

一种面向低延迟的内存 HDFS 数据存储策略

英昌甜, 于 炯, 廖 彬, 鲁 亮

(新疆大学 信息科学与工程学院, 新疆 乌鲁木齐 830046)

摘 要: 从底层存储介质的选择和上层分布式文件系统效率的角度出发, 提出基于 HDFS 的内存分布式文件系统架构 Mem-HDFS, 同时利用集群 DataNode 的内存和磁盘存储数据, 以提高基于 HDFS 应用的读取性能, 并对其性能优化, 提出一种适应的同分布数据存储策略以及并行读取访问算法。对比实验表明: 提出的算法能够较好地降低读取访问延迟。

关键词: 内存分布式文件系统; 数据存储策略; 低延迟; 并行读取算法

中图分类号: TP393

文献标识码: A

文章编号: 1000-7180(2014)11-0160-07

A Data Storage Strategy for Low-latency in Memory HDFS

YING Chang-tian, YU Jiong, LIAO Bin, LU Liang

(College of Information Science and Engineering, Xinjiang University, Urumqi 830046, China)

Abstract: In this paper, we present a storage system, Mem-HDFS, to enable low-latency read accesses. Mem-HDFS integrates the available memory resources in an HDFS clusters to form a cloud storage system. We also present an adaptive identically distributed storage strategy and concurrently read access algorithm. By serving data from memory instead of disks, Mem-HDFS can yield high I/O performance with low latency.

Key words: Memory HDFS; data storage strategy; low-latency

1 引言

近年来, 作为云存储的核心基础平台, 分布式文件系统已经成为研究的重点。目前, 具有代表性的分布式文件系统有 Google 的 GFS, 开源的 Lustre、HDFS(Hadoop Distributed File System)、MooseFS 等, 这些分布式文件系统均面向大规模的云存储应用^[1]。

构建分布式文件系统需要考虑多方面的因素, 底层存储系统的性价比和上层分布式文件系统的效率是两个最基本和最重要的影响因素。目前, 基于磁盘的存储系统 I/O 性能较差, 磁盘 I/O 性能与处理器的性能差距正变得越来越大, 已经成为分布式文件系统的性能瓶颈^[2]。

基于内存的动态随机访问存储器(Dynamic Random Access Memory, DRAM)分布式文件系统

能够提供高 I/O 吞吐率和低访问延迟, 仅将集群中内存用来存储数据, 磁盘作为备份, 不仅价格昂贵、容量受限、使用不便。

为解决分布式文件系统低延迟读取访问的问题, 学术界与工业界提出了各种解决方案和框架, 例如 Dremel, Spark, Impala 等。

本文在研究现有技术^[2-3]基础上, 提出基于 HDFS 的内存分布式文件系统架构 Mem-HDFS, 提高了仿真性能。

2 Mem-HDFS 架构

Mem-HDFS 基于 HDFS, 使用 HDFS 作为备份。由于 HDFS 具有备份、同步和监控机制, 故系统可靠性和数据可用性可由 HDFS 保证。数据初始写入 HDFS 底层磁盘, 需访问时迅速写入 RAM 中。Mem-HDFS 目标在于加速 HDFS 读取访问速度,

收稿日期: 2014-01-17; 修回日期: 2014-03-09

基金项目: 国家自然科学基金资助项目(61262088, 61063042); 新疆维吾尔自治区自然科学基金资助项目(2011211A011)

而应用数据可根据访问频率而分成频繁访问和较少访问,因此仅在内存中存储频繁访问的文件,从而提高访问速度。

Mem-HDFS 架构如图 1 所示,与 HDFS 类似,在 HDFS 中创建目录作为 Mem-HDFS 目录,并将其他目录放在该根目录下,Mem-HDFS 命名空间作为 HDFS 子空间,可统一管理。Mem-HDFS 与原 HDFS 同样管理 NameNode 和 DataNode。NameNode 包含命名空间,管理磁盘和内存中数据块整体信息。

