Journal of Computer Applications
计算机应用,2014,34(11):3091-3095,3099
ISSN 1001-9081
CODEN JYIIDU
2014-11-10
http://www.joca.cn

文章编号:1001-9081(2014)11-3091-05
doi:10.11772/j.issn.1001-9081.2014.11.3091

面向 Hadoop 分布式文件系统的小文件存取优化方法

李　铁,燕彩蓉*,黄永锋,宋亚龙
(东华大学 计算机科学与技术学院,上海 201620)
(*通信作者电子邮箱 cryan@dhu.edu.cn)

摘　要:为提高 Hadoop 分布式文件系统(HDFS)的小文件处理效率,提出了一种面向 HDFS 的智能小文件存取优化方法——SmartFS。SmartFS 通过分析小文件访问日志,获取用户访问行为,建立文件关联概率模型,并根据基于文件关联关系的合并算法将小文件组装成大文件之后存至 HDFS;当从 HDFS 获取文件时,根据基于文件关联关系的预取算法来提高文件访问效率,并提出基于预取的缓存替换算法来管理缓存空间,从而提高文件的命中率。实验结果表明,SmartFS 有效减少了 HDFS 中 NameNode 的元数据空间,减少了用户与 HDFS 的交互次数,提高了小文件的存储效率和访问速度。

关键词:Hadoop 分布式文件系统;小文件;文件关联;预取;缓存
中图分类号:TP316.4　　**文献标志码**:A

Optimization of small files storage and accessing on Hadoop distributed file system

LI Tie, YAN Cairong*, HUANG Yongfeng, SONG Yalong
(*School of Computer Science and Technology, Donghua University, Shanghai* 201620, *China*)

Abstract: In order to improve the efficiency of processing small files in Hadoop Distributed File System (HDFS), a new efficient approach named SmartFS was proposed. By analyzing the file accessing log to obtain the accessing behavior of users, SmartFS established a probability model of file associations. This model was the reference of merging algorithm to merge the relevant small files into large files which would be stored on HDFS. When a file was accessed, SmartFS prefetched the related files according to the prefetching algorithm to accelerate the access speed. To guarantee the enough cache space, a cache replacement algorithm was put forward. The experimental results show that SmartFS can save the metadata space of NameNode in HDFS, reduce the interaction between users and HDFS, and improve the storing and accessing efficiency of small files on HDFS.

Key words: Hadoop Distributed File System (HDFS); small file; file relation; prefetching; caching

0　引言

Hadoop 是近几年发展比较成熟的云计算平台之一,凭借其可靠、高效、可伸缩的特性在互联网领域得到了广泛应用。Hadoop 分布式文件系统(Hadoop Distributed File System, HDFS)已经成为海量存储集群上部署的主流文件系统。但 HDFS 主要是为了存储、分析大文件,忽略了存取小文件的问题,而当今各个领域,比如能源、天文、电子商务、在线学习等包含了大量的小文件。HDFS 不适合处理小文件的原因有以下几点:1)大量的小文件占用了 NameNode 大量空间,导致存储效率低下;2)HDFS 适合大文件的存取机制并不适合小文件;3)HDFS 本身缺乏 IO 优化策略,如预取和缓存[1]。

当前 HDFS 小文件优化的基本思想主要体现为:1)合并思想,即将小文件合并成大文件来缩减 NameNode 的元数据大小,并建立一套索引机制来解决小文件的读取问题。已有文献的合并过程基本都是静态的,即将事先已知的有关联的小文件合并在一起存至 HDFS[1-9]。2)预取与缓存思想,即通过预取块内其他小文件来提高小文件下载效率和减少与 HDFS 的交互。预取与缓存是建立在合并和块内文件关联的基础上,访问块内某一文件时将该块预取并缓存。不过已有文献的预取与缓存思想没有相应的针对预取的缓存替换策略[1-4]。

针对上述 Hadoop 小文件处理存在的问题,本文提出了一种面向 HDFS 的小文件处理优化方法——SmartFS(Smart File System)。它首先分析大量的历史文件访问日志得到其文件关联性,将相关联的文件合并到一起存至 HDFS 中,这是一种动态的基于文件关联的小文件合并方法,保证了块内文件的关联性;然后通过预取算法将关联文件预取出,其预取算法综合考虑了文件关联概率、用户等待时间、预取消耗因素;在缓存替换策略上本文提出了基于预取的缓存替换算法——Prefetching-LFU(Least Frequently Used with Prefetching),它改进了带有时间变化的最不经常使用算法——LFU-Aging(Least Frequently Used with Aging),增大了预取文件的缓存权值,避免了最近预取的文件还未被访问就被替换的情况。

