一种面向 HDFS 中海量小文件的存取优化方法*

顾玉宛，王文闻，孙玉强†
(常州大学 信息科学与工程学院，江苏 常州 213164)

摘 要： 为了解决 HDFS(Hadoop distributed file system) 在存储海量小文件时遇到的 NameNode 内存瓶颈等问题，提高 HDFS 处理海量小文件的效率，提出一种基于小文件合并与预取的存取优化方案。首先通过分析大量小文件历史访问日志，得到小文件之间的关联关系，然后根据文件相关性将相关联的小文件合并成大文件后再存储到 HDFS。从 HDFS 中读取数据时，根据文件之间的相关性，对接下来用户最有可能访问的文件进行预取，减少了客户端对 NameNode 节点的访问次数，提高了文件命中率和处理速度。实验结果证明，该方法有效提升了 Hadoop 对小文件的存取效率，降低了 NameNode 节点的内存占用率。

关键词： 海量小文件；文件相关性；合并；预取
中图分类号： TP391 **文献标志码：** A **文章编号：** 1001-3695(2017)08-2319-05
doi:10.3969/j.issn.1001-3695.2017.08.019

Optimization of massive small files storage and accessing on HDFS

Gu Yuwan, Wang Wenwen, Sun Yuqiang†
(*School of Information Science & Engineering, Changzhou University, Changzhou Jiangsu* 213164, *China*)

Abstract: In order to solve the problem of NameNode memory bottleneck when HDFS stored a massive amount of small files, this paper proposed an optimization of massive small files storage and accessing on HDFS to improve the efficiency of HDFS. First, it could get the relationship between small files by analyzing a large number of history access logs, and then merged these correlative small files into a big file which would be stored on HDFS. When the client read data from HDFS, the system would prefetch the related files which were most likely to be visited next according to the relevance of small files to reduce the number of request for NameNode, thereby increasing the hit rate and processing speed. The results of experiment show that this method can effectively improve the efficiency of storing and accessing mass small files on HDFS, and cuts down the memory utilization of NameNode.

Key words: massive small files; relationship between files; merge; prefetch

随着信息技术的不断发展及大数据时代的来临，每天都会有大量数据产生，已经很难估算全球存储设备中总共有多少数据。据统计，在物理实验研究中，大型强子对撞机每年产生的数据约为 15 PB; Facebook 已存储超过 500 亿张照片，且每天分享约 20 亿张照片; 阿里巴巴 Galaxy 每日处理超过 2 500 亿条数据记录，日处理量近 2 PB。显然，大数据最大的特点就是数据量巨大，大到单机系统根本不可能解决存储和数据分析等问题。这就需要分布式系统为大数据提供支撑服务，而 Hadoop 正是这样一个平台。Hadoop 核心部分包括分布式文件系统 HDFS、并行计算模型 MapReduce 和分布式数据库 HBase[1]。在 Hadoop 中，HDFS 负责数据存储，它是谷歌 GFS 的一个 Java 开源实现，为分布式计算提供了底层支持。

HDFS 最初的设计目的是为了能够存储和分析大文件，在实际应用中也取得了非常好的效果。但在现实应用中，如电子商务、社交网站、科研计算，都存在着大量小文件。例如上文提到的 Facebook 上存储的海量照片，这些照片的大小通常在 10 KB ~ 10 MB，远小于 HDFS 中 block 块大小。HDFS 主从式的架构设计在极大简化系统结构的同时也带来了小文件存取效率低下的问题。原因主要有以下几点: a) 海量小文件的元数据信息都存储在 NameNode 中，造成 NameNode 节点的内存瓶颈问题; b) 读取大量小文件导致客户端频繁与 NameNode 节点进行通信，降低了元数据节点的 I/O 性能; c) 从 HDFS 中读取小文件，数据读取粒度小，且大量小文件存储空间连续性不足，难以发挥 HDFS 顺序式文件访问的优势。

