面向大数据分布式存储的动态负载均衡算法

张栗粽 崔 园 罗光春 陈爱国 卢国明 王晓雪
(电子科技大学计算机科学与工程学院 成都 611731)

摘 要 随着大数据时代的到来,分布式存储技术应运而生。目前主流大数据技术Hadoop的HDFS分布式存储系统的元数据存储架构上一直存在可扩展性差和写延迟高等问题,其在官方2.0版本中针对可扩展性的解决方案(Federation)仍不完美,仅解决了原有HDFS扩展性的问题,在元数据分配的问题上没有考虑NameNode的异构性能差异,也未解决NameNode集群动态负载均衡的问题。针对该情况,提出了一种动态负载均衡的分布NameNode算法,通过元数据多副本异构节点的动态适应性备份,使元数据在考虑节点性能及负载的情况下实现了动态分布,保证了元数据服务器集群的性能;同时结合缓存策略及自动恢复机制,提高了元数据的读写性及可用性。该算法在试验验证中达到了较为理想的效果。

关键词 大数据,分布式存储,元数据管理,HDFS
中图法分类号 TP274 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.05.032

Dynamic Load Balance Algorithm for Big-data Distributed Storage

ZHANG Li-zong CUI Yuan LUO Guang-chun CHEN Ai-guo LU Guo-ming WANG Xiao-xue
(School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China)

Abstract Distributed storage is the major approach for handling the "Big Data". Currently, the major technology is hadoop distributed file system (HDFS), which has been beset by the issues of scalability and write latency. In official 2.0 version, a new feature 'HDFS Federation' addresses this limitation by adding support for multiple NameNodes/name spaces to HDFS. However, it does not take the isomerism of NameNode into account, and still lacks of dynamic load balance ability. Consequently, a dynamic load balance algorithm for HDFS NameNode was proposed, and it dynamically allocated the metadata into a NameNodes cluster with multiple copies, in order to improve the performance of metadata utilizations. In addition, the proposed algorithm increases the readability by the adoption of metadata caches, and improves the stability by a built-in failover mechanism. Finally, an experiment was carried out, to illustrate and evaluate the utilizations of the proposed algorithm.

Keywords Big data, Distributed file storage, Meta data management, Hadoop distributed file system (HDFS)

1 引言

随着大数据时代的到来[1-2],传统技术[3-4]已经无法满足日益增长的存储需求,分布式存储技术应运而生。HDFS[5]是Apache[6]开发的大数据平台Hadoop的分布式存储系统[7],具有较高的容错能力,针对超大数据集的处理进行了优化,具有较高的数据吞吐量,是Google文件系统GFS[8]的开源实现[9]。得益于软件层面的高容错性设计,HDFS对硬件要求不高,可以运行在一般的廉价的服务器上。

元数据在HDFS中负责记录文件的逻辑地址与物理地址的映射关系,其包含文件访问控制所需要的信息。对文件进行访问时,首先需要向元数据服务器发起请求,得到该文件对应的元数据后,才能对文件进行定位,并进行后续的文件读写等操作。

为了提高系统性能,HDFS使用了元数据和文件数据相分离的模式。元数据存放在元数据服务器上,而文件数据存放在文件数据服务器上。

HDFS采用主从式架构,由控制节点NameNode(元数据服务器)和数据节点DataNode(数据服务器)组成,它们分别起着存储元数据和数据的功能。NameNode管理着整个文件系统的命名空间,它将文件系统的元数据保存在内存中,但实际的数据存放在DataNode中。

在Hadoop版本1.0中,每一个集群只有一个NameNode,其失效会导致集群故障,同时也存在节点扩展性问

到稿日期:2016-12-30 返修日期:2017-02-23 本文受四川省科技厅应用基础(2015JY0228),科技支撑计划(2015SZ0045,2014GZ0174),电子科大基础研究(ZYGX2015J063),海外留学回国人员科研启动费项目基金资助。

张栗粽(1981-),男,博士,讲师,主要研究方向为大数据、知识工程,E-mail:l.zhang@uestc.edu.cn;崔 园(1990-),男,硕士,主要研究方向为大数据;罗光春(1974-),男,教授,博士生导师,主要研究方向为机器学习、大数据;陈爱国(1981-),男,博士,副教授,主要研究方向为云计算、大数据;卢国明(1990-),男,硕士,主要研究方向为云计算、大数据;王晓雪(1993-),女,硕士,主要研究方向为大数据。

