadata migration which has low cost and fine granularity. Experiments show that this system can achieve the load balance of metadata, obtain low respond latency and effective distribution of metadata. and improve the scalability and availability of distributed systems.

Key words: metadata; HDFS; Hash; directory tree; load balance

随着信息技术的发展, 人们的生活越来越离不开互联网和计算机, 众多的互联网应用每时每刻都会产生大量数据, 面对日益增长的海量数据, 分布式文件系统被越来越多的采用. 在分布式文件系统中, 元数据管理系统至关重要, 据有关资料显示文件系统中所有操作的 50%以上都是对元数据的操作, 因此高效、可扩展的元数据管理系统对提高系统的性能格外重要.

元数据管理策略包括元数据无分割策略和元数据扩展管理策略. 无分割策略就是把文件系统的整个命名空间和元数据放在一个元数据服务器上, 如 GFS[1] 和 HDFS. 扩展管理策略就是按一定的策略将所有元数据分散地存储在多台元数据服务器上.

分布式的元数据服务器设计已得到一定的关注与研究[2,3]. Ceph[2]实现了一个元数据集群, 并用动态子树算法将命名空间树均匀的映射到服务器上, 但这种算法的分割粒度是元数据子树, 粒度较粗, 不能保证负载均衡. Lustre[4]在 2.2 版本上已经实现了命名空间的集群. 目的是将一个目录拆分到多个服务器上, 每个服务器上包含不相关联的命名空间. 文件通过将文件名哈希的方法, 指定到一个元数据服务器上, 但这

① 收稿时间:2016-01-19;收到修改稿时间:2016-03-17 [doi:10.15888/j.cnki.csa.005392]

种方法在增删元数据服务器时会产生大量的元数据移动. HDFS2[3]实现了多个 NameNode 并存的架构, 由用户指定所使用元数据服务器. 这在一定程度上有效的分担单个 NameNode 的存储与操作请求负载, 但该方法不支持 NameNode 间的负载均衡, 元数据迁移及服务器数量的动态维护.

本文提出了一种动态的分布式元数据管理策略, 并在 HDFS 上进行了实现. 这种策略支持各个服务器上有完整的目录树层级, 所有文件可以通过路径进行索引. 文件可以通过哈希方式快速定位服务器所在位置. 系统可根据当前负载情况与运行状态, 计算并调整负载分布情况, 在 NameNode 间对元数据进行迁移. 系统可根据负载峰值动态的添加或删除 NameNode, 提高系统资源利用. 这种方法较好的实现元数据的负载均衡, 并且避免了在增删元数据服务器时元数据的大规模移动.

1　HDFS性能测试

HDFS 是 Hadoop 的系统组件. HDFS 分离文件系统的元数据与应用数据进行操作与存储. 如其他的分布式文件系统, Lustre[4] 和 GFS[1], HDFS 将元数据存储在指定的服务器上, 称为 NameNode. 应用数据存储在其他服务器上, 称为 DataNode. 所有的服务器均通过 TCP 协议进行连接与通信. NameNode 负责处理采集客户端的元数据操作请求, 告知其所需应用数据所在的 DataNode, 同时周期性与 DataNode 交互,汇总各节点状态信息与数据块校验.

由于 NameNode 负载所有用户对元数据操作的请求, 因此, NameNode 的负载能力和操作响应速度决定了系统的可用性与可靠性. NameNode 在初始设计时, 定义的系统负载容量为 10PB[5]. 在实际使用过程中, 负载达到了 14PB[6]. 高负载占用了 NameNode 的大量内存, 严重影响 NameNode 的正常使用, 严重会导致 NameNode 所在服务器发生宕机, 降低了系统的可靠性. HDFS 系统组件 NNThoughputBenchmark[6]是在本地测试 HDFS 性能标准测试工具, 这个测试能在 HDFS 上模拟创建、读取、重命名和删除文件等操作. 我们对本地的调用改为远程 RPC 调用, 更契合实际生产环境. 在单台处理器 Intel(R) Xeon(R) CPU E5-26200 @ 2.00GHz 24 枚, 内存 64GB 的计算机上进行测试, 结果如图 1, 图 2 所示.