1 相关工作

针对 HDFS 存取小文件效率低下的问题，目前国内外已有大量研究，主要分为以下两种思路:

a) 小文件合并。基于小文件合并的 HDFS 存储优化方法主要有以下两种:

(a) 基于 Hadoop archive(HAR)[2]、sequenceFile[3] 和 mapFile[4] 的小文件合并。HAR 利用层次化结构将小文件打包成一个归档文件(.har)，以缓解 NameNode 节点的内存压力。尽管如此，通过 HAR 读取一个小文件并不比直接从原 HDFS 中读取来得高效。事实上，每次访问一个 HAR 文件都要经过两层 index 索引，再加上读取文件本身，反而使读取文件的时间变得更长。HAR 的另一个缺点就是不支持文件的修改。Archive 文件一旦被创建就无法对其进行更改，这就意味着当需

收稿日期：2016-08-19; 修回日期：2016-09-26 基金项目：国家自然科学基金资助项目(11271057,61640211); 江苏省普通高校研究生科研创新计划项目(SCZ1412800004)

作者简介：顾玉宛(1982-)，女，讲师，博士，主要研究方向为软件工程、并行算法; 王文闻(1991-)，男，硕士研究生，主要研究方向为软件工程、并行算法; 孙玉强(1956-)，男(通信作者)，江苏常州人，教授，博士，主要研究方向为并行算法、软件工程、云计算(sunyuqiang0@126.com).

要变更文件时只能重新创建归档文件。可以将 sequenceFile 和 mapFile 看做是一个将小文件组织起来统一存储的容器。sequenceFile 以键值对的方式存储每条记录，通常文件名为 key、文件内容为 value。将小文件合并成大文件，同时还可以对记录或数据块进行压缩，降低系统存储空间消耗。但是由于没有建立索引，所以无法根据键值查找文件，读取每个小文件都必须遍历整个 sequenceFile 大文件。MapFile 是已经排过序的 sequenceFile，mapFile 有索引，可按键值检索文件，在这一点上 mapFile 效率明显高于 sequenceFile。从结构上看，由 index 和 data 两个文件构成，index 存储每个小文件的 key 值和偏移量，data 存储文件的内容。默认情况下每隔 128 个键才有一个键被存储在 index 文件中，因为只有部分文件可以被检索，所以 mapFile 随机读取文件的性能并不理想。

(b) 基于数据库的小文件合并。数据库分为关系型数据库(RDBMS)和非关系型数据库(NoSQL)。RDBMS 中小文件的合并大致有两种方法：第一种，通过 HDFS 提供的 append 方法将小文件合并，同时利用 RDBMS 为每一个小文件建立索引和记录偏移量，提高小文件的读取效率[5]；第二种，将 RDBMS 作为 HDFS 前端预处理模块，存入数据时，RDBMS 先将小文件合并，当数据量达到一定规模后，再存储至 HDFS，读取数据时，先查询 RDBMS，获得文件的存储位置，再从 HDFS 中读取数据[6]。非关系型数据库中，基于 HBase 的小文件合并方法主要受限于：存储文件的大小受 block 块的限制、逐渐增多的 HBase 合并与分解处理将占用大量系统资源。

b) 数据预调度。数据预取是提升 HDFS 小文件处理效率的一种重要方法，主要有以下两种策略：

(a) 文件之间的相关性。通过分析大量历史访问日志，获得文件之间的关联概率模型。执行预取时，将与该文件有较强关联的文件预先读取出来。例如文献[7]利用向量空间模型计算文本相似度：首先分析文本数据，以关键词权重为分量，得到文本对应的向量，再利用向量余弦公式计算相似度，值越大，说明两个文本越相似。但是这种方法只能说明两个文本相似度有多大，并不能说明文本之间的相关性有多大。也就是说，尽管文本 A 和 B 相似度很大，但并不能认为这两个文本在一段时间内都被访问的可能性很大。