收稿日期:2014-06-05;**修回日期**:2014-08-31。　**基金项目**:国家自然科学基金资助项目(61300100, 61402100);中央高校基本科研业务费专项资金资助项目(14D111210);上海市自然科学基金资助项目(13ZR1451000)。

作者简介:李铁(1989-),男,湖南永州人,硕士研究生,主要研究方向:分布式存储、分布式计算;　燕彩蓉(1978-),女,湖北仙桃人,副教授,博士,主要研究方向:并行计算、分布式计算、大数据处理;　黄永锋(1971-),男,山东泰安人,副教授,博士,主要研究方向:数据挖掘、机器学习、图像处理;　宋亚龙(1988-),男,河南鹤壁人,硕士研究生,主要研究方向:分布式存储、分布式计算。

1 SmartFS 架构

SmartFS 由离线日志分析模块、合并模块及在线预取模块三部分组成，如图 1 所示。

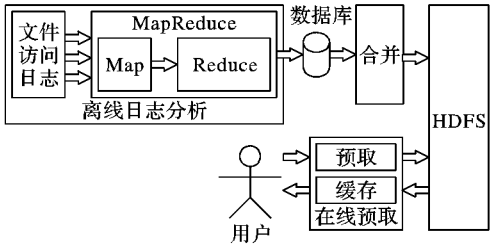

图 1 SmartFS 架构

1）离线日志分析模块：使用 Hadoop 中的 MapReduce 通过分析大量的历史文件访问日志得到文件之间的关联性，并将文件的关联性保存至数据库中。

2）合并模块：根据数据库中保存的文件之间的关联性，将相关联的小文件合并成大文件存储到 HDFS 中。

3）在线预取缓存模块：当用户访问某一文件时，将该文件与其相关联的部分文件从 HDFS 预取出来并存储到缓存中。

2 基于关联的小文件合并

2.1 文件关联概率

很多应用之间的文件是有关联的，比如一篇博文的两张插图 A 和 B，访问 A 后很可能会访问 B，假设这两张图片保存在 HDFS 中，这两张图片是不存在关联的，可以通过历史的访问日志来分析得到 A 与 B 的关联度从而将 A 与 B 合并成一个块，当访问 A 的时候将 B 预取出来，这样既减少了 NameNode 的元数据大小又通过预取提高了文件访问速度。

本文介绍的文件关联算法是通过分析 Web 日志的文件访问顺序得到文件的关联性。文件访问日志示例如表 1 所示。

表 1 文件访问日志示例

Time	File Id	File Path	Status
2013-09-01T12:01:00	0001	/user/book1/file1	200
2013-09-01T12:01:03	0002	/user/book1/file2	200
2013-09-01T12:01:05	0003	/user/book1/file3	200
2013-09-01T12:01:06	0005	/user/book1/file5	200
2013-09-01T12:01:07	0100	/user/book1/file1	404
2013-09-01T12:01:10	0003	/user/book1/file3	200
2013-09-01T12:01:15	0008	/user/book1/file8	200
2013-09-01T12:02:20	0001	/user/book1/file1	200
2013-09-01T12:02:25	0002	/user/book1/file2	200
2013-09-01T12:02:31	0007	/user/book1/file7	200

定义 1 $P(B \mid A)$：表示用户访问文件 A 后 T 时间范围内访问文件 B 的概率，计算公式如式(1) 所示。

$$P(B \mid A) = \frac{N_b}{N_a} \times 100\% \tag{1}$$

其中：N_a 表示访问日志中 A 文件出现的次数，N_b 表示访问 A 后在 T 时间范围内 B 的出现次数。$P(B \mid A)$ 值越大表示文件 A 与 B 的关联性就越大，$P(B \mid A) = 100\%$ 表示访问 A 后 T 时间内必然访问 B。