题[10-11]。因此，在 Hadoop 2.0 官方版本中，分别引入了 HDFS High Availability(HA)和 HDFS Federation 两种解决方案。HA 的主要目标是解决单点失效问题，而 Federation 主要解决 Na-meNode节点的扩展性问题。但 HDFS Federation 策略存在无动态负载均衡的缺陷。

2　相关研究

2.1　HDFS Federation 策略

HDFS Federation[12]是为解决 HDFS NameNode 扩展性问题，在 HDFS 2.0 中提出的 NameNode 水平扩展方案。该方案允许 HDFS 创建多个 NameNode 以提高集群的扩展性。HDFS Federation 有两个优点：1)设计和实现简单，NameNode 本身的改变比较小，这样对 NameNode 原来的鲁棒性影响比较小；2)具有良好的向后兼容性。

但 HDFS Federation 并没有解决负载均衡问题，它采用了客户端挂载表的形式分担文件和负载，其每一个 NN 只管理其 NameSpace 下的文件。因此，当某一个文件请求特别高时，其管理 NN 的负载相应升高，此时，该集群中的其余 NN 并不能帮其分担。如何挂载命名空间是由人工进行配置的，难以达到理想的负载均衡状态。

2.2　NCUC 策略

文献[13]提出了另一种对元数据管理的方案 NCUC (NameNode Clustered Using Chord)。其通过在分布式 NameNode集群中使用 Chord[14]协议来提供一种快速的一致性哈希计算。在一个拥有 N 个 NameNode 节点的 NCUC 集群中，每个 NameNode 记录一个拥有项关于其他 NameNode 的路由表，得益于 chord 协议，每次查表仅需要 O(logN)的复杂度。

NCUC 策略可以将元数据自动分配到 NameNode 节点中，而无需人工参与，同时，chord 协议的使用使 NameNode 离开或加入集群时的影响比较小，且查找目标节点时的速度较快。

但 NCUC 策略也有如下缺点：

1)没有动态负载均衡。虽然在 NameNode 集群足够大的情况下，NCUC 在同构 NameNode 集群中能够将元数据均匀地分布到 NameNode 中，但是客户有可能在一段时间内突然对一些文件发出大量的请求，这时需要通过动态负载均衡来提高 NameNode 集群的性能，但 NCUC 策略并没有这种能力。

2)元数据分配没有考虑服务器的异构差异。在生产环境中服务器集群经常扩容，导致服务器之间存在着性能的异构差异。

3　一种动态负载均衡的分布 NameNode 算法

HDFS Federation 和 NCUC 解决了原有 HDFS 扩展性的问题，但是在元数据分配的问题上都没有考虑到 NameNode 的异构性能差异，且都没有解决 NameNode 集群动态负载均衡的问题；同时元数据都采用单副本的形式进行存放，仍然有单点失效的问题存在。由此本文提出一种动态负载均衡的分布式 NameNode 算法。

3.1　元数据服务器集群架构

为了克服 HDFS 中 NameNode 的缺点[15-17]，本文对经典的 HDFS 的设计进行了改进，设计了一套分布式 NameNode 的系统模型。其系统的架构如图 1 所示，由 4 个主要的角色构成：Client(客户端)、NameNodeAdmin(元数据服务器集群管理模块)、NameNode 集群以及 DataNode 集群。在 NameNodeAdmin里面会存储元数据到 NameNode 地址的映射表。在 NameNodeAdmin 之外还有一个 Secondary NameNodeAdmin 保证了 NameNodeAdmin 的高可用，一旦 NameNodeAdmin 出现故障，Secondary NameNodeAdmin 可以代替它对外提供服务。