图 1 中显示了 NameNode 每秒钟执行次数开始时随用户线程的增加而增加, 而当线程数达到较高水平时, 每秒执行次数保持不变, 即达到了系统最高性能. 图 2 中显示了随着用户线程数目的增多, 用户单项操作的延迟时间有着明显的增长. 由此可以看出, HDFS 的单一 NameNode 设计, 难以满足高并发用户数目低延迟的需求, 分布式的元数据服务器设计是必然趋势.

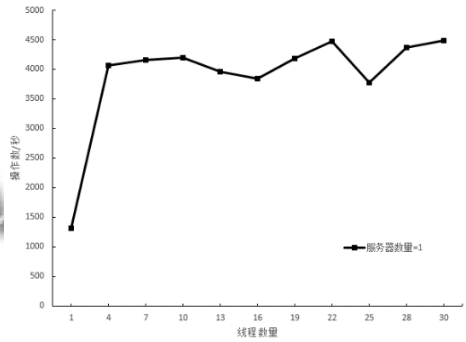

图 1　NameNode 执行操作与线程数目变化关系

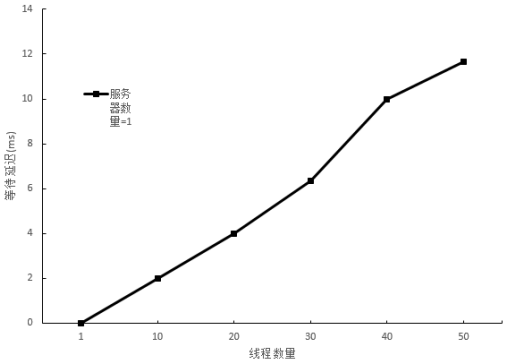

图 2　单项操作延迟时间与用户线程数目测试结果

2　元数据管理策略的设计

2.1　整体架构

针对海量数据的存储应用环境, 本文提出了一种基于一致性哈希和目录树的元数据管理系统架构, 该系统架构结合一致性哈希算法与元数据服务器各自的优势, 将客户端的请求通过一致性哈希算法定位到特定的元数据服务器上, 再在元数据服务器上查询请求对象的元数据信息, 从而确定到具体的数据存储节点上, 这种方法可以实现元数据的均衡分布, 减少元数据的移动, 从而提高系统的效率. 整体系统架构如图 3 所示.

主要包括四个部分. DN 指 DataNode, 作为应用数据的存储节点, 存储文件切分后的数据块. DataNode 周期性向 NameNode 发送心跳信息, 告知其自己在线

并可用. DataNode 也会周期性向 NameNode 发送块报告. NN 指 NameNode, 作为元数据响应以及更新节点, 负责维护全局命名空间, 其中包含文件与文件夹属性. NameNode 维护命名空间树并保存文件中数据块到 DataNode 的映射. NameNode 在一个集群中有一个或多个. Client 指用户应用程序, 支持对文件系统的读、写、删除文件、创建删除目录等操作. Client 可通过调用文件系统接口实现对系统操作. 在整个系统中, 数据流与控制流分开. 即 Client 与 NameNode 交互控制信息, 与 DataNode 交互数据流. NNManager 指 NameNode 的管理节点, 负责周期性收集各 NameNode 状态信息, 维护 NameNode 列表. NNT 指 NameNode Table, 用于存储 NameNode 列表. NNP 指 NameNode Personal, 用于存储所在 NameNode 对应项信息. NNT 与 NNP 均由 NNManger 负责维护和更新.

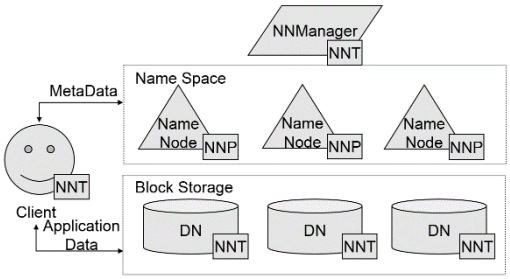

图 3 整体架构

2.2 元数据服务器节点列表管理

NameNode 列表是记录有 NameNode 服务器访问信息的表. 系统启动后, 表内项数不变. 为使服务器负载调整过程更为灵活, 粒度更小, 项数在一定范围内要足够大. 系统初始启动的服务器在一项或多项内, 根据表生成算法填入表内.

