蚂蚁绊倒大象？不起眼的小文件竟拖了Hadoop大佬的后腿

在使用Hadoop过程中，小文件是一种比较常见的挑战，如果不小心处理，可能会带来一系列的问题。HDFS是为了存储和处理大数据集（M以上）而开发的，大量小文件会导致Namenode内存利用率和RPC调用效率低下，block扫描吞吐量下降，应用层性能降低。通过本文，我们将定义小文件存储的问题，并探讨如何对小文件进行治理。

什么是小文件

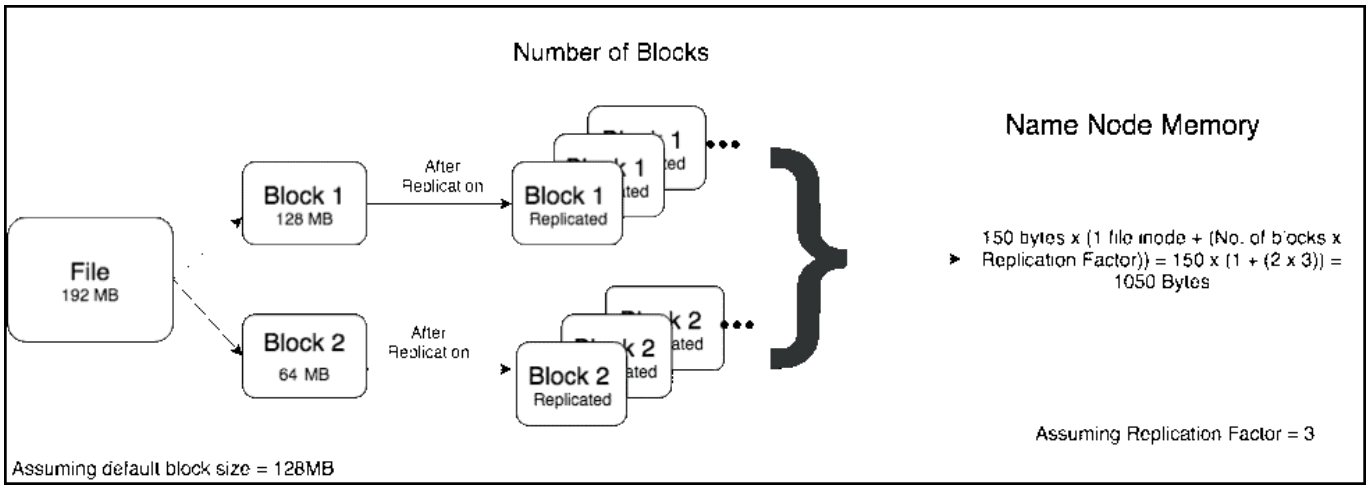
小文件是指比HDFS默认的block大小（默认配置为128MB，网易大数据集群配置为256M）明显小的文件。需要注意的是，在HDFS上有一些小文件是不可避免的。这些文件如库jars、XML配置文件、临时暂存文件等。但当小文件变的大量，以致集群中小文件成为主流，此时就需要对小文件进行治理，治理的目标是让文件大小尽可能接近HDFS block大小的倍数。