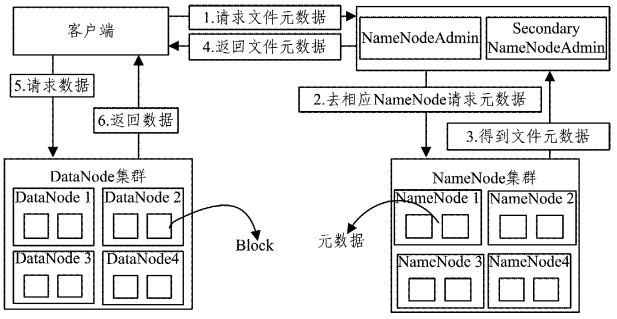

图 1　分布式 NameNode 架构下文件请求流程

SecondaryName NodeAdmin(SNNA)与 NameNodeAdmin(NNA)的工作方式参考了 HDFS2.0 中的 HA With QJM 方案并加以简化，这是考虑到 NNA 所管理的数据为“元数据的元数据”，与 NN 中管理的数据相比，其数据量较小，且变动更少。本策略中，NNA 与 SNNA 是主备关系，通过 Hot Standby 机制，NNA 在进行任何状态修改操作时，会将事件同步到 SNNA 节点，并且 NameNode 的心跳信息也会同时发往 SNNA，但 SNNA 不会发出任何控制指令，只是与 NNA 节点简单保持一致状态，当 NNA 宕机无法做出响应时，请求才会被发往 SNNA 节点，此时 SNNA 代替 NNA 节点开始工作。

本方案采用多个元数据服务器组成元数据服务器集群，提高了存储系统的数据存储能力，但带来了元数据管理问题，在此提出了元数据集群中元数据管理的目标：1)元数据的分布必须考虑元数据服务器的性能差异；2)应该有动态的负载均衡策略；3)应该有客户端缓存机制；4)应该有自动的元数据恢复机制。

3.2　元数据动态分布

本文结合基于表的映射、基于哈希的映射和动态子树分区，提出了一种新的元数据分布算法。访问元数据时，首先在全局映射表中寻找，找到即进行元数据操作；否则根据文件目录进行哈希运算，得到元数据服务器地址，将这个映射关系存放到全局映射表中。本文提出的哈希映射充分考虑元数据服务器集群的负载情况，根据负载对元数据位置进行分配。

哈希函数将元数据映射到环上的某个点。以文件的目录的路径全名为参数，这样同一个目录下的所有文件将会得到相同的值，利用局部性原理可以提高存储系统的性能。

$$f_1 : dir_i \rightarrow [0,1] \tag{1}$$

哈希函数 f_2 将元数据服务器 m_i 映射到[0,1]环上的某个点。f_2 以元数据服务器的 IP 地址和元数据服务对外端口串 $addr_i$ 为参数。

$$f_2 : addr_i \rightarrow [0,1] \tag{2}$$

哈希函数 f_1 和 f_2 可以取为 SHA1 函数[18]。

按照距离函数(3)求得距离 d_i 最近的元数据服务器 m_j，该函数不仅考虑了距离，还考虑了元数据服务器的处理能力，因为元数据服务器的处理能力越强越适合存储更多的元数据，如此才使得负载更均衡，性能更高。因此在距离函数中还加入了元数据服务器处理的能力 $c_i(t)$，并加权计算各主机的处理能力。确定最近的元数据服务器 m_j 后，将 d_i 存放在 m_j 上，同时在全局映射表中记录该对应关系。

$$dis(d_i+m_i)=\frac{-(\ln(1-|f_1(dir_i)-f_2(addr_i)|))}{c_i(t)} \tag{3}$$

令 d_i 到 m_j 的映射过程为 f_0：

$$f_0(d_i)=m_j \tag{4}$$

在元数据服务器 m_i 没有存放任何元数据时，$c_i(0)$的计算方式如式(5)所示：

$$c_i(0)=r1*cpu_i+r2*mem_i+r3*io_i+r4*disk_i \tag{5}$$

其中 $r1,r2,r3,r4$ 分别是 CPU 计算性能、内存性能、IO 性能、磁盘大小所占的权重，它们之间的关系如式(6)所示：

$$r1+r2+r3+r4=1 \tag{6}$$

因为元数据存储是 IO 密集型，对 CPU 资源的消耗很低，不会出现 CPU 利用率超高而内存使用率过低的情况，所以 $r2,r3$ 和 $r4$ 比 $r1$ 更重要。4 个参数可以根据 TOPSIS 算法[19]来进行计算取得最优值，TOPSIS 算法专门用来解决多参数决策的问题，输入 4 个参数的重要程度对比表，通过 TOPSIS 算法计算即可得出 4 个参数的最优取值，这方面的研究较多，因此不再赘述，本文取 $r1=0.116,r2=0.368,r3=0.258,r4=0.258$。