(b) 局部性原理。这种方法的主要思想就是在数据读取时将相邻物理块的数据一同取出至缓存。例如文献[8]在访存失败时，将该数据所在文件全部读取出来存储在缓存中。这种方法在很大程度上发挥了大文件顺序访问的局部性优势，但不适用于海量小文件的随机访问。

基于上述研究基础，本文提出了一种基于小文件合并与预取的存取优化方法。首先，通过分析大量历史访问日志，得到小文件之间的关联关系，即哪些文件很有可能会在一段时间内都被读取。再将这些相关联的小文件合并成大文件存储到 HDFS，这样就使得关联小文件在存储空间上具有一定的连续性，为以后的预取操作提供了便利。在读取文件时，根据文件相关性大小，并综合考虑预取时间和访问延迟等因素，将符合条件的文件预先调度到客户端缓存，之后客户端便可直接从本地缓存中读取数据，避免了从 HDFS 中读取数据的时间开销。如此不仅可以降低元数据节点的内存占用率，而且减少了客户端与元数据节点的交互次数，提升了系统性能。整个系统主要由客户端服务器、小文件处理模块、预取模块三个部分组成，如图 1 所示。

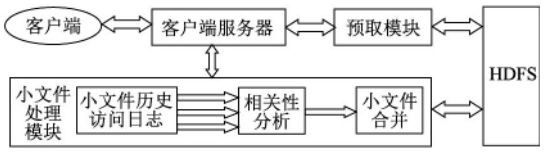

图1 文件系统结构

小文件处理模块：分析大量历史访问日志，获得小文件之间的相关性，再将相关联的小文件合并成大文件存储至 HDFS，同时建立索引文件并存储在客户端服务器中。

预取模块：当客户端访问某个小文件时，根据相关性，将与该文件相关的多个小文件预取出来。

2 基于文件相关性的小文件合并

2.1 文件相关性

人们总是出于某种目的或完成某个任务才会通过客户端向服务器发出请求，正因如此，很多被访问的文件之间才有了某些关联。例如同一篇新闻的多张照片很有可能都被访问；教育类的文件通常会被循序渐进地访问，因为只有学习了前面的知识，后面的知识才会被学习。很多文件之间都是有关联的，可以通过分析访问日志，获取文件之间的相关性，再将相关联的小文件合并成大文件存储至 HDFS。在读取文件时，将与该文件相关的文件一并取出。这样既降低了 NameNode 内存消耗，又提高了系统处理速度。

定义 1 $P(B|A)$。用户访问过文件 A 之后 T 时间范围内再访问文件 B 的概率，即

$$P(B|A)=\frac{N_{AB}}{N_A}\times 100\% \tag{1}$$

其中：N_A 表示文件 A 被访问的次数；N_{AB} 表示文件 A 和 B 都被访问的次数；$P(B|A)\in(0,1)$。

定义 2 $P(AB)$。一定时间范围内，文件 A 和 B 都被访问的概率，即

$$P(AB)=\frac{N_{AB}}{N}\times 100\% \tag{2}$$

其中：N 表示历史日志的总数，$P(AB)\in(0,1)$。

定义 3 $I(B|A)$。用于描述文件 A 被访问对 T 时间范围内文件 B 被访问的影响力，即

$$I(B|A)=\frac{N\times N_{AB}}{N_A\times N_B} \tag{3}$$

其中：N_B 表示文件 B 被访问的次数；$I(B|A)>0$，而且只有当 $I(B|A)>1$ 时才认为文件 A 被访问这一事件对文件 B 被访问具有促进作用，这时的相关性才有价值。

需要设定 $P(B|A)$、$P(AB)$、$I(B|A)$ 的最小阈值 $\min\_P(B|A)$、$\min\_P(AB)$、$\min\_I(B|A)$，则相关联的小文件之间需满足：

$$(\text{file}A,\text{file}B)=\left\{(\text{file}A,\text{file}B)\,|\,\begin{matrix}P(B|A)>\min\_P(B|A)\ \&\&\\ P(AB)>\min\_P(AB)\ \&\&\ (B|A)>\min\_I(B|A)\end{matrix}\right\}$$