Hadoop的存储层和应用层的设计并不是为了在大量小文件的情况下高效运行。在说到这个问题的意义之前，我们先来回顾一下HDFS是如何存储文件的。

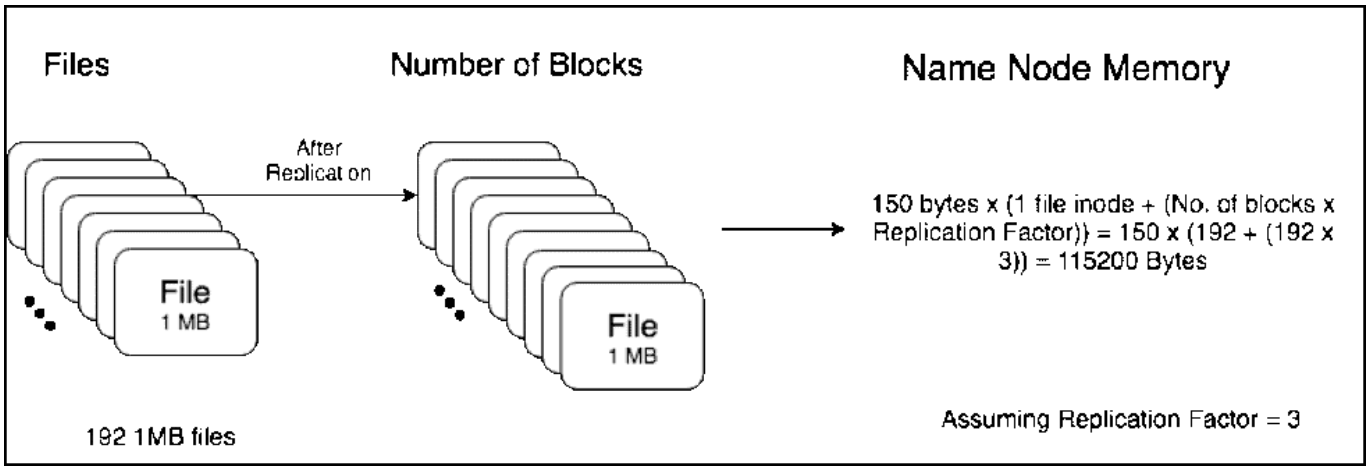
在HDFS中，数据和元数据是独立的实体。文件被分割成block，这些块被存储在DataNode的本地文件系统中，并在整个集群中复制。HDFS 命名空间树和相关的元数据作为对象保存在 NameNode 的内存中（并备份到磁盘上），每个对象一般占用大约 150 个字节。

下面的两个方案说明了小文件的问题。

方案1（1个192M的大文件）



方案2（192个小文件，每个1M的小文件）。



方案1有一个192MB的文件，分解成2个大小为128MB和64MB的块。复制后，存储一个文件的元数据所需的总内存 = 150字节 x (1个文件的inode + (block数 x 副本数量[通常副本为3]))。

按照这样计算，在Namenode上存储这个文件的元数据所需的总内存 = 150 x (1 + (2 x 3))=1050 Bytes。

相比之下，方案2有192个1MB的文件，然后这些文件在集群中复制。Namenode存储这些文件的元数据所需的总内存 = 150 x (192 + (192 x 3)) = 115200 Bytes。

因此我们可以看到，相对于一个192MB的大文件，在Namenode堆上需要100倍以上的内存来存储多个小文件。

对存储层的影响

当NameNode重启时，它必须将文件系统元数据从本地磁盘加载到内存中。这意味着，如果NameNode的元数据很大，重启速度会非常慢。NameNode还必须跟踪集群上的block位置的变化，太多的小文件也会导致NameNode在DataNode耗尽磁盘上的数据空间之前，就先耗尽内存中的元数据空间。DataNode还会通过网络向NameNode报告块的变化；更多的block意味着要通过网络发送更多的元数据变更。

更多的文件意味着更多的读取请求需要请求NameNode，这可能最终会堵塞NameNode的容量，增加RPC队列和处理延迟，进而导致性能和响应能力下降。按照经验，RPC工作负载接近40K~50K的RPCs/s是比较高的。

对应用层的影响

一般来说，在通过Impala这样的Ad HOC SQL引擎或MapReduce或Spark这样的应用框架运行计算时，拥有大量的小文件会产生更多的磁盘请求。

MapReduce/Spark

在Hadoop中，block是可以进行计算的最细粒度的数据单位。因此，它影响着一个应用的吞吐量。在MapReduce中，每读取一个block都需要1个Map Container。因此，小文件会降低性能，增加应用开销，因为每个任务都需要自己的JVM进程。

对于Spark来说，小文件也是类似的，在Spark中，每个“map”相当于Spark任务在执行器中每次读取和处理一个分区。每个分区默认情况下是一个block。这意味着，如果你有很多小文件，每个文件都在不同的分区中读取，这将导致大量的任务开销。

另外，MapReduce作业也会创建空间文件，如_SUCCESS和_FAILURE，用于标记MapReduce任务的finish状态。这些文件仍然会在Namenode中注册为一个 inode item，如前文所述，每个 使用 150 个字节。清除这些文件的一个简单有效的方法是使用下面的HDFS命令：