2.2 文件关联算法

计算文件的关联概率首先要统计每个文件出现的次数，并分析每个文件后面 T 时间内访问的相关联文件及次数，再利用式(1) 计算文件的关联概率。文件关联统计如算法 1 所示。

算法 1 文件关联统计算法。

```
输入  list(logs)。
输出  {cnt, relatedFiles}。
1)    stat(logs) {
2)      cnt = {}                                  //每个文件的访问次数
3)      relatedFiles = {}                         //访问次数
4)      //所有之前时间差在 T 内的 log
5)      everLogs = []
6)      for log in logs {
7)        for var i = 0 -> len(everLogs) {
8)          logEver = everLogs[i]
9)          If (logEver.time - log.time < T){
10)           everLogs.push(log)
11)         break;
12)       } else {
13)           name = logEver.fileName
14)           cnt[name]++
15)           for i = j -> len(everLogs) {
16)             relatedFiles[logEver][everLogs[j].fName]++
17)           }
18)           everLogs.dequeue();
19)       }
20)     }
21)   }
22)   return cnt, relatedFiles;
23)   }
```

当日志文件较大时采用传统串行方式执行算法 1 来统计关联文件速度较慢，此时可以使用 MapReduce 结合算法 1 来计算关联概率。

2.2.1 文件关联 Map 算法

对于每个 Map 执行算法 1，Map 的输出为 <*fileName*，{*count*：*n*，relatedFiles：<*fileI*：*n* >} >，其中 *key* 为文件名。*value* 有两类数据：*count* 表示该 Map 中 *fileName* 的数目，*relatedFiles* 表示该文件下在 T 时间内访问的集合。如算法 2 所示。

算法 2 文件关联 Map 算法。

```
输入  list(logs)
输出  <filename, {cnt, relatedFiles} >
1)    Map (key,logs) {
2)      cnt, relatedFiles = stat(logs);
3)      for file in relatedFiles {
4)        Emit(file.getFileName(),{
5)        count: cnt[file],
6)        relatedFiles: relatedFiles[file]);
7)      }
8)    }
```

2.2.2 文件关联 Reduce 算法

Reduce 中将每个 *fileName* 的 *count*，*relatedFiles* 作累加统计，再计算关联概率，如算法 3 所示。

算法 3　文件关联 Reduce 算法。

```
输入　<fileName, list({cnt, relatedFiles})>。
输出　<fileName|relatedFileName, p>。
1)　Reduce(filename, values) {
2)　totalCount = 0;
3)　relatedFiles = {}
4)　for value in values {
5)　  totalCount += value. count;
6)　  for file in value. relatedFiles {
7)　    relatedFiles[file[key]] += file[value]
8)　  }
9)　}
10)　for relatedFile in relatedFiles {
11)　  Emit(filename + "|" + relatedFiles. key,
12)　    relatedFiles. value/totalCount);
13)　  }
14)　}
```

若 T 为 10 s,那么计算表 1 的日志可得到如表 2 所示的结果(取 A 为 file1)。

表 2　文件关联概率

$B\|A$	N_b	N_a	$P(B\|A)/\%$
file2|file1	2	2	100
file3|file1	1	2	50
file5|file1	1	2	50
file6|file1	1	2	50
file7|file1	1	2	50

相应地 A 为其他文件时也可得到其关联文件概率。

2.3　小文件合并算法

假设有一本书中的所有图片都存到了 HDFS 中,读者从前往后阅读该书时就会依次从 HDFS 中访问图片,假设其访问顺序为(A,B,C,D,E,F),如果将(A,B,C,D,E,F)合并成一个大文件存至 HDFS 中,那么当访问 A 时就将该块其他文件预取出来存至缓存中,当访问 B、C、D、E 和 F 时只需从缓存中读取即可,这势必会提高访问效率。该思路需要解决的问题有:1)确定触发文件,触发文件为当访问该文件时预取与其关联文件的文件;2)确定将哪些文件进行合并。

一个块中只有一个触发文件,其他文件都是该文件的关联文件,那么只有访问该触发文件时才将整块预取出来,访问其他文件时若不能命中缓存只将该文件从 HDFS 中取出,不触发预取。