元数据服务器集群对外提供元数据管理服务，伴随着时间的变化，元数据服务器的负载发生了变化，当新的元数据进入到集群中存储时必须考虑该动态变化，式(3)距离函数应该采用增量动态变化来适应此改变。$c_i(t)$随着时间变化的表示如式(7)所示。

$$c_i(t)=r1*cpu_i(t)+r2*mem_i(t)+r3*io_i(t)+r4*disk_i(t) \tag{7}$$

算法实现过程如下：

1)得到将要存放的元数据 d_i 的目录路径名 dir_i，根据式(1)计算 $f_1(dir_i)$，将结果映射到元数据[0,1]环上的某个点。

2)得到每一个元数据服务器 m_i 的 IP 地址和端口号，组合成串 $addr_i$，根据式(2)计算 $f_2(addr_i)$，将结果映射到元数据服务器[0,1]环上的某个点。

3)从所有元数据服务器上收集当前元数据服务器的 CPU 处理能力、剩余内存大小、IO 带宽剩余大小、磁盘剩余容量，根据式(7)计算元数据服务器当前的处理能力。

4)将元数据[0,1]环和元数据服务器[0,1]环重叠在一起，根据式(3)计算出元数据和所有元数据服务器之间的相对距离集合 D。

5)在集合 D 中选择最小的值，相对应的元数据服务器就是元数据要存放的地点。将元数据与存储位置的映射添加到全局映射表中。

算法伪代码如算法 1 所示。

算法 1 元数据动态分布算法

```
Input:
    M={m1,m2,…,mn}//元数据服务器集群节点序列
    di//要存储的元数据
    r1,r2,r3,r4//CPU、内存、网络带宽、磁盘的权重
    cpui(t),memi(t),ioi(t),diski(t)//元数据服务器 mi 当前的处理能力参数
Output:
    mj//元数据最终要存储的元数据服务器
Procedure:
    A=f1(diri)
    for(i=0;i<n;i++){//计算元数据服务器和要存储的元数据在哈希环上的距离
    ci(t)=r1*cpui(t)+r2*memi(t)+r3*ioi(t)+r4*diski(t)Bi
          =f2(addri)
    disti=(-(ln(1-|A-Bi|)))/ci(t) }
    j=min(dist) //选择哈希换上距离要存储的元数据最近的元数据服务器
return mj
```

3.3 元数据动态负载均衡

存储系统中会出现热点文件进而出现热点元数据，导致元数据服务器集群负载不均衡，通过修改存储位置无法解决此问题，基于此提出了一种动态的元数据负载迁移策略。

考虑元数据服务器的负载应从元数据服务器的资源利用率的角度入手。定义元数据服务器 m_i 的资源利用率 $\alpha_i(t)$如式(8)所示：

$$\alpha_i(t)=r1*cpuU_i(t)+r2*memU_i(t)+r3*ioU_i(t)+r4*diskU_i(t) \tag{8}$$

为了避免某个时刻元数据服务器的资源利用率出现突变的情况，使用一个基于权重的移动平均计算修正后的元数据服务器的资源利用率，其中参数 $0<\theta<1$。

$$\alpha_i'(t)=\theta*a_i(t)+(1-\theta)*a_i'(t-1) \tag{9}$$

考虑了元数据服务器的资源利用率，同时也应该综合考虑元数据服务器现在的服务质量，通过元数据请求在元数据服务器内的平均逗留时间可以很好地判断出现在元数据服务器的服务质量，如果元数据请求的平均逗留时间短，则说明元数据服务器的服务质量比较高，负载较低。

元数据服务器 m_i 在 t 时刻最近一段时间内处理的元数据请求集合为 $Q_i=\{q_1,q_2,\cdots,q_y\}$，其在系统内的平均逗留时间 $\beta_i(t)$如式(10)所示：

$$\beta_i(t)=\frac{1}{y}\sum_{q_j\in Q_i}(w_j+s_j) \tag{10}$$

定义元数据服务器的负载 $load_i(t)$如式(11)所示：

$$load_i(t)=\alpha_i'(t)+\beta_i(t) \tag{11}$$

定义元数据服务器集群 $M=\{m_1,m_2,\cdots,m_n\}$的平均负载 $load(t)$如式(12)所示：

$$load(t)=\frac{1}{n}\sum_{m_i\in M}load_i(t) \tag{12}$$

定义元数据服务器集群的负载阈值高位 $load_h(t)$如式(13)所示，放大参数 $\chi>1$。

$$load_h(t)=\chi\times load(t) \tag{13}$$

定义元数据服务器集群的负载阈值低位 $load_l(t)$如式(14)所示，缩小参数 $0<\tau<1$。

$$load_l(t)=\tau\times load(t) \tag{14}$$

在元数据服务器集群中若某个元数据服务器 m_i 的负载 $load_i(t) > load_h(t)$，则说明该元数据服务器处于高负载状态，反之亦然。动态负载迁移算法会定期检查集群中所有元数据的负载情况，将负载较高的元数据服务器上的数据向负载较低的服务器迁移。