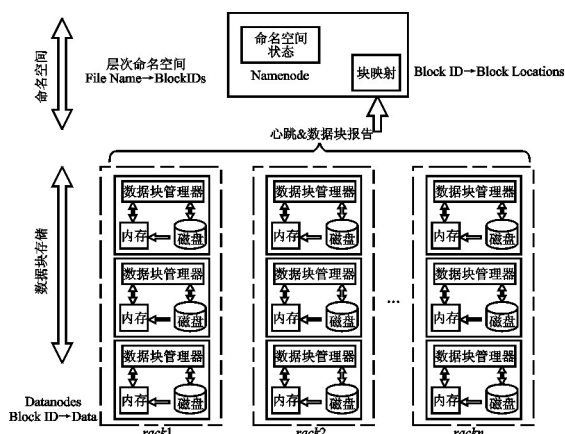


图 1 Mem-HDFS 架构图

当获取文件时,Mem-HDFS 首先判断是否存在于 Mem-HDFS 的命名空间,若文件在其命名空间,则由 Mem-HDFS 处理 I/O 请求,否则 HDFS 会处理 I/O 请求。通过对数据块的 Location Blocks 添加属性以扩展,即是否在内存中,由(Disk, Ram)二元组表示,表明 Mem-HDFS 可选择的工作方式包括三种,(1,0)时表示 Disk_only(仅使用磁盘)、(0,1)时表示 Ram_only(仅使用内存)和(1,1)表示 Disk_RAM(磁盘和内存同时使用)。

当工作方式 Disk_only 时,即为传统的基于磁盘的 HDFS。当工作方式 Ram_only 时,即为基于内存的分布式文件系统,此时磁盘仅作为备份设备。当工作方式 Disk_RAM 时,即为磁盘作为主存储设备,内存用来存储需要频繁访问的数据。本文重点考虑第三种工作方式,即当内存和磁盘同时存储数据的情况。

节点划分出的内存主要起三种作用:第一个作用就是存储频繁访问过的数据;二是预读取;三是对写入动作进行缓存。

Mem-HDFS 在选择访问节点时采取选择与最近的 DataNode 连接策略。由于 Mem-HDFS 将频繁

访问的数据存储在 DataNode 内存,以提高访问效率,因而同一个数据块的 m 个副本存储在不同 DataNode 的内存或磁盘中。当客户端 Client 读取数据时,首先需对 DataNode 排序,DFSClient 通过 DataNode 的排序来获得每个数据块。如果在排序队列中获取 DataNode 某个数据块失败,则会尝试下一个 DataNode。排序策略为:(1)DataNode 离 DFSClient 距离近的优先选择;(2)对所有与 DFSClient 具有相同距离的 DataNode,优先选择内存中存储副本的 DataNode。数据块存储位置的不同,其与 Client 距离也不相同。与 Client 的距离由近到远的排列如下:本地主机内存、本地主机磁盘、本地机架内存、本地机架磁盘、跨机架内存、跨机架磁盘。

Mem-HDFS 进行数据块存储时使用同分布策略,由于机架内访问速率较高,因此将同一文件的不同数据块存储在同一机架,尽可能访问本机架内的数据,因此可以较好地降低访问延迟。同时基于该策略提出三种算法,将在以下部分详细介绍。这三种算法分别如下:

(1)内存访问算法。通过将频繁访问的数据存储在内存中,访问时直接读取内存而不受磁盘的读写效率影响。该算法只能对于部分的数据的访问性能具有较好提升。适用场景为迭代计算和频繁访问的数据。

(2)仅使用文件级并行算法,并行读取文件的多个数据块。

(3)同时使用文件级和数据块级并行算法,对于同一数据块,并行读取多个副本。

3 基于同分布的存储策略

HDFS 中文件都被拆分成多个数据块存放在 DataNode 节点中,并采用在同一个 HDFS 集群中存储多个副本的机架感知策略来提高数据的可靠性。基于机架感知的数据块存储策略具有较大随机性。某文件分为 j 个数据块,由于数据块存储的随机性,当访问该文件时,并不能预测 j 个数据块存储在哪个机架的 DataNode。当访问该文件的数据块时,需要跨越多个机架传输数据,因此增加了网络延迟^[4]。利用同分布策略,将文件 F 的 $b_{11}, b_{21}, \dots, b_{m1}$ 放在同一机架的 n 个 DataNode,其副本存储在相邻机架的 DataNode 中,尽可能访问本机架内的数据,通过数据本地化来降低网络延迟^[4]。

假设 HDFS 集群中有 k 个 DataNode,数据块副本系数为 m ,数据块大小为 bs ,则 HDFS 中某个大小为 $n * bs$ 的文件 F 会有 $n * m$ 数据块随机存储在 k 个