如(A,B,C,D,E,F),A 为触发文件,其他文件都是 A 的关联文件。

2.3.1　确定触发文件

触发文件是该块文件的关键,只要访问了该触发文件,该块其他文件很可能在接下来一段时间内访问到,所以必须要确定触发文件,那么触发文件必须是经常访问到的文件,且有多个关联文件的概率大于 50%,按照该思路,可得到确定触发文件的算法。

由算法 2 得到各个文件的关联文件概率,及每个文件的访问次数;将各个文件按访问次数以逆序排序得到集合 S;遍历集合 S,判断文件的关联文件概率大于 50% 的文件数是否大于 M,若为真则将文件加入触发文件集合 *TriggerFiles* 中。如算法 4 所示。

算法 4　确定触发文件。

```
输入　list(logs)。
输出　list(files)。
1)　getTriggerFiles(logs) {
2)　  triggerFiles = [ ]; M = 2;
3)　  eachFileTimes, raletedFiles =
        getProbabilities(logs);
4)　  eachFileTimes. asort( );
5)　  for file in eachFileTimes {
6)　    if times(raletedFiles [file] > 0.5) > M
7)　      triggerFiles. append(file)
8)　  }
9)　  return triggerFiles;
10)　}
```

2.3.2　文件合并

通过确定触发文件算法得到触发文件集合 *triggerFiles*,遍历每一个触发文件,将该文件与其相关联的文件合并成一个不大于块大小(64 MB)的大文件,并将该大文件存储到 HDFS。如算法 5 所示。

算法 5　合并算法。

```
输入　list(TriggerFiles)。
输出　HDFS blocks。
1)　merge(TriggerFiles) {
2)　  for triggerFile in TriggerFiles {
3)　    if (triggerFile not in mergedFiles) {
4)　      continue;
5)　    }
6)　    for file in raletedFiles[triggerFile] {
7)　    if (totalSize + fileLength(file) < 64 MB)
8)　      mergedFile. push(file)
9)　    }
10)　   block = mergeFile(mergedFiles)
11)　   store block in HDFS
12)　 }
13)　}
```

需要注意的方面:1)为了避免跨块请求文件,合并后的文件不能大于 HDFS 块大小;2)为了保证不产生冗余数据,已经合并后的文件不能作为触发文件并且不能在其他合并文件中。

3　小文件预取与缓存

3.1　小文件预取

由基于文件关联的小文件合并算法已将有关联的小文件合并在一个块中,传统的预取方法是当访问某一文件时将该文件所在块的所有文件预取出来。在实验中发现,因为 HDFS 块大小远远大于一个小文件的大小,当只访问某一小文件时却要将整块都读出,这虽然能够减少之后访问文件的时间,但却增加了本次请求的时间。虽然在块的文件是有关联的,还是不能将所有文件都预取,需要综合考虑文件关联概率、用户等待时间、预取消耗因素。

定义 2　*SRTT*(SmartFS Round Trip Time):表示 SmartFS 接收到用户请求开始到 SmartFS 发送结果给用户的时间。该时间包含 SmartFS 访问 HDFS 和处理数据,不包含预取与缓存时间。

定义 3　*CRTT*(Cache Round Trip Time):表示 SmartFS 接收到用户请求开始到 SmartFS 发送结果给用户的时间,该时

间只包含 SmartFS 从缓存中访问文件的时间。

定义 4 *PT*(Prefetching Time):表示预取一个文件的时间消耗,那么访问某文件并预取一个文件的总时间为 *SRTT* + *PT*。访问 2 个文件的总时间如式(2)所示:

$$COST = SRTT + PT + CRTT \tag{2}$$

若不采取预取,那么访问 2 个文件的总时间为 2*SRTT*,因为 *CRTT* 的时间主要是访问内存的时间,可以忽略不计,而 *PT* 主要是访问 HDFS 时间(若该文件已在缓存中则 *CRTT* 为 0),故 *COST* < 2*SRTT*,故可以采用预取策略。将文件关联的概率 *P* 计算在内,其评判是否预取的标准如式(3)所示:

$$SRTT + (PT + CRTT) \times P < (1 - P) \times 2SRTT \tag{3}$$

若有多个文件与当前文件存在关联,则将关联文件的概率由大到小依次代入式(3)直到不满足为止。

定义 5 *MSRTT*(Max SRTT):表示 SmartFS 最大处理时间,也可等价为用户等待时间上限,当预取过多文件时会影响当前文件的响应时间,若 *SRTT* 大于 *MSRTT* 会影响用户体验。