由此可以看出，$P(B|A)$ 描述了在文件 A 被访问的条件下，文件 B 被访问的概率，是对文件相关性的准确性度量；$P(AB)$ 描述了文件 A、B 都被访问在所有访问日志中所占比重，是对文件相关性的代表性度量；$I(B|A)$ 描述了文件 A 被访问对文件 B 被访问有多大影响力，是对文件相关性的实用性度量。有时候虽然 $P(B|A)$ 很高，但是 $P(AB)$ 却很低，说明文件之间的相关性在实际应用中出现次数不多，即代表性不足，因此不具有实用价值。

2.2　文件相关性并行算法

由以上定义可以看出，计算文件相关性首先要统计每个文件的访问次数，即 N_A 和 N_B。本算法先获得满足 min_$P(B|A)$ 条件的关联文件集合，再计算满足 min_$P(AB)$ 和 min_$I(B|A)$ 条件的关联文件集合，最终得到全部关联小文件。

算法 1

输入： 历史访问日志数据库 DB，日志记录总数 N，min_$P(B|A)$、min_$P(AB)$ 和 min_$I(B|A)$。

输出： 关联文件集合 L_2。

```
for all records r ∈ DB do
for each item i ∈ r do
  i. count ++
L1 = { ik | ik • count / N ≥ min_P(B|A) }
for each itemset i1 ∈ L1 do
  for each itemset i2 ∈ L1 do
  if i1 [0] = i2 [0] then
    C = (i1, i2)
    for each subset s ∈ C
      if L1 contain s then
        add C to C2
      else remove C
for all records r ∈ DB do
  for each mid_temp mt ∈ subset(C2, r)
      mt. count ++
L2 = { mt ∈ C2 | mt. count / N ≥ min_P(B|A) }
for each correlation cr ∈ L2 do
  for all records r ∈ DB do
    if r contain cr. first
      NA ++
    if r contain cr. second
      NB ++
P(AB) = NAB / N = cr. count × NA / N^2
I(B|A) = N × NAB / (NA × NB) = cr. count / NB
if P(AB) ≤ min_P(AB) or I(B|A) ≤ min_I(B|A) then
    remove cr
```

2.3　小文件合并算法

通过算法 1 得到关联小文件集合，但这样的关联关系只限于两个小文件之间，比如有两个关联关系（file$A$，file$B$）和（file$B$，file$C$），不可能简单地将 file$B$ 与 file$A$、file$B$ 与 file$C$ 分别合并存储。一方面是因为合并后的文件大小依然远小于 block 块，另一方面是因为 file$B$ 会被重复存储，浪费了存储空间。所以在合并小文件之前还要做一些准备工作。

2.3.1　关联关系合并

定义 4　在关联关系（file$A$，file$X$，…，file$Y$，file$B$）中，将 file$A$ 称为前驱（former），file$B$ 称为后继（latter）。

假定当前读取的小文件是 file$A$，在关联文件集合 L_2 中有多个关系包含 file$A$，如果不对关联文件集合 L_2 进行处理，那么在执行预取时就必须判定：要预取哪些与 file$A$ 有关联的文件或者是否预取全部与 file$A$ 有关联的文件？但不管是全部预取还是部分预取，都要遍历集合 L_2。当 L_2 较小时，遍历 L_2 耗费的时间尚可接受；但当 HDFS 存储了海量小文件时，L_2 变得很大，遍历 L_2 将浪费大量时间。所以，将 L_2 中两两之间的关联关系合并成一个大的关联集合不但有利于节省存储空间，而且还可以简化预取操作。具体步骤如下：

a）若关联文件集合 L_2 为空，合并完成，返回结果；否则，转步骤 b）。

b）在所有日志记录中选定一个小文件，这个小文件作为关联关系的前驱出现的次数最多。

c）从所有包含此文件的关联关系中选取关联概率最大的作为基准（首个关联关系）。

d）依次将其余关联关系中前驱与此关联关系后继相同的关联关系合并在一起，若有多个关联关系满足条件，则选取关联概率最大的合并。每次合并一个关联关系就将它从 L_2 中删除。