表 1 NameNode 列表

index	0	1	2	3	4	5	6
NameNode idetification	A	A	B	B	C	C	D

如表 1 所示, 列表中 7 项, 分别对应有 A, B, C 和 D 等 4 台 NameNode.

在系统启动时, NameNode Manager 会根据各 NameNode 信息及列表项配置信息生成 NameNode 列表. 每个 NameNode 填入表内项数函数如下:

$$U_i = \left\lceil (C - \sum_{1}^{i-1} U_i) / (n - i) \right\rceil \quad (1)$$

其中, U_i 表示第 i 台 NameNode 在列表内所出现的次数, C 表示列表的项数, n 表示 NameNode 总数. 通过此函数计算出的每台 NameNode 出现在表内项数, 极差为 1.

NameNode 列表记录列表项与 NameNode 信息的对应关系. 每个 NameNode 维护自身出现在表中的项的位置, 通过有序二叉树进行索引. 该索引在 NameNode 与 NameNode Manager 上各维护一份.

生成后, NameNode 会向 NameNode Manager 请求所属 NameNode 的列表索引, DataNode 会向 NameNode Manager 请求 NameNode 列表信息. NameNode Manager 在接收到请求后, 会将相应的表发送至请求方. Client 在初始化时, 会向 NameNode Manager 请求 NameNode 服务器列表. NameNode Manager 在接受到请求后, 会将 NameName 列表发送至 Client. Client 会一直缓存此列表, 之后 Client 在向 NameNode 请求过程中, 会附带 NameNode 列表信息, NameNode 同时拥有一份 NameNode 列表信息, 当两个信息经过比较一致后, NameNode 才会响应 Client 发出的请求, 保证了元数据更新的一致性. 倘若不一致, 则被告知访问失效, 会再次向 NameNode Manager 请求列表信息.

2.3 元数据服务器节点选择策略

根据 NameNode 列表选择 NameNode 出现在两个操作中. 一是 Client 根据文件或文件夹信息, 选择 NameNode 进行操作; 二是 DataNode 根据数据块中的项映射, 选择对应的服务器进行块报告.

Client 在对文件夹进行操作时, 将文件夹操作请求提交至所有 NameNode, 并等待所有 NameNode 返回成功信息后, 确认执行成功. Client 在对文件进行操作时, 会根据文件名称进行计算, 计算函数为:

$$NameNode\_Locator=Hash(f) mod\ NNT\_Length \quad (2)$$

其中 NameNode_Locator 表示选择的 NNT 中项, f 为文件的完整路径, NNT_Length 为 NNT 项数. 在操作时, Client 会选择 NameNode_Locator 对应的 NameNode 进行通信与操作.

DataNode 在发送心跳信息时, 会向所有 NameNode 节点发送, 包含是否在线, 并发读写读, 容量, 使用比例等信息. DataNode 在进行数据块报告时, 会根据数据块中的 NameNode_Locator, 选择 NameNode 列表中对应的 NameNode, 对向同一 NameNode 报告的数据生成块队列后, 进行通信报告.

2.4 负载均衡策略

在实际运行过程中, 需要根据各 NameNode 运行情况, 对 NameNode 表进行调整, 实现负载均衡. 负载均衡考察基本单位为 NameNode, 调度基本单位为 NameNode 列表中项对应的元数据, 即粗粒度考察, 细粒度调度. 在元数据操作过程中, 主要考虑用户操作延迟及元数据存储情况, 根据各项进行计算, 统计函数如下:

$$T_i = \eta_1 d_i + \eta_2 m_i \tag{3}$$

其中, $\eta_1+\eta_2=1$; T_i 是在 t 时刻内 NameNode 列表中 i 项的负载指标, d_i 是 t 时刻内 NameNode 列表中 i 项的操作响应延迟, m_i 是 t 时刻 NameNode 列表中 i 项 inode 数目.