算法对元数据集合中的访问频率进行降序排列，排在最前面的 20% 的元素对应的元数据被认定为局部热点元数据，另外，集群中所有元数据中访问频率在前 20% 的也被认定为全局热点元数据。

本文的动态负载均衡策略分为主动的负载复制和被动的负载迁移。如果低负载的元数据服务器上存在着全局热点元数据，那么该元数据服务器会主动发出负载复制的请求，全局热点元数据将会被复制到别的空闲元数据服务器上分担负载压力。但是如果元数据服务器的负载超过集群的平均阈值，成为热点元数据服务器，则需要将热点元数据服务器上的局部热点元数据移动到空闲服务器上。

算法的实现过程如下：

1）收集所有元数据服务器当前的 CPU 利用率、内存使用率、IO 使用率、磁盘使用率，根据式(8)计算元数据服务器的资源利用率 $\alpha_i(t)$。

2）使用式(9)修正服务器的资源利用率得到 $\alpha_i'(t)$。

3）从每一个元数据服务器上收集其过去一段时间处理的元数据请求信息，得到元数据请求的平均逗留时间 $\beta_i(t)$。

4）使用式(11)计算所有元数据服务器的当前负载，得到序列 $load=\{load_1, load_2, \cdots, load_n\}$。

5）使用式(12)计算元数据服务器集群的平均负载，使用式(13)计算出集群的负载迁移阈值 $load_{th}(t)$。

6）找到高负载元数据服务器中的局部热点元数据和低负载元数据服务器中的全局热点元数据。

7）对低负载元数据服务器上的全局热点元数据执行主动负载复制。

8）对高负载元数据服务器上的局部热点元数据执行被动负载迁移。

算法伪代码如算法 2 所示。

算法 2　元数据负载动态均衡算法

```
Input:
    M={m1,m2,…,mn}//元数据服务器集群节点序列
    r1,r2,r3,r4//CPU、内存、网络带宽、磁盘的权重
    cpuUi(t),memUi(t),ioUi(t),diskUi((t)//元数据服务器 mi 当前资源利用率参数
    Qi={q1,q2,…,qy}//元数据服务器 mi 过去一段时间内处理的元数据请求
    wj,sj//元数据请求 qj∈Qi 在系统中的等待时间和服务时间
    θ,χ,τ//资源利用率修正参数和负载阈值参数
Output:
    从负载超过阈值的元数据服务器中迁移热点元数据到负载轻的元数据服务器
Procedure:
    load_sum=0
    for(i=0;i++;i<n){ //计算总体负载
        αi(t)=r1*cpuUi(t)+r2*memUi(t)+r3*ioUi(t)+r4*diskUi(t)
        αi'(t)=θ*ai(t)+(1-θ)*ai'(t-1)
        stay_time=0
        for(j=0;j<y;j++){
            stay_time=stay_time+wj+sj
        }
        βi(t)=stay_time/y
        loadi=αi'(t)+βi(t)
        load_sum=load_sum+loadi
    }
    load(t)=load_sum/n
    loadh(t)=χ×load(t)//计算高负载阈值
    loadl(t)=τ×load(t)//计算低负载阈值
    OverHeadMdsSet=∅
    LowHeadMdsSet=∅
    for(i=0;i<n;i++){
        if(loadi(t)>loadh(t))
            OverHeadMdsSet.add(mi)//标记高负载元数据服务器
        if(loadi(t)<loadl(t))
            LowHeadMdsSet.add(mi)//标记低负载元数据服务器
    }
    foreach lowHeadMds:LowHeadMdsSet{ //低负载服务器上的全局热点元数据复制
        if lowHeadMds.hasGlobalHotMetaData(){
            mto=LowHeadMdsSet.pickWithout(lowHeadMds)
            MetaData.copy(GlobalHotMetaData,lowHeadMds,mto)
        }
    }
    //高负载服务器上的局部热点元数据迁移
    while (OverHeadMdsSet≠∅&&LowHeadMdsSet≠∅){
        mfrom=OverHeadMdsSet.pick()
        mto=LowHeadMdsSet.pick()
        D=mfrom.getHotMetaData()
        MetaData.move(D,mfrom,mto)
        OverHeadMdsSet.delete(mfrom)
    }
```