e）重复步骤 c），若找不到合适的关联关系，转步骤 f）。若合并后文件大小大于 block 块，以这个不能被合并的关联关系为基准，转步骤 d）。

f）从 L_2 中删除此新的关联关系，并返回步骤 a），开始下一次合并。一趟合并过程如图 2 所示，主要算法如算法 2 所示。

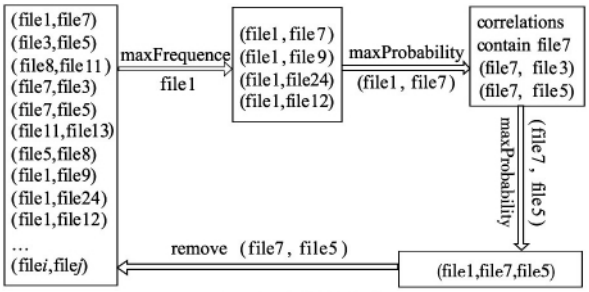

图2　合并关联关系

算法 2

输入： 关联文件集合 L_2。

输出： 合并关联关系后的关联文件集合 L_2'。

```
combine(correlation, L2) {
  if(L2 = null) { return L2'; }
if(correlation = null) {
    选取第一个文件 firstFile，并产生候选关系集合 candidateCorrelations;
    选取第一个待合并关联关系 firstCorrelation;
    latter = firstCorrelation. latter;
    candidates = {all correlations Except firstCorrelation};
    选取要合并的关联关系 nextCorrelation;
    if((firstCorrelation + nextCorrelation) <64 MB) {
    firstCorrelation. combine(nextCorrelation. latter);
    remove nextCorrealtion from L2;
    combine(firstCorrelation, L2);
    }
} //第一趟合并结束
else{
    firstCorrelation = correlation;
    latter = firstCorrelation. latter;
    candidates = {all correlations Except firstCorrelation};
    //若没有找到可合并的关联关系
    if(candidate = null) {
    L2'. add(firstCorrelation);
    remove firstCorrelation from L2;
    combine(null, L2); //开始下一次合并
    }
    选取下一个待合并关联关系 nextCorrelation;
    if((firstCorrelation + nextCorrelation) < 64MB) {
    firstCorrelation. combine(nextCorrelation);
    remove nextCorrealtion from L2;
    combine(firstCorrelation, L2);
    }
    else{
      L2'. add(firstCorrelation);
```

```
        remove firstCorrelation from L2';
        combine( nextCorrelation,L2') ; }
    }
  }
```

2.3.2 小文件合并

确定要合并的小文件之后,遍历集合 L_2',将每个关联关系中的所有小文件合并成大文件存储至 HDFS。主要算法如下:

算法 3

输入: 关联文件集合 L_2'。
输出: store in HDFS。

```
amalgamate( L2') {
  for correlation in L2'{
    for file in correlation{
      mergedFile = Block. mergeFile( file) ;
    }
    store mergedFile in HDFS;
  }
}
```

2.3.3 索引文件

首先为所有大文件创建一个索引文件,用于记录每个大文件所在 block 信息。合并时为每一个大文件创建一个局部索引文件,用于记录小文件到大文件的映射关系以及小文件的长度和偏移量。将索引文件存储于客户端服务器中,这样每次读取小文件时可直接查询索引文件获得小文件所在的大文件、小文件长度和在大文件中的偏移量等信息,而无须与 NameNode 进行交互,减少了 I/O 请求,提升了文件访问效率[9,10]。

3 文件预取

3.1 预取条件

当客户端访问某个小文件时,考虑到预取时间、用户等待时间等因素,不可能将与该文件相关的所有数据都预取出来,所以必须确定到底要预取多少数据才能做到既执行了预取又不延长本次访问时间。