通过 NameNode 列表与各项负载指标, 可计算出各 NameNode 节点负载状态, 计算函数如下:

$$w_i = \sum_{k=1}^{n} T_k \tag{4}$$

其中, w_i 是 t 时刻内 NameNode i 节点的负载指标, T_k 是 NameNode 列表中对应的 NameNode k 项的负载指标, 共 n 项.

系统通过 NameNode Manager 收集各 NameNode 节点信息, 包含 NameNode 列表中对应项的指标. 以 NameNode 为负载均衡力度进行考察, 计算公式如下:

$$j_i = w_i - \sum_{k=1}^{n} w_k / n \tag{5}$$

其中 j_i 表示 i 节点在时间 t 内负载均衡指标, w_i 由公式(4)得出, n 为 NameNode 台数. 当 NameNode i 节点的 j_i 大于系统中设定的阈值 α 时, 系统选择 NameNode 列表中 j_i 对应项中负载指标最大值的项, 替换为当前 t 时刻 NameNode 节点最小负载均衡指标对应的节点 m, 并重新进行负载计算, 若仍大于 α, 则重复上述操作. 当系统中出现多个 j_i 值大于阈值 α 时, 重复上述操作.

2.5 元数据服务器节点增删策略

随着用户请求的高峰来临, 系统所相应的用户线程操作, 使 CPU 处于高负载状态, 用户提交的请求与系统本身提交的任务操作均被延迟, 导致系统可用性降低. 由于元数据的历史积累, 内存的物理存储空间有限, 将导致系统崩溃, 无法正常运行维护程序, 导致可靠性降低. 此时, 由于各节点均处于高负载状态, 仅靠对各节点负载进行均衡无法解决上述问题. 因此, 在高负载情况下, 添加服务器, 分担系统整体负载, 可以提高系统的可用性与可靠性.

系统负载由各 NameNode 节点负载指标计算得出, 计算函数如下:

$$E = \sum_{i=1}^{n} w_i / n \tag{6}$$

其中, E 为系统的负载指标, n 为 NameNode 节点个数. 当 E 大于用户设定的负载上限阈值 β 时, 系统添加 1 台 NameNode 节点, 并重新计算 E 值. 若 E 值仍大于 β 时, 系统重复上述操作, 直到 E 小于用户设定的阈值 β. 在增加策略确定后, 进行 2.3 中的操作, 进行负载均衡计算与迁移.

同理, 当系统阈值在大于 β 后, 第一次低于负载下限阈值 δ 时, 系统根据公式(4)选择负载最小 NameNode 节点进行删除, 并重新计算 E 值. 若 E 值仍小于 δ 时, 系统重复上述操作, 直到 E 值大于用户设定的阈值 δ. 在删除策略确定后, 进行 2.3 中的操作, 进行负载均衡计算与迁移.

2.6 重命名操作策略分析

当用户对目录执行重命名操作时, 会引起该目录下文件或文件夹的哈希值发生变化, 从而会导致其元数据存储位置的变化, 会产生元数据的移动, 为了解决这个问题, 拟采取的方案是在每一个元数据服务器上都维护了一张 DPRT(目录路径重定向表格), 用来存储元数据信息不在本地的目录列表. 表中每一项是一对键值<hash(目录路径), 虚拟节点>, 前者是重命名后的目录路径的哈希值, 后者是需要移动的元数据当前的存储位置. 这样我们就可以将元数据的移动推后, 在元数据服务器负载较轻的时候再对元数据进行转移, 提高系统的效率. DPRT 只需要记录路径进行重定向的元数据信息, 并且元数据通过使用哈希算法均匀的分布到了各个元数据服务器上, 所以目录重定向的存储也均匀的分布在各个服务器上, 避免了热点瓶颈. 当用户需要对文件夹执行操作时, 会将文件夹的请求操作提交到所有的 NameNode, 当所有 NameNode 都返回成功时, 则操作成功.

3 系统实现

在文件系统启动时, NameNode Manager 根据系统配置或上次记录, 初始化 NameNode 列表. NameNode 和 DataNode 初始化时, 会向 NameNode Manager 请求

NameNode 服务器列表. 系统运行过程中, 由于负载产生的不均衡, 元数据服务器的数量会发生变化, NameNode Manager 会根据收集的各 NameNode 状态信息对 NameNode 列表进行调整. NameNode Manager 会将调整的内容发送至所有的 NameNode 与 DataNode. Client 在第一次向 NameNode 发送请求时, 会向 NameNode Manager 请求 NameNode 列表 . 在通信过程中, 当被告知失效后, 会重新向 NameNode Manager 请求更新后的 NameNode 列表.