DataNode 中,其数据块可以表示为矩阵 $\mathbf{DB}_{n \times m}$,其中,原文件 F 由数据块 $b_{11}, b_{21}, \dots, b_{n1}$ 组成, $\mathbf{DB}_{n \times (m-1)}$ 表示该文件的副本.

$$\mathbf{DB}_{n \times m} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} \quad (1)$$

$$\mathbf{DB}_{n \times (m-1)} = \begin{bmatrix} b_{12} & \cdots & b_{1m} \\ b_{22} & \cdots & b_{2m} \\ \vdots & \ddots & \vdots \\ b_{n2} & \cdots & b_{nm} \end{bmatrix} \quad (2)$$

将数据块分布矩阵 $\mathbf{D}_{n \times m}$ 按照同分布的数据存储策略的放入 HDFS 集群中,其中 rack1 中存放着文件 F 的 n 个原始数据块,机架 rack2 到 rack m 存放备份数据块,即为矩阵 $\mathbf{DB}_{n \times (m-1)}$ 中表示的数据块. 可将副本系数为 m 的 DataNode 节点结构表示为矩阵 $\mathbf{DN}_{r \times m}$,并将 $\mathbf{DN}_{r \times m}$ 定义为副本系数为 m 的 DataNode 节点组(group),则 $\mathbf{DN}_{r \times m}$ 可表示为

$$\mathbf{DN}_{r \times m} = \begin{bmatrix} dn_{11} & dn_{12} & \cdots & dn_{1m} \\ dn_{21} & dn_{22} & \cdots & dn_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ dn_{r1} & dn_{r2} & \cdots & dn_{rm} \end{bmatrix} \quad (3)$$

基于同分布的数据块存储策略使得 $\mathbf{DN}_{r \times m}$ 矩阵中每一行中的 m 个 DataNode 数据块存储结构完全一致,其中第一列 r 个 DataNode 均存放为原始数据块,2 到 m 列存放备份数据块. 由于副本系数 m 为可变因子,所以一个 HDFS 集群可由多个组构成,其结构可表示为

$$\text{HDFS} = \mathbf{DN}_{r1 \times 1} + \mathbf{DN}_{r2 \times 2} + \dots + \mathbf{DN}_{rm \times m} \quad (4)$$

当用户上传某个副本系数为 3 的文件到 HDFS 集群时,相应地该文件应该存放在 $\mathbf{DN}_{r3 \times 3}$ 组中,若数据块的副本系数 m 发生改变,则放在相应组中.

4 低延迟并行访问算法

4.1 基于内存的访问算法

在访问 DataNode 时,系统的磁盘 I/O 和网络 I/O 往往会成为系统的瓶颈. 需要读取磁盘上的数据包括三种情况:(1)客户端 Client 读数据时,需要从磁盘获取数据;(2)DataNode 间根据名字节点指令进行的数据块复制;(3)数据块扫描器在周期性地验证 DataNode 存储的数据块时,也需要读磁盘上的数据. 因此,针对该问题,利用集群内存存储访问频率高的数据,在客户端读取数据时,可以直接访问内

存而不是磁盘.

文献[5]中通过对 Yahoo 公司 HDFS 集群内部数据块访问日志分析得出,90.26%的数据块都会在其上传 2 天内进行第一次访问,89.61%的数据块都会再其上传后的 10 内进行最后一次访问,40%的数据块最后一次读取时间到最后删除的时间跨度都会超过 20 天. 由此可见 HDFS 集群内部数据块的访问具有较强的规律性,数据块的访问的热点期大多集中在其上传后较短的一段时间内,过后则进入空闲期. 根据这种规律提出算法 1,同分布数据块存储策略下的内存数据块配置算法,通过将频繁访问数据块放入内存以提高响应速度,同时将空闲数据块从内存中删除以节省空间.

另外在文献[6]中,三大门户网站,包括新浪、搜狐、QQ 某天用户访问波动图显示在凌晨 0 点到 7 时段用户访问数较少,此时服务器负载较低. 由于用户请求少,服务器负载低,服务器内存存储数据块替换和配置并不影响用户体验. 大多数系统都存在三大门户类似的负载变化情况,故可将此时段设置为数据块配置算法执行时段,在不影响应用性能与用户 QoS 的前提下,根据集群实际情况进行设置.