3.4　元数据客户端缓存

元数据管理的目标从客户端的角度来看应该是缩小用户元数据查询的时间。本小节提出一种客户端缓存机制来提高客户端查询元数据的效率。

本文提出的元数据管理策略中由一个全局的元数据存储映射表 DLT 和每一个元数据服务器自己的 SDLT 组成。为 SDLT 设置一个版本号 $Version_{SDLT}$，每当 SDLT 更新以后，$Version_{SDLT}$ 便递增一次。根据式(4)可知，同一个目录下的所有文件的元数据都保存在同一个元数据服务器上。根据局部性原理，如果访问了同一个目录下的一个文件，那么该目录下的其他文件有很大的可能被访问到。可以在客户端中设置一个环形缓冲区来存储 SDLT，每次访问元数据服务器获得元数据信息时，同时也得到 $Version_{SDLT}$。将得到的版本号与缓存中的 SDLT 的版本号进行对比，若前者的版本号比较小，就更新到最新版本的 SDLT。将所有的 SDLT 按照访问的先后顺序存储在环形缓冲区中，若存储区满则覆盖环开始的地方。

为了在客户端缓冲区中保存更多的信息，考虑对 SDLT 进行压缩编码，同时减少对网络 IO 的压力。本文考虑采用布隆过滤器来对 SDLT 进行压缩编码产生一段摘要。采用布隆

过滤器不仅可以压缩 SDLT 的空间，而且可以提高查表的速度。

3.5 元数据自动恢复

因为元数据服务器在本系统中起着至关重要的作用，若有一个元数据服务器宕机，则必须立即进行恢复；否则，当所有元数据的副本全部失效，系统会处于不可用状态，将无法为外部继续提供任何服务。

优先级任务队列用于按照数据恢复的优先级存储当前系统所有待处理的恢复任务。将元数据的存活副本数作为恢复任务的权重，权重越小任务的优先级越高。

在实现中，优先级任务队列由一组子队列组成：每个子队列存储了特定存活副本数的待恢复元数据列表。挑选任务时优先从存活副本数量少的队列中选择任务。同时在每个子队列内部维护了两种状态的任务列表：TAST_UNINITED 和 TASK_INITED，前者代表尚未初始化的任务（任务的恢复源和目的尚未选择），后者代表已经初始化的任务。子队列内部 TASK_INITED 队列中任务的优先级高于 TASK_UNINITED 队列中的任务。NameNodeAdmin 将从优先级任务队列中获取当前优先级最高的恢复任务来执行。元数据自动恢复的执行步骤如表 1 所列。

表 1 元数据自动恢复的执行步骤

1. 检查待恢复元数据副本是否足够，若是则无须恢复
2. 判断任务状态，如果任务状态为 TASK_INITED，则转步骤 4
3. 选择恢复源，恢复源选择为该元数据任意可用副本。选择恢复目的，恢复目的选择为元数据服务器集群内最近一段时间内 I/O 压力较小的元数据服务器。更新任务状态为 TASK_INITED，更新元数据副本位置分布信息
4. 计算恢复限速，如果恢复限速<=0，则意味着当前无法执行恢复，转步骤 8
5. 向恢复源发送数据恢复任务
6. 修改任务状态为 TASK_PROGRESSING
7. 等待任务汇报状态
8. 从任务队列中删除任务

4 实验及测试结果

为了验证分布式 NameNode 算法，实现了原型系统，并搭建了一个集群实验环境，进行实验测试并且和现有的分布式 NameNode 策略 HDFS Federation，NCUC 进行对比。主要验证以下几个问题：

1）元数据的分布是否考虑到 NameNode 性能的差异。

2）NameNode 集群是否能够进行动态负载均衡。

3）NameNode 集群增加或者退出节点时对系统造成的影响。

4）元数据查找时间与其他策略的对比。

4.1 实验环境

本系统的开发和测试均部署在 Linux 环境下，操作系统使用 Ubuntu 14.04LTS，使用的 Hadoop 版本为 Hadoop-2.3.0，开发环境使用 Eclipse 4.3.2；网络环境为 100Mbps 局域网。