因为存在 *MSRTT*,为了保证当前文件能及时响应,所以即使式(3)成立也不能将符合条件的文件预取。最多能预取的文件个数的计算公式如式(4)所示:

$$MN = \left\lceil \frac{MSRTT - SRTT}{PT} \right\rceil \tag{4}$$

通过式(3)~(4)即可判断哪些文件需要预取,其算法如算法 6 所示。

算法 6 预取算法。

```
输入 fileName。
输出 list(File)。
1)   prefetch (fileName) {
2)     if(file in cache) return null;
3)     relatedFiles = getRelatedFiles(fileName);
4)     files = [ ];
5)     mNum = math.floor((MSRTT - SRTT)/PT)
6)     for(i, file in relatedFiles) {
7)       if(i > maxNum) break;
8)       p = getProbility(fileName, file)
9)       if(SRTT + (PT + CRTT) * P < (1 - P) * 2SRTT)
10)      files.push(file)
11)    }
12)    return files;
13)  }
```

3.2 缓存替换算法

因为内存的容量有限,所以不能无限将预取的文件缓存到内存中,需要一定的替换策略。

目前常见的缓存替换算有:1)最近最少使用(Least Recently Used,LRU)算法,是 Web 缓存中最常使用的一种算法,其原理是把最近一次访问时间最长的对象从缓存中移除。该算法实现简单但没有考虑到对象访问频率、对象大小等其他因素。2)最不经常使用(Least Frequently Used,LFU)算法,其原理是替换访问频率最小的 Web 对象。该算法的缺点是如果一段时间内访问频率较高的对象一直保存在缓存中,可能会导致缓存“垃圾”的出现。3)LFU-Aging 算法,它在 LFU 基础上增加了时间因素,能解决 LFU 缓存污染的问题,但没有针对预取的优化。

文献[10]采取的是将预测与缓存机制相结合的缓存替换策略;文献[11]使用文档大小、访问频率、文档访问剩余寿命作为计算文档价值的要权衡缓存的页面。本文提出的 Prefetching-LFU 与其思想相仿,都是基于预取的结果进行分析是否需要缓存替换。

本文采用文献[12]的带有时间变化的最不经常使用改进算法 LFU*-Aging,该算法实现了缓存替换功能,但在实验过程中发现命中率并不理想,检查发现将新预取的文件保存到缓存中并设置每个文件的计数器设为 1,但在 *T* 时间内却不能命中缓存的文件,这是因为在 *T* 时间内又有新的预取文件替换了之前预取但未访问的文件。经过分析本文提出了基于预取的缓存替换算法 Prefetching-LFU。

Prefetching-LFU 的每个缓存块都有一个引用计数 *count*、创建时间 *createdTime* 和最近访问时间 *accessTime*。

Prefetching-LRU 缓存替换算法包含获取缓存 get()与添加缓存 put()两个算法。

3.2.1 获取缓存算法

当预取时命中了缓存区的文件,将该文件的 *count* 为 1,*createdTime* 设为当前时间,*accessTime* 设为空;当命中某个文件时将该文件的 *count* 加 1,*createdTime* 设为空,*accessTime* 设为当前时间。如算法 7 所示。

算法 7 获取缓存算法。

```
输入 fileName,isPrefetch。
输出 fileContent。
1)   get(fileName, isPrefetch) {
2)     file = cache.get(fileName)
3)     if(!file) return;
4)     if (isPrefetch) {
5)       file.count = 1
6)       file.createdTime = now;
7)       file.accessTime = null
8)     } else {
9)       file.count ++
10)      file.createdTime = null;
11)      file.accessTime = now;
12)    }
13)  }
```

3.2.2 添加缓存算法

将预取文件放入缓存区时,每个文件 *count* 为 1,*createdTime* 设为当前时间,*accessTime* 设为空。

当缓存区不足需要替换文件时:首先查找 *count* 为 1,*createdTime* 不为空的数据,判断 *createdTime* 与当前时间差是否超过 *T*,若全部未超过则查找 *count* 最小且 *accessTime* 最久的文件将其替换。若有超过 *T* 的将其置换,若没有找到可替换的文件则不替换。如算法 8 所示。