算法 1:内存数据块配置算法

- 1 初始化:设置数据块配置算法开始执行时间 t (一般选择集群空闲时),算法执行周期 n ,提升热点数据块最低访问次数 k ,副本系数 m ,主数据块的 ID 为 BlockID,主数据块与其副本访问次数的累加 VisitTime.
- 2 当达到执行时间要求时,读取集群数据块访问日志,提取数据块和其访问次数放入 $\text{LOG}[\text{BlockID}] \rightarrow [\text{VisitTime}]$ 数据结构.
- 3 $\text{LOG}[\text{BlockID}] \rightarrow [\text{VisitTime}]$ 中 BlockID 按照 VisitTime 降序排序.
- 4 排序后,集群所有 BlockID 放入 $\text{ALL_BLOCK_ID}[]$ 数组,日志中访问的所有 BlockID 放入 $\text{VISITED_BLOCK_ID}[]$ 数组,VisitTime 大于系数 k 的主数据块 ID 放入 $\text{HOT_BLOCK_ID}[]$ 数组.
- 5 循环 $\text{HOT_BLOCK_ID}[]$,将其对应数据块添加至相应 $\mathbf{DN}_{m \times m}$ 组的第一个 DataNode 内存中,若无法添加,则存储在 $\mathbf{DN}_{r \times m}$ 组中同行的活动 DataNode 节点内存中,循环至最后一个 DataNode.
- 6 $\text{ALL_BLOCK_ID}[]$ 与 $\text{VISITED_BLOCK_ID}[]$ 差运算,得到在周期 n 内都没有访问过的数据块的 ID 数组 $\text{IDLE_BLOCK_ID}[]$. 循环 $\text{IDLE_BLOCK_ID}[]$,若存储在 DataNode 内存中,将该 Block 从内存中删除.

由于内存数据块配置算法按照数据访问频率,

将常用数据存储在内存在中,而 HDFS 下的读取算法需要根据数据块存储位置的不同,选择从内存或是磁盘读取数据.因此,

算法 2 对原 HDFS 的读取算法进行改进.

- 1 client 调用 *FileSystem* 的 *open()* 函数打开文件.
- 2 *DistributedFileSystem* 用 RPC 调用 *NameNode* 节点,得到文件的数据块信息.对于每一个数据块,*NameNode* 返回保存数据块的 *DataNode* 的地址. *DistributedFileSystem* 返回 *FSDDataInputStream* 给客户端,用来读取数据.
- 3 client 调用 *stream* 的 *read()* 函数开始读取数据, *DFSInputStream* 连接保存此文件第一个数据块的最近的 *DataNode*,即排序后可用节点 *availableDataNodes* [] 的第一个数据节点.根据对应数据块存储位置属性,从内存或磁盘中读取数据块.
- 4 当此数据块读取完毕时, *DFSInputStream* 关闭和此数据节点的连接,然后连接此文件下一个数据块的最近的数据节点.
- 5 当 client 读取完毕数据时,调用 *FSDDataInputStream* 的 *close()* 函数.在读取数据的过程中,若与 *DataNode* 通信出现错误,则尝试连接包含此数据块的下一个 *DataNode*.
- 6 失败的 *DataNode* 将被记录,以后不再连接.

在基于机架感知的数据块存储策略下,原 HDFS 写入算法采取了流式写的机制进行数据块的写入.而在同分布的数据块存储策略下,数据块的写入因为数据块的存储策略的不同而需要做出改变,如算法 3.

算法 3:基于内存的写入算法

- 1 Client 调用 *create()* 来创建文件.
- 2 *DistributedFileSystem* 用 RPC 调用 *NameNode*,在 *NameSpace* 中创建一个新的文件. *NameNode* 确定文件不存在且 client 有相应权限,创建新文件.
- 3 *DistributedFileSystem* 返回 *DFSOutputStream*,客户端用于写数据.
- 4 client 开始写入数据, *DFSOutputStream* 将数据分成 n 块,写入 *data queue*.
- 5 *Data queue* 由 *Data Streamer* 读取,并通知 *NameNode* 分配 *DataNode* 列表.该 *DataNode* 列表由副本系数为 m 的 $DN_{r \times m}$ *DataNode* 组中选取的 n 个 *DataNode* 节点组成,其 $DN_{r \times m}$ 组中每一行只能选取一个 *DataNode* 节点. *DataNode* 列表形成一个管道 *pipeline*.
- 6 *Data Streamer* 将数据块写入 *pipeline* 中的第一个 *DataNode*,第一个 *DataNode* 再将数据块传入 $DN_{r \times m}$ 组中同行的活动 *DataNode* 节点.若同行存在非活动 *DataNode* 节点, *NameNode* 将会继续创建备份.