定义 5 $\text{time}_{\text{HDFS}}$,表示在没有预取的情况下,从客户端发出访问请求到获得 HDFS 返回数据的时间。

定义 6 $\text{tims}_{\text{prefetch}}$,表示预取一个文件所耗费的时间。

定义 7 $\text{time}_{\text{cache}}$,表示缓存命中,客户端直接从缓存读取文件数据的时间。

定义 8 $\text{time}_{\text{wait}}$,表示用户最大等待时间。

那么,访问一个小文件并预取多个关联小文件的时间为

$$\text{time}_{\text{HDFS}} + \sum_{i=1}^{\text{count}} (\text{time}_{\text{prefetch}} \times P_i)$$

访问这 count + 1 个小文件的时间为

$$\text{cost}_{\text{prefetch}} = \text{time}_{\text{HDFS}} + \sum_{i=1}^{\text{count}} (\text{time}_{\text{prefetch}} + \text{time}_{\text{cache}}) \times P_i$$

在没有预取的情况下,访问 count + 1 个小文件的时间为

$$\text{cost} = \text{time}_{\text{HDFS}} \times (\text{count} + 1)$$

显然,$\text{cost}_{\text{prefetch}} < \text{cost}$。

虽然预取总会为下一次访问带来便利,但当预取的文件数量太多会导致本次请求时间延长,影响用户体验。所以在考虑 $\text{time}_{\text{wait}}$ 的情况下,预取文件的数量 count 应满足

$$\text{time}_{\text{HDFS}} + \sum_{i=1}^{\text{count}} (\text{time}_{\text{prefetch}}) \times P_i) < \text{time}_{\text{wait}}$$

化简为

$$\sum_{i=1}^{\text{count}} P_i < \frac{\text{time}_{\text{wait}} - \text{time}_{\text{HDFS}}}{\text{time}_{\text{prefetch}}} \quad (4)$$

即执行预取时将关联小文件的概率依次代入式(4),直到条件不成立。

3.2 预取算法

算法 4

输入: file。
输出: files。

```
prefetchFile( file) {
  list files = null;
  if( file in cache)
    return;
  else{
    for file in files
      P + = file. getProbability( ) ;
      if( P < (time_wait - time_HDFS) / time_prefetch)
        files. add( file) ;
  }
  return files;
}
```

4 实验与分析

4.1 实验环境

本实验环境基于五个节点的 Hadoop 集群,其中 NameNode 节点 16 GB 内存,处理器 Intel i5 CPU 2.40 GHz,1 TB 硬盘; DataNode 节点 4 GB 内存,处理器 Intel Pentium 2.40 GHz,500 GB 硬盘。操作系统 Ubuntu 12.04,Hadoop 版本 2.2.0,JDK 版本 1.6,HDFS 文件块大小默认 64 MB,副本数三个,网络环境为千兆以太网。

4.2 实验对比

本文设计了三组对比实验,评价指标分别为文件平均存储时间、平均读取时间和 NameNode 内存占用率。每组实验依次处理 10 000、20 000、30 000 和 40 000 个小文件。每组实验中,将本文提出的优化方法与原 HDFS、HAR 进行对比。

4.2.1 文件相关性实验

因为本文提出的小文件合并方法是基于历史访问日志,所以在进行本方法的实验之前需要先通过学习才能建立文件关联模型,进而分别合并 10 000、20 000、30 000 和 40 000 个小文件存储至 HDFS。该实验可借助 SPSS Clementine 进行。通过试错法,设定 $\min\_P(B|A)$ 为 30%,$\min\_P(AB)$ 为 50%。获得文件两两之间的相关性之后,再用 $\min\_I(B|A)$ 验证此相关性是否有实际利用价值,即验证 $\min\_P(B|A) > 1$。由于文件数量较大,无法枚举所有文件之间的相关性数据,所以只列出部分,如表 1 所示。