算法 8 添加缓存算法。

```
输入 file。
输出 cache file。
1)   put(file) {
2)     if (hasCapacity()) {
3)       cache.put(
4)       { fileName: file.fileName,
5)         count:1,
6)         createdTime:null,
7)         accessTime:null,
8)         content: file.content});
```

```
9)        } else {
10)           for(cacheFile in cache) {
11)             if (file. count ==1 &&
12)               file. createdTime != null &&
13)               now-file. createdTime > T) {
14)               cacheFile =file;
15)               return;
16)             }
17)           }
18)           cacheFile =
19)           cache. findMinCountMaxAccessTime( )
20)           cacheFile =file;
21)       }
22)   }
```

该算法可以确保已预取且时间未超过 T 未访问的文件不会被替换。

4　实验与分析

4.1　实验环境

本实验环境基于 4 个节点的 Hadoop 集群,其中:NameNode 是 16 GB 内存,Intel Xeon CPU E5530 2.40 GHz,1 TB硬盘;3 个 DataNode 是 16 GB 内存,Intel Xeon CPU E5530 2.40 GHz,2 TB 硬盘。

每个节点的操作系统都是 Ubuntu server 12.04,Hadoop 版本 1.2.1,Java 版本 1.6.0,HDFS 副本数为 3,HDFS 块大小是 64 MB。

4.2　内存使用测试

本实验使用的指标:文件平均内存使用量(Average Memory Consuming Per File, AMCPF) = 总内存使用/文件个数。

本实验对 HDFS、Hadoop Archives(HAR 文件)分别上传 2 000,4 000,6 000,8 000,10 000 个小文件。对 SmartFS 随机产生 1 000 000 访问日志作为文件关联算法的数据,先通过学习建立文件关联关系并将关联文件合并,分别合并 2 000,4 000,6 000,8 000,10 000 个小文件。内存使用量结果如图 2 所示。

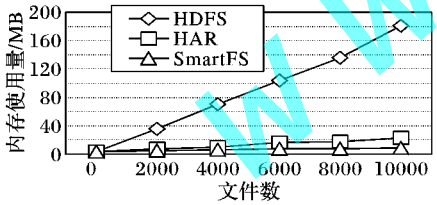

图 2　HDFS、HAR 与 SmartFS 内存使用量对比

HDFS 的 $AMCPF$ = 0.018 MB,HAR 的 $AMCPF$ = 0.002 2 MB,SmartFS 的 $AMCPF$ = 0.000 8 MB。从图 2 中可以看到 HDFS 的内存使用量呈线性增加,对于海量的小文件这将是一个瓶颈;HDFS 的小文件优化方案 HAR 能显著地减少内存使用;而本文提出的 SmartFS 使得内存使用量再一次降低,能容纳海量的小文件。

4.3　文件下载测试

本实验所使用的指标:平均下载时间(Average Download Time Per File, ADTPF) = 总下载时间/文件个数。

首先随机产生 1 000 000 访问日志作为文件关联算法的数据,先通过学习建立文件关联关系并将关联文件合并,然后使用 SmartFS 和 HDFS 根据访问日志分别访问 2 000,4 000,6 000,8 000,10 000 次并记录每次实验的时间。实验结果如图 3 所示。

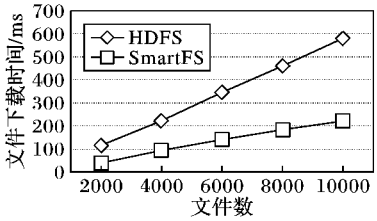

图 3　HDFS 与 SmartFS 的文件下载时间对比

SmartFS 的 $ADTPF$ = 0.020 ms,HDFS 的 $ADTPF$ = 0.055 ms。从图 3 可看出 SmartFS 的速度比 HDFS 快,其速度提高了 2.7 倍,证明了 SmartFS 基于文件关联能有效预取出关联文件,从而提高了文件的下载速度。

4.4　缓存替换算法对比测试

SmartFS 分别使用 LFU*-Aging 缓存替换算法与 Prefetching-LFU 缓存替换算法。其过程与文件下载测试实验步骤一致,不再赘述。结果如图 4 所示。