- 7 *DFSOutputStream* 为发送数据块保存 *ack queue*,等待 *pipeline* 中的数据节点告知数据已经写入成功.
- 8 当 client 结束写入数据,则调用 *stream* 的 *close()* 函数.将所有的数据块写入 *pipeline* 中的数据节点,并等待 *ack queue* 返回成功.通知 *NameNode* 写入完毕.

4.2 文件级读取算法

HDFS 的文件读取策略为顺序的读取文件的 n 个数据块 $b_1, b_2, b_3, \dots, b_n$,直到全部数据块读取完成后,在 Client 端将数据块组合生成文件 F .当其他数据块所在 *DataNode* 节点网络资源空闲时,该策略会浪费大量空闲资源.

基于以上考虑,本文提出算法 4,文件级数据块读取算法,以提高资源利用率,同时降低文件读取访问时延.基本思想是同时生成多个下载进程并行地读取每一个数据块,最后将各个读取完成的数据块组装成完整文件.

算法 4:文件级读取算法

- 1 client 调用 *FileSystem* 的 *open()* 函数打开文件,建立与 *NameNode* 的连接.
- 2 *DistributedFileSystem* 通过 RPC 访问 *NameNode* 节点,得到文件的数据块信息.对于每一个数据块, *NameNode* 返回 *LocationBlocks* 给 *FSDDataInputStream*,即数据块所在的 *DataNode* 的地址.调用 *LocationBlocks.size()* 得到文件块数 n ,HDFS 产生 n 个读取线程,初始化访问标记 $flag[n] = \{0, 0, \dots, 0\}$.
- 3 每一个读取线程顺序读取 *DataNode* 节点信息,从 *availableDataNode* 中选择第一个 *DataNode* 作为访问点. client 调用 *FSDDataInputStream* 的 *read()*.
- 4 下载线程利用 *FSDDataInputStream.read()* 进行数据读取. *DFSInputStream* 连接保存此文件第 1 个 *Block* 的最近的 *DataNode*.若与 *DataNode* 通信出现错误,则尝试连接下一个数据节点.
- 5 当第 i 块 *Block* 下载完成,调用 *FSDDataInputStream.close()* 方法进行连接关闭和资源释放,设置访问标记数组 $flag[i] = 1$.
- 6 循环 $flag[n]$ 数组,当全为 1 则表示完成,将读取的 n 个数据块进行 *checksum* 校验,若校验通过将 n 个 *Block* 在 client 合并为完整文件.

4.3 数据块级读取算法

当 client 读取文件 F 中数据块 $b_i (1 \leq i \leq n)$ 时,只与存有该数据块或副本的距离最近的 *DataNode* 建立连接,直到读取完成,才能访问下一个数据块 b_{i+1} .即使选择最优的 *DataNode*,当数据块所在

其他节点 *DataNode* 节点网络资源空闲时,也会使大量空闲资源浪费.所以 *client* 与闲置 *DataNode* 建立连接,可以从数据块级并行读取多个副本以提高访问速度,降低延迟,如算法 5 所示.

算法 5:数据块级读取算法

- 1 *client* 调用对象 *DistributedFileSystem* 的 *Open()* 方法,建立与 *NameNode* 的连接.
- 2 *DistributedFileSystem* 调用 *RPC*, *NameNode* 返回 *LocationBlocks* 对象给 *DFSInputStream*.调用 *LocationBlocks.size()* 得到该文件块数 *n*,初始化访问校验数组 $file_flag[n] = \{0, 0, \dots, 0\}$.
- 3 从 *LocationBlocks* 取出一个未读取的 *LocationBlock* 对象,得到 *DataNodeInfo[]* 属性,以及该数据块或副本可用的 *DataNode* 列表 *availableDataNodes[]*.
- 4 若可用 *DataNode* 个数为 *t*,将该数据块的长度平均分解为 *t* 个区间,产生 *t* 个访问线程,初始化访问校验数组 $block_flag[t] = \{0, 0, \dots, 0\}$,为每一个线程分配一个 *DataNode* 的地址.
- 5 循环 $block_flag[t]$ 数组,当全为 1 表示该数据块读取完成,将 *t* 个数据块在 *client* 端合并为完整块.否则,继续读取.
- 6 循环 $file_flag[n]$ 数组,当全为 1 则表示完成.进行 *checksum* 校验,通过将 *n* 个数据组在 *client* 端合并为完整文件,否则,继续读取数据块.