表 1 文件相关性部分实验数据

former	latter	$P(B\|A)$/%	$P(AB)$/%	former	latter	$P(B\|A)$/%	$P(AB)$/%
file1	file3	33.48	61.88	file7	file13	52.07	67.82
file1	file7	32.67	57.23	file1	file6	53.21	68.91
file3	file1	51.62	61.88	file3	file7	57.82	73.52
file7	file12	27.13	68.56	file2	file4	35.57	48.15
file5	file15	41.02	65.71	file8	file4	43.74	67.37
file5	file17	48.61	69.13	file12	file7	47.86	68.56

从表 1 中可看出,客户在访问过 file1 后再访问 file3 的可能性为 33.48%,且两个文件被同时访问的概率为61.88%,可以认为这两个文件应该被合并在一起; 而 file2 和 file4 被同时访问的概率为 48.15,低于设定的 $\min\_P(AB)$,故该关联关系也是无效; 同样,客户在访问过 file7 再访问 file12 的可能性只

有 37.13%，低于设定的 min_$P(B|A)$，故该关联关系无效。但是反过来，客户在访问过 file12 再访问 file7 的概率达47.86%，说明客户通常是在访问完 file12 后才会访问 file7。这样的情况在教育资源的访问中经常出现：file12 是 file7 的前置课程，学生总是先学习了 file12 再学习 file7。

4.2.2　文件平均存储时间

文件平均存储时间为

$$\text{FAST}(\text{FileAverag eStoreTime}) = \frac{\sum_{i=1}^{\text{total}} \text{time}_i}{\text{total}}$$

其中：time_i 表示每个小文件存储所耗费的时间；total 表示存储的文件总数；FAST 值的大小表征了系统存储小文件的效率，FAST 值越大，说明效率越高。实验结果如图 3 所示。图中横坐标代表小文件数量，纵坐标代表平均存储时间 FAST。

由图 3 可以看出，随着小文件数量的增多，原 HDFS 和 HAR 平均存储时间明显增加，而优化后 HDFS 的 FAST 值变化幅度比较平缓，主要原因是基于相关性将小文件合并存储，在很大程度上减少了与 NameNode 节点的交互次数，降低了 I/O 占用率，提升了系统写入速度。观察图 3 可以发现，在文件存储数量较少时，优化后的 HFDS 存储性能并没有提高，相反甚至比原 HDFS 耗费了更多的时间。这是因为优化后的 HDFS 在存储海量小文件之前要进行历史访问日志分析和小文件合并两个操作。但由于原 HDFS 的 NameNode 节点需要维护所有小文件的元数据信息，随着文件数量的增加，NameNode 节点的内存占用率急剧上升，系统性能随之下降，而优化后的 HDFS 将众多小文件合并成一个大文件再存储到 HDFS 中，所以 NameNode 节点的内存消耗并不会增加太多。

4.2.3　文件平均读取时间

文件平均读取时间为

$$\text{FART}(\text{FileAverageReadTime}) = \frac{\sum_{i=1}^{\text{total}} \text{time}_i}{\text{total}}$$

其中：time_i 表示读取一个文件所用的时间；total 表示读取的文件总数；FART 值的大小表征了系统读取小文件的效率，FART 值越大，说明效率越高。实验结果如图 4 所示。图中横坐标代表小文件数量，纵坐标代表平均读取时间 FART。

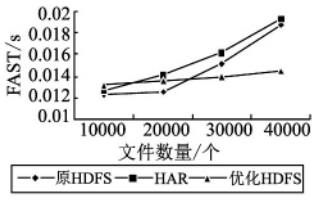

图3　小文件平均存储时间

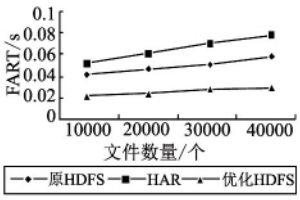

图4　文件平均读取时间

由图 4 可以看出，优化后 HDFS 的性能在小文件读取方面明显优于原 HDFS 和 HAR。随着小文件数量的增多，原 HDFS 和 HAR 系统性能下降，读取文件的用时显著增加，而优化后 HDFS 的 FART 值变化幅度却很平缓。其主要原因是基于预取的小文件读取策略，在访问某一文件时将与该文件有关联的多个文件预取缓存，不仅减少了客户端与 NameNode 节点的交互，而且加快了数据访问的速度。正如前文所述，HAR 虽然为 HDFS 海量小文件存取提供了解决方案，但两层索引结构大大降低了其文件读取性能，效率甚至比原 HDFS 更差。