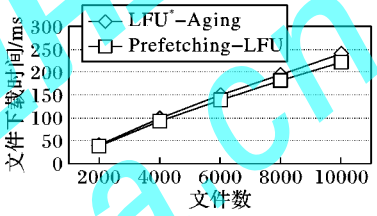

图 4　LFU*-Aging 与 Prefetching-LFU 文件下载时间对比

Prefetching-LFU 的 $ADTPF$ = 0.020 ms, LFU*-Aging 的 $ADTPF$ = 0.025 ms。从图 4 可看出,Prefetching-LFU 缓存替换算法效率要略高于 LFU*-Aging 缓存替换算法,因为该算法可以确保已预取的文件且时间未超过 T 的文件不会被替换。

5　结语

本文就 HDFS 缺乏预取与缓存机制的问题提出了一个面向 HDFS 的小文件处理方法 SmartFS,它包含离线日志分析、合并、在线预取三大模块。SmartFS 基于历史访问日志的文件关联算法来对文件建立关联,克服了传统的小文件合并基于事先已知的文件关联或基于时间的文件关联这种静态关联方法的不足,是一种动态的基于文件历史访问的文件关联合并方法;在文件的预取上 SmartFS 提出了基于文件关联关系的预取算法,它综合考虑了文件关联、用户等待时间、预取消耗因素;在文件的缓存上 SmartFS 提出了基于预取的缓存替换算法 Prefetching-LFU,它改进了 LFU-Aging 算法,加大预取文件的缓存权值,避免最近预取的文件还没被访问就被替换的情况。实验表明 SmartFS 能够通过合并文件减少 NameNode 元数据大小,并能正确预测并预取文件,提高了 HDFS 的文件下载速度。本文提出的文件关联算法没有考虑路径访问的关联概率,仅仅考虑单个文件被访问的情况,这是需要改进的,也是今后的研究方向,即改进文件关联合并算法,增加缓存命中概率。

参考文献:

[1]　DONG B, QIU J, ZHENG Q, *et al*. A novel approach to improving the efficiency of storing and accessing small files on Hadoop: a case study by PowerPoint files [C]// SCC 2010: Proceedings of the 7th IEEE International Conference on Services Computing. Piscataway: IEEE Press, 2010: 65 - 72.

(下转第 3099 页)

得越细,这种数据量在多核处理器上可以较快地完成。

通过这三个程序的实验测试表明,waxberry 在执行上可以获得较好的性能,并且性能要优于 JOMP。

3　结语

本文针对 JOMP 在提升程序性能方面存在的不足,提出了一个并行框架,该框架可以分离并行程序中的并行逻辑,采用面向方面和运行时反射技术以并行库的方式实现了该并行框架。在实验中,使用 JGF 基准测试程序对并行库的性能进行了测试,并与 JOMP 进行了对比分析,实验结果表明本文设计的并行库在性能上要优于 JOMP。

参考文献:

[1] WILKINSON B, ALLEN M. Parallel programming[M]. Upper Saddle River: Prentice Hall, 1999.

[2] STAS N, MIHAI C, DANNY D, *et al.* Mining fine-grained code changes to detect unknown change patterns[C]// ICSE 2004: Proceedings of the 36th International Conference of Software Engineering. New York: ACM Press, 2014:803 – 813.

[3] LIU X, YUE L, CHEN B, *et al.* Method of target recognition in remote sensing images based on parallel processing [J]. Journal of Computer Applications, 2007, 27(9): 2123 – 2125. (刘晓沐, 岳丽华, 陈博, 等. 遥感图像目标识别的并行处理方法[J]. 计算机应用, 2007, 27(9): 2123 – 2125.)

[4] OTTO F, PANKRATIUS V, TICHY W F. XJava: Exploiting parallelism with object-oriented stream programming[C]// Proceedings of the 15th International Euro-Par Conference, LNCS 5704. Berlin: Springer-Verlag, 2009: 875 – 886.

[5] KOSTOPOULOS S, GLOTSOS D, SIDIROPOULOS K. A pattern recognition system for prostate mass spectra discrimination based on the CUDA parallel programming model[J]. Journal of Physics: Conference Series, 2014, 490(1): 120 – 144.

[6] SMITH B. A new parallel programming model for computer simulation[EB/OL]. [2014-02-10]. http://www.mcs.anl.gov/papers/P5135-0414.pdf.