5 实验与分析

5.1 实验环境及参数设置

本节使用 Matlab、cloudsim^[18] 对本文算法进行数值与实验模拟分析,通过性能评价与比较验证算法的有效性. Cloudsim 模拟产生的 *DataNode* 节点配置参数如表 1 所示. 利用 Cloudsim 模拟产生 5 个 RACK 且每个 RACK 中有 10 个 *DataNode* 节点即 $k=50$ 的 HDFS 集群,数据块副本默认值为 *m*. 集群中 51 个 PC,一个节点作为 *NameNode*,其余 50 个节点作为 *DataNode*. 每个节点为 2 GB 内存,100 GB 硬盘,操作系统为 Ubuntu Linux. 每个 *DataNode* 预留 1 GB 内存给 Mem-HDFS 和 100 GB 磁盘空间给 HDFS,因此,Mem-HDFS 为 5 TB 磁盘和 50 GB 内存. 为了模拟出文献[16]数据块访问规律,设文件在上传到集群后时间周期 *t* 的访问次数取 $random[0, 100]$, $2t$: $random[0, 90]$, $3t$: $random[0, 90]$, 依次类推, $10t$ 及以后访问次数取 $random[0, 10]$. 因此,根据访问次数配置热点数据块放入内存.

表 1 *DataNode* 节点配置参数

参数	属性
CPU(cores/MIPS)	1 core/10 000 mips
内存 RAM	2 GB
操作系统 OS	Linux
硬盘容量	100 GB
RACK 内部数据传输速度	RANDOM(8,10)
RACK 之间数据传输速度	RANDOM(4,5)
Client 到 HDFS 集群数据传输速度	RANDOM(2,3)

5.2 实验分析

基于磁盘的 HDFS 能够提供流式顺序访问,却不能提供高效的访问效率. 现有的数据和性能要求,使得它不能简单地用现有的技术来解决这个问题.

因此,文中提出基于内存的 Mem-HDFS 架构,基于同分布的数据存储策略和并行读取算法. 其中通过三个实验对比分析,(1)Mem-HDFS 和 HDFS 读写效率对比;(2)不同数据块策略对比;(3)并行算法与原 HDFS 数据块读取算法对比.

实验一:Mem-HDFS 和 HDFS 读写效率对比

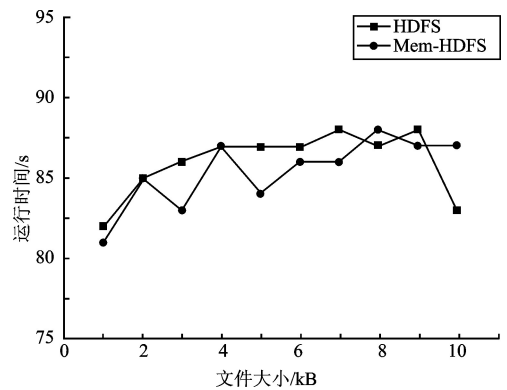
由于 Mem-HDFS 的目标是提高读取访问性能,因此实验对比文件存储在 HDFS 磁盘和 Mem-HDFS 中内存的读写性能.

首先,我们测量 Mem-HDFS 的写性能,结果展示在图 2(a)、(b),分别为写入 1 000 次小文件 1KB~10 kB 和写入 1 次大文件 100~1 000 MB 的结果. 由于 Mem-HDFS 直接将所有数据都写入底层 HDFS,与 HDFS 相同,所以 Mem-HDFS 的写性能与 HDFS 相近.