4.2 实验结果分析

4.2.1 元数据分布分析

在元数据分布上，本文对提出的算法、HDFS Federation 和 NCUC 算法进行了对比实验。实验所用的文件在各目录命名空间中均匀分布。实验结果如图 2 所示，在 NameNode 集群中，NameNode 1 的处理性能最强，NameNode 2 次之，NameNode 3 最弱。实验结果显示，本文提出的算法充分考虑了元数据服务器的性能，处理能力高的元数据服务器（NameNode1）中存储的元数据多。

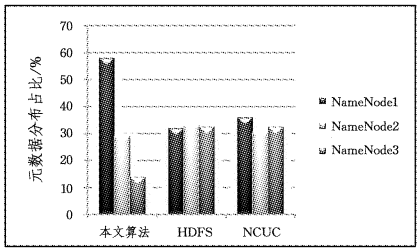

图 2 元数据分布实验结果

4.2.2 动态负载均衡分析

实验中，客户端开始不断向 NameNode 3 中存放的元数据对应的文件发起请求，NameNode 3 的负载会变大，而其他元数据服务器的负载较低，此时 NameNode 集群中的负载出现不均衡的状况，大量请求在 NameNode 3 中排队等候，造成系统性能低下。

如图 3 所示，本文算法会执行动态负载均衡，将热点元数据迁移到负载较低的 NameNode 1 中，NameNode 1 分担了 NameNode 3 的一部分负载，NameNode 3 的负载降低，NameNode 1 的负载升高，最终集群达到了一个负载均衡的状态。NCUC 策略和 HDFS Federation 均没有动态负载均衡的功能，各 NameNode 的负载并没有变化。

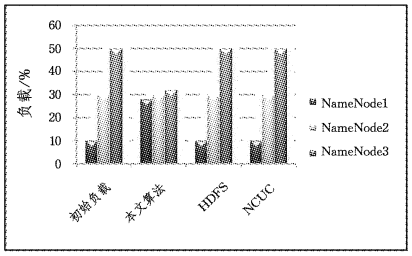

图 3 元数据动态负载均衡实验结果

4.2.3 集群变化影响分析

首先考虑集群中加入新的 NameNode 节点的情况：NCUC 的哈希环产生了变化，在哈希函数均匀的情况下，最多需要迁移 $1/N$ 的元数据（N 为 NameNode 节点的个数）；HDFS Federation 因为每个 NameNode 管理独立的命名空间，当新加入 NameNode 节点时，元数据不需要迁移；本算法元数据分布根据哈希来定位 NameNode 的位置，但是会将位置映射记录下来，因此也不需要迁移元数据，但是根据本算法的元数据动态分布算法，后续新的元数据会优先存放在新加入的 NameNode 节点中。实验结果如表 2 所列。

表 2 NameNode 加入或离开集群的影响

	加入	离开
HDFS Federation	无需迁移元数据	会出现单点故障
NCUC	迁移至多 $1/N$ 的元数据	会出现单点故障
本文算法	无需迁移元数据	元数据自动恢复

当 NameNode 节点离开集群时，在 NCUC 和 HDFS Federation中元数据仍然是单副本存储，所以 NameNode 仍极易发生单点故障问题。本策略中元数据采用三副本的方式进行存放，大大提高了系统的可用性，如果一个 NameNode 离开集

群，则会启动元数据自动恢复，将副本数不足 3 份的元数据恢复到其他可用的 NameNode 节点上。

4.2.4　元数据查找时间分析

元数据查找时间定义为从用户发起读请求到得到元数据的时间。发起文件读请求，分别记录 3 种分布式 NameNode 策略的元数据查找时间，实验结果如表 3 所列。

表 3　随机元数据查找时间/ms

	第一次	第二次	第三次	第四次	第五次	平均
HDFS Federation	62.90	63.20	64.30	63.20	61.70	63.05
NCUC	50.10	49.80	55.30	52.40	51.70	51.86
本文算法（顺序文件）	64.30	41.20	42.20	42.90	43.80	46.88
本文算法（随机访问）	63.20	64.80	63.70	62.80	63.60	63.62

实验方法为对同一个命名空间目录下的文件顺序访问时，由于本文算法采用了基于布隆过滤器的客户端缓存，元数据缓存命中的情况下，元数据的查找时间优于 NCUC 策略和 HDFS Federation。实验方法为随机访问文件时，本文的缓存策略无效，等价于不使用缓存的情况，其结果如表 3 最后一行记录所列，本文算法采用了元数据动态分布与负载均衡策略，元数据查找时间和 HDFS Federation 相当，NCUC 使用 chord 协议来定位元数据，通过在每个NameNode上使用一个局部路由表，加速了元数据的查找，因此元数据查找速度最快。