4.2.4　NameNode 内存占用率

分别将 10 000、20 000、30 000 和 40 000 个小文件依次存储到原 HDFS、HAR 和优化后的 HDFS 中，对 NamNode 内存占用率进行测试，如图 5 所示。

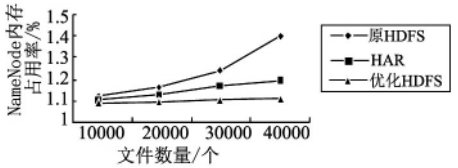

图5　NameNode内存占用率

由图 5 可以看出，与原 HDFS 和 HAR 相比，优化后的 HDFS 能有效降低 NameNode 节点的内存占用率。由于 HAR 将小文件进行打包归档，所以与原 HDFS 系统相比，NameNode 节点内存占用率得以降低。当小文件数量较少时，三种方案的 NameNode 内存占用率相差无几，但随着小文件数量逐步增多，原 HDFS 和 HAR 的 NamNode 节点内存占用率不断升高，而优化后的 HDFS 内存占用率增幅较小，尤其当文件数量很大时，效果更明显。其主要原因是优化后的 HDFS 将小文件合并存储，NameNode 节点元数据信息量减少，内存占用率大幅度降低。

5　结束语

本文提出了一种面向 HDFS 海量小文件存取的优化方法，主要包括小文件相关性分析、合并和预取三个部分。通过分析大量历史访问日志获得小文件之间的关联关系，再将相关联的小文件合并成大文件存储到 HDFS。在读取文件时，根据文件相关性，并兼顾到用户最大等待时间等因素，将与该文件相关的多个小文件预取缓存。实验结果表明，该方法能有效解决 NameNode 节点的内存瓶颈问题，提高 HDFS 处理海量小文件的效率。该方法的缺点是：a) 额外增加小文件历史访问日志分析、小文件合并和预取三个模块；b) 在合并关联关系时，尚没有找到一种方法证明合并后的关联关系是最优解。所以，关于 HDFS 存取海量小文件的问题仍需以后更深入的研究。

参考文献：

[1] Tom W. Hadoop 权威指南 [M]. 北京：清华大学出版社，2010.

[2] Hadoop archives [EB/OL]. http://hadoop.apache.org/common/docs/current/hadoop_archives.html.

[3] Sequence file Wiki [EB/OL]. http://wiki.apache.org/hadoop/SequenceFile.

[4] MapFile [EB/OL]. http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/MapFile.html.

[5] 张海，马建红. 基于 HDFS 的小文件存储与读取优化策略 [J]. 计算机系统应用，2014，23(5)：167-171.

[6] 刘小俊，徐正全，潘少明. 一种结合 RDBMS 和 Hadoop 的海量小文件存储方法 [J]. 武汉大学学报：信息科学版，2013，38(1)：113-115.

[7] 游小容，曹晟. 海量教育资源中小文件的存储研究 [J]. 计算机科学，2015，42(10)：76-80.

[8] 黄启峰，郑纬民，沈美明. 一种机群文件系统的缓存模型 [J]. 小型微型计算机系统，2003，24(10)：1748-1752.

[9] Konstantin S，Hairing K，Sanyjy R，*et al*. The Hadoop distributed file system [C]//Proc of the 26th Symposium on Mass Storage Systems and Technologies. 2010：1-10.

[10] Chandrasekar S，Dakshinamurthy R，Seshakumar P G，*et al*. A novel indexing scheme for efficient handing of small files in Hadoop distributed file system [C]//Proc of International Conference on Computer Communication and Informatics. Piscataway：IEEE Press，2013：1-8.