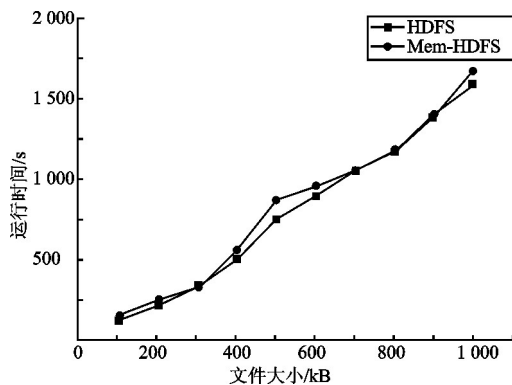
其次,我们测量 Mem-HDFS 两种情况下的随机访问性能:随机读取小文件 1 000 次 2 kB 到 10 kB 和写入 1 次大文件 100~1 000 MB 中随机查找和读取 1 kB 大小的数据. 结果展示在图 3(a)、(b). 当从 Mem-HDFS 中读取数据时,对于不同大小的文件,其平均读取开销保持平稳,而数据从 HDFS 中读取,平均读取开销随着文件大小的增长而增长.

实验二:不同数据块策略下文件写入效率对比

基于同分布的数据块存储策略,将同一文件的不同数据块存储至同一 rack,以提高访问文件的效率,同时会将同一数据块的多个副本存放至不同的 rack,带来额外的网络开销. 因此对比当取不同副本系数值时,不同数据块策略下的文件写入效率对比,实验结果如图 4 所示.

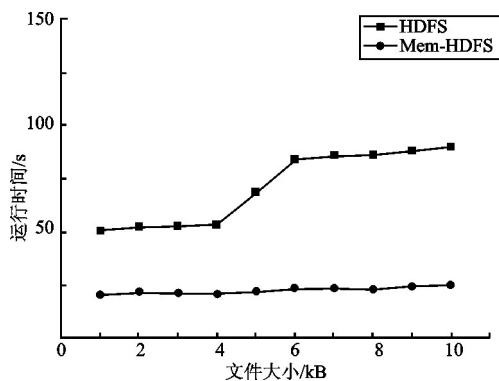


(a) 写入 1000 次小文件

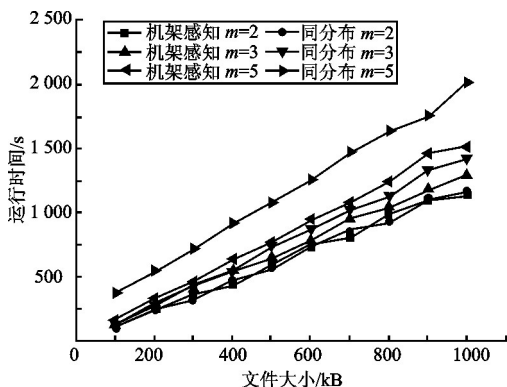


(b) 写入一次大文件

图2 Mem-HDFS 和 HDFS 写入效率对比



(a) 读取 1000 次小文件



(b) 读取一次大文件

图3 Mem-HDFS 和 HDFS 随机读取效率对比

可以看出当 $m=2$ 时,两种数据块策略文件上传时间相当.随着副本系数 m 值的增大,基于同分布的数据块存储策略效率相对较低.主要原因在于该策略数据块复制在不同的 rack 之间,而不同的 rack 之间的备份效率低于在同一 rack 内部之间的备份.

实验三:并行算法与原 HDFS 数据块读取算法对比

如图 5 所示原 HDFS 数据块读取算法与不同线程数并行算法的效率对比,其中黑色线 HDFS 表示原 HDFS 读取算法,其他表示在线程数(即同一数据块可用节点个数)取不同值时的下载效率.通过改变线程个数来分析文件大小对于下载时间的影响.

通过图 5 可以看出原 HDFS 数据块读取算法与并行算法效率对比时,两种算法都随着文件大小的增加读取时间也不断的增加.但随着文件大小的不断增加,并行算法效率比原 HDFS 变得更为高效.后三种算法由于采用了多线程访问,随着文件的增大,下载线程数量增多,下载时间增长并不明显,较好的提高了读取访问效率.