Client 通过使用文件与目录路径来寻找在命名空间中的元数据. 对于文件夹的操作, Client 广播至所有的 NameNode, 等待所有 NameNode 相应操作完成后, 返回成功状态. 这样的目的在于对于每一台服务器, 都有完整的目录层级结构. 对于文件的操作, Client 通过文件名与 NameNode 列表查找出相应的处理 NameNode, 并将请求发送至对应的 NameNode, Client 之后根据从 NameNode 获取的文件数据块所在 DataNode 的信息, 寻找离自身最近的节点获取数据. 然后 Client 直接与 DataNode 进行数据传输. 另外为实现冗余可靠, 数据块有副本散布在各 DataNode 上.

4 实验结果与分析

4.1 实验过程与分析

本文通过实验验证了上述基于 NameNode Table 与目录树的元数据管理系统的有效性. 实验环境: 与 HDFS 的参数设置大致相同, 8 台服务器统一配置为 CPU Intel(R) Xeon(R) CPU E5-26200 @ 2.00GHz 24 枚, 内存 64GB, 同一机架, 千兆网卡, 光纤交换机. 实验方法: 我们采用与 NNThoughputBenchMark 相同的测试方法, 即采用名字生成器生成有序文件名称, 同时将用户线程数作为比较参数. 由于采用多节点进行测试, 我们将原有本地调用, 改为了 RPC 远程调用方式. 我们采用创建操作作为 NameNode 验证操作.

实验中设定的 NNT 的项数为 1000 项, 通过真实操作模拟了不同 NameNode 数目, 不同用户线程数的情况下, NameNode 在单位时间内执行的创建操作数及线程单个操作延迟. 设定默认服务器为 3, 负载值上限 α 为 80%, 负载均衡度下限 δ 设定为 5%, 用户线程数为 6.

采用上述实验环境, 分三个方面来对系统性能进行测试:

1) 以一定频率对数据文件进行创建操作, 测试元数据系统在不同服务器数量的情况下, 系统可执行并发数的变化.

2) 以一定频率对数据文件进行创建操作, 测试元数据系统在不同服务器数量的情况下, 用户操作延迟的变化.

3) 以一定频率对数据文件进行创建操作, 测试元数据系统在增删元数据服务器的情况下, 系统负载情况的变化.

首先测试元数据服务器数量变化对系统可执行并发数的影响, 测试结果如图 4 所示.

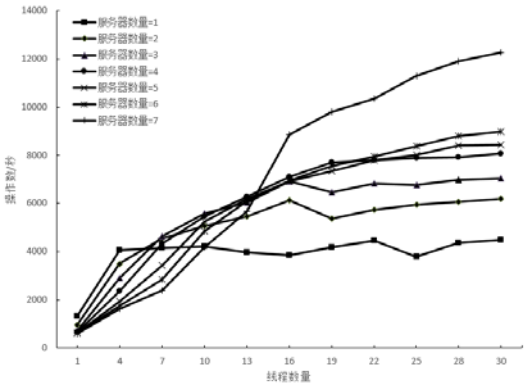

图 4 多 NameNode 执行操作性能

由图 4 中可以看出, 当服务器数目不变时, 在较多的用户线程访问系统时, 系统的平均完成数不会随着线程数增多而增多. 当系统中访问用户线程不变的情况下, 系统的可执行并发操作数随着服务器的增多而增加. 即在用户线程一定的情况下, 特定数目的服务器可以使系统达到高效.

其次测试元数据服务器数量变化对用户操作延迟的影响, 测试结果如图 5 所示.