[7] NOBRE R, PINTO P, CARVALHO T, *et al.* On expressing strategies for directive-driven multicore programing models[C]// PARMA-DITAM 2014: Proceedings of the 2014 Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms. New York: ACM Press, 2014: 7 – 14.

[8] DAGUM L, MENON R. OpenMP: an industry standard API for shared-memory programming[J]. Computational Science and Engineering, 1998, 5(1): 46 – 55.

[9] BULL J M, KAMBITES M E. JOMP — an OpenMP-like interface for Java[C]// Proceedings of the ACM Conference on Java Grande. New York: ACM Press, 2000: 53 – 61.

[10] SMITH L A, BULL J M, OBDRIZALEK J A. Parallel Java Grande benchmark suite [C]// Proceedings of the 2006 ACM/IEEE Conference of Supercomputing. New York: ACM Press, 2006: 6 – 13.

[11] KICZALES G, LAMPING J, MENDHEKAR A, *et al.* Aspect-oriented programming[C]// Proceedings of the 1997 Europe Conference of Object-Oriented Programming, LNCS 1241. Berlin: Springer-Verlag, 1997: 220 – 242.

[12] TANTER E, FIGUEROA I, TABAREAU N. Execution levels for aspect-oriented programming: design, semantics, implementations and applications [J]. Science of Computer Programming, 2014, 80(2): 311 – 342.

(上接第 3095 页)

[2] LIU X, HAN J, ZHONG Y, *et al.* Implementing WebGIS on Hadoop: a case study of improving small file I/O performance on HDFS [C]// CLUSTER 2009: Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops. Piscataway: IEEE Press, 2009: 1 – 8.

[3] JIANG L, LI B, SONG M. The optimization of HDFS based on small files [C]// IC-BNMT 2010: Proceedings of the 3rd IEEE International Conference on Broadband Network and Multimedia Technology. Piscataway: IEEE Press, 2010: 912 – 915.

[4] DONG B, ZHENG Q, TIAN F, *et al.* An optimized approach for storing and accessing small files on cloud storage[J]. Journal of Network and Computer Applications, 2012, 35(6): 1847 – 1862.

[5] GOHIL P, PANCHAL B. Efficient ways to improve the performance of HDFS for small files[J]. Computer Engineering and Intelligent Systems, 2014, 5(1): 45 – 49.

[6] CHEN J, WANG D, FU L, *et al.* An improved small file processing method for HDFS[J]. International Journal of Digital Content Technology and its Applications, 2012, 6(20): 296 – 304.

[7] HUA X, WU H, LI Z, *et al.* Enhancing throughput of the Hadoop distributed file system for interaction-intensive tasks[J]. Journal of Parallel and Distributed Computing, 2014, 74(8): 2770 – 2779.

[8] CHANDRASEKAR S, DAKSHINAMURTHY R, SESHAKUMAR P, *et al.* A novel indexing scheme for efficient handling of small files in Hadoop distributed file system [C]// ICCCI 2013: Proceedings of the 2013 International Conference on Computer Communication and Informatics. Piscataway: IEEE Press, 2013: 1 – 8.

[9] MACKEY G, SEHRISH S, WANG J. Improving metadata management for small files in HDFS [C]// CLUSTER 2009: Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops. Piscataway: IEEE Press, 2009: 1 – 4.

[10] HAO Q, ZHU M, HAO J. New proxy cache replacement policy [J]. Journal of Computer Research and Development, 2002, 39(10): 1178 – 1185. (郝沁汾, 祝明发, 郝继升. 一种新的代理缓存替换策略[J]. 计算机研究与发展, 2002, 39(10): 1178 – 1185.)

[11] SHI L, MENG C, HAN Y. Web replacement policy based on prediction[J]. Journal of Computer Applications, 2007, 27(8): 1842 – 1845. (石磊, 孟彩霞, 韩英杰. 基于预测的 Web 缓存替换策略[J]. 计算机应用, 2007, 27(8): 1842 – 1845.)

[12] JIN Z, ZHANG G. Intelligent prefetch and cache techniques based on network performance[J]. Journal of Computer Research and Development, 2001, 38(8): 1000 – 1004. (金志刚, 张钢. 基于网络性能的智能 Web 加速技术——缓存与预取[J]. 计算机研究与发展, 2001, 38(8): 1000 – 1004.)