实际环境中，顺序访问文件的可能性很高，因此命中缓存的概率也很高。本文算法虽然在随机访问文件时的速度与 HDFS Federation 及 NCUC 相当，但此结果是在考虑了动态负载缺陷和元数据服务器的性能异构的情况下得出的。因此，综合命中缓存的情况，本文算法是优于 HDFS Federation 及 NCUC 的。

4.3　实验结论

通过实验可以看出，本文提出的分布式 NameNode 算法在元数据分布时考虑到了 NameNode 的性能差异，并且拥有动态负载迁移能力，在 NameNode 离开集群时，无需人工介入即能够自动启动元数据恢复，在 NameNode 加入集群时无需元数据迁移，在顺序访问文件的情况下，元数据读取表现较好。

结束语　本文首先对 HDFS2.0 算法进行了分析，但官方针对可扩展性的解决方案(Federation)仍存在缺陷，其无法解决元数据的负载均衡问题。因此本文提出了一种动态负载均衡的分布 NameNode 算法，使得元数据在元数据服务器中均匀分布，并通过动态的负载均衡策略来保证元数据服务器集群的性能，通过一种使用布隆过滤器的客户端缓存策略提升存储系统的读写性能，通过元数据自动恢复机制保障元数据的可用性，最后通过实验得到验证，达到了较为理想的效果。

目前本算法在实验室环境下测试表现良好，在接下来的研究工作中，将会在真实环境下对算法进行测试，并进行进一步改进，以提高算法的稳健性。

参 考 文 献

[1] GANTZ J, REINSEL D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. IDC iView: IDC Analyze the future [R/OL]. https://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf.

[2] GANTZ J F. The Diverse and Exploding Digital Universe. An Idc White Paper Retrieved [R/OL]. https://italy.emc.com/collateral/analyst-reports/emc-digital-universe-china-brief.pdf.

[3] TATE J, LUCCHESE F, Moore R, et al. Introduction to Storage Area Networks[M]. Vervante, 2006.

[4] GIBSON G A, VAN METER R. Network attached storage architecture[J]. Communications of the Acm, 2000, 43(11): 37-45.

[5] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]//2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2010: 1-10.

[6] WHITE T. Hadoop: The Definitive Guide[M]. Yahoo! Press, 2011.

[7] ZHANG X. Research and Implementation of Cloud Storage Platform Based on Hadoop[D]. Chengdu: University of Electronics and Technology of China, 2013. (in Chinese)
张兴. 基于 Hadoop 的云存储平台的研究与实现[D]. 成都：电子科技大学，2013.

[8] GHEMAWAT S, GOBIOFF H, LEUNG S T. The Google file system[J]. Acm Sigops Operating Systems Review, 2003, 37(5): 29-43.

[9] BORTHAKUR D. HDFS architecture guide[EB/OL]. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf.

[10] SASHI K, THANAMANI A S. Dynamic replication in a data grid using a Modified BHR Region Based Algorithm[J]. Future Generation Computer Systems, 2011, 27(2): 202-210.

[11] TATEBE O, HIRAGE K, SODA N. Gfarm Grid File System [J]. New Generation Computing, 2010, 28(3): 257-275.

[12] Hadoop Apache Project, HDFS Federation [OL]. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html.

[13] AZZEDIN F. Towards a scalable HDFS architecture[C]//2013 International Conference on Collaboration Technologies and Systems (CTS). IEEE, 2013: 155-161.

[14] STOICA I, MORRIS R, KARGER D, et al. Chord: A scalable peer-to-peer lookup service for internet applications [J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149-160.

[15] BREWER E A. Towards robust distributed systems (abstract) [C]//Nineteenth ACM Symposium on Principles of Distributed Computing. ACM, 2000: 7.

[16] GRAY J. The transaction concept: virtues and limitations (invited paper)[C]//International Conference on Very Large Data Bases. VLDB Endowment, 1981: 144-154.

[17] GRAY J, REUTER A. Transaction Processing: Concepts and Techniques[M]. Morgan Kaufmann Publishers Inc., 1992.

[18] EASTLAKE R D, JONES P. US Secure Hash Algorithm 1 (SHA1)[M]. RFC Editor, 2001.

[19] TZENG G H, HUANG J J. Multiple Attribute Decision Making: Methods and Applications[J]. Lecture Notes in Economics & Mathematical Systems, 2011, 375(4): 1-531.