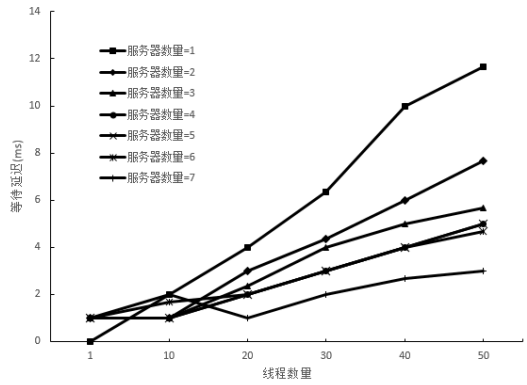

图 5 多 NameNode 用户操作执行延迟

由图 5 中可以分析得到, 本论文提出的系统在多服务器的情况下, 可以有效降低用户操作延迟, 使用户有更好的操作体验.

最后测试在增删元数据服务器的情况下, 元数据服务器负载的变化, 测试结果如图 6 所示.

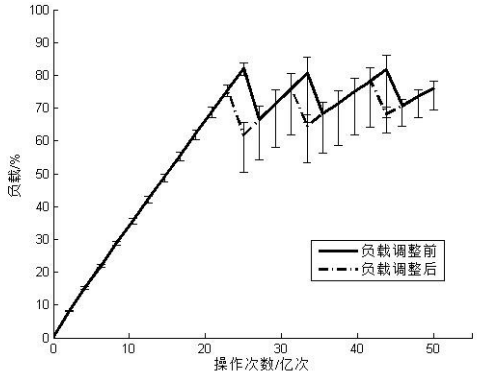

图 6 多 NameNode 负载变化图

由图 6 中可以看出, 系统可以很好的对用户的请求进行均衡的分配, 有效的利用各个 NameNode 节点的资源. 当系统负载高于阈值时, 系统通过添加服务器, 降低负载, 降低用户的系统操作延迟, 降低单一 NameNode 节点的最大内存开销, 保证在服务器上的其他程序正常可用.

4.2 实验结论

通过图 4-6 的实验结果, 我们可以得出论文中提出的系统, 可以很好的对用户请求进行负载均衡, 根据参数的设定, 合理的维护服务器数目, 提高系统的整体资源利用效率, 降低用户的访问与请求延迟.

5 结语

本文提出了一种基于一致性 hash 与目录树的元数据管理系统. 通过对现有 NameNode 的瓶颈分析, 及社区提出的方案分析比较, 给出了基于 NameNode Table(NNT)的解决方案. 本文中给出了负载均衡的计算函数, NameNode 的选择方法, 元数据的迁移策略及服务器的增删策略. 通过以上策略, 可以使系统中的元数据达到负载均衡的同时, 有效控制系统负载, 降低用户操作延迟, 提高系统资源利用率. 在下一步的研究中, 需要对目录重命名导致的元数据移动进行更加深入的研究, 降低该项操作会引起的资源消耗.

参考文献

1 McKusick MK, Quinlan S. GFS: Evolution on Fast- forward. ACM Queue, 2009, 7(7): 10.

2 Weil SA, Brandt SA, Miller EL, et al. Ceph: A scalable, high-performance distributed file system. Proc. of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association, 2006: 307–320.

3 Radia S, Srinivas S. Scaling HDFS Cluster Using NameNode Federation. HDFS-1052, August, 2010.

4 Lustre: http://www.lustre.org

5 Shvachko KV. The Hadoop Distributed File System Requirements. Hadoop Wiki,June 2006://http://wiki.apache.org/hadoop/DFS_requirements.

6 Shvachko KV. HDFS Scalability: The limits to growth. Login, 2010, 35(2): 6–16.

7 Huang Q, Cheng Y, Du R, et al. Design of Scalable Distributed Metadata Management System. Computer Engineering, 2015, 41(5):26–32, 37.

8 Xu Q, Arumugam RV, Yong KL, et al. Efficient and Scalable Metadata Management in EB–Scale File Systems. IEEE Trans. on Parallel & Distributed Systems, 2014, 25(11): 2840–2850.

9 Gupta A, Gundety R, Fernando V, et al. Survey on Metadata Management Schemes in HDFS. International Journal of Computer Science & Information Technolo, 2014.

10 Xiong AP, Ma JY. HDFS distributed metadata management research. International Conference on Applied Science and Engineering Innovation, 2015.