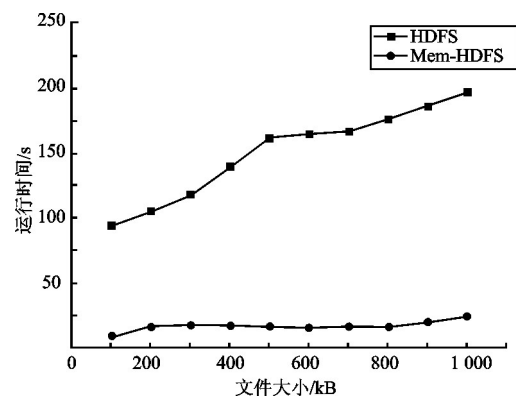


图4 不同数据块策略下的文件写入效率对比

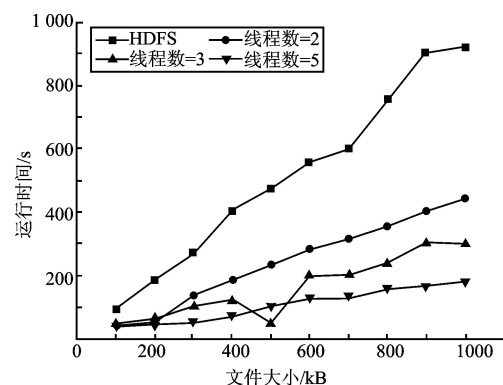


图5 并行算法效率对比

6 结束语

本文对 HDFS 中读取数据块的时延问题进行了分析,提出基于内存的 Mem-HDFS 架构,在保证原 HDFS 可靠性、高效性与可扩展性等优点的同时,通过数据配置算法将访问频繁的数据块存入内存,以提高读取访问的效率.同时提出同分布数据块的存储策略和该数据块存储策略下的数据块读取与写入算法.同分布数据块存储策略通过数据块本地化来提高 DataNode 节点的数据块访问效率,并同时从文件级和数据块级两个方面提出并行读取访问算法.新的数据块读取算法通过配置并行访问线程数,提高了 HDFS 读取文件的效率.新的数据块写入算法较原算法虽在性能上不如原算法,但能更好地匹配基于同分布的数据块存储策略的要求.通过模拟实验表明,算法能够较好的降低读取数据的延迟.

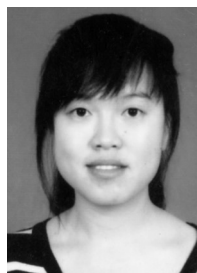
下一步工作主要集中在以下几个方面:(1)对于低延迟算法的传输优化和调度优化研究;(2)探寻在内存分布式文件系统特点下数据块动态配置算法中参数 t 、 n 、 k 的最佳值;(3)针对 mapreduce 的低延迟访问策略改进;(4)基于内存数据库如 GemFire 的内存计算优化;(5)基于分布式文件系统应用数据层的低延迟访问是将来研究的另一个方向.

参考文献:

- [1] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing[J]. Communications of the ACM, 2010, 53(4): 50-58.
- [2] Yifeng L, Siqiang L, Jihong G, et al. A RAMCloud Storage System based on HDFS: Architecture, implementation and evaluation[J]. Journal of Systems and Software, 2013, 86(3): 744-750.
- [3] Ousterhout J, Agrawal P, Erickson D, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM[J]. ACM SIGOPS Operating Systems Review, 2010, 43(4): 92-105.

- [4] 廖彬,于炯,张陶,等. 基于分布式文件系统 HDFS 的节能算法[J]. 计算机学报, 2013, 36(5): 1047-1064.
- [5] Nitesh M, Nanduri R, Varma V. Dynamic energy efficient data placement and cluster reconfiguration algorithm for MapReduce framework [J]. Future Generation Computer Systems, 2011, 28(1): 119-127.
- [6] 廖彬, 于炯, 张陶, 等. 一种适应节能的云存储系统元数据动态建模与管理方法[J]. 小型微型计算机系统, 2013, 10(34): 2407-2412.

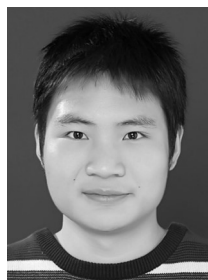
作者简介:



英昌甜 女,(1989-),博士研究生.研究方向为分布式文件系统、内存计算.



于 炯 男,(1964-),教授,博士.研究方向为网络安全、网络与分布式计算.



廖 彬 男,(1986-),博士研究生.研究方向为绿色计算、数据库.

鲁 亮 男,(1989-),硕士研究生.研究方向为云计算、分布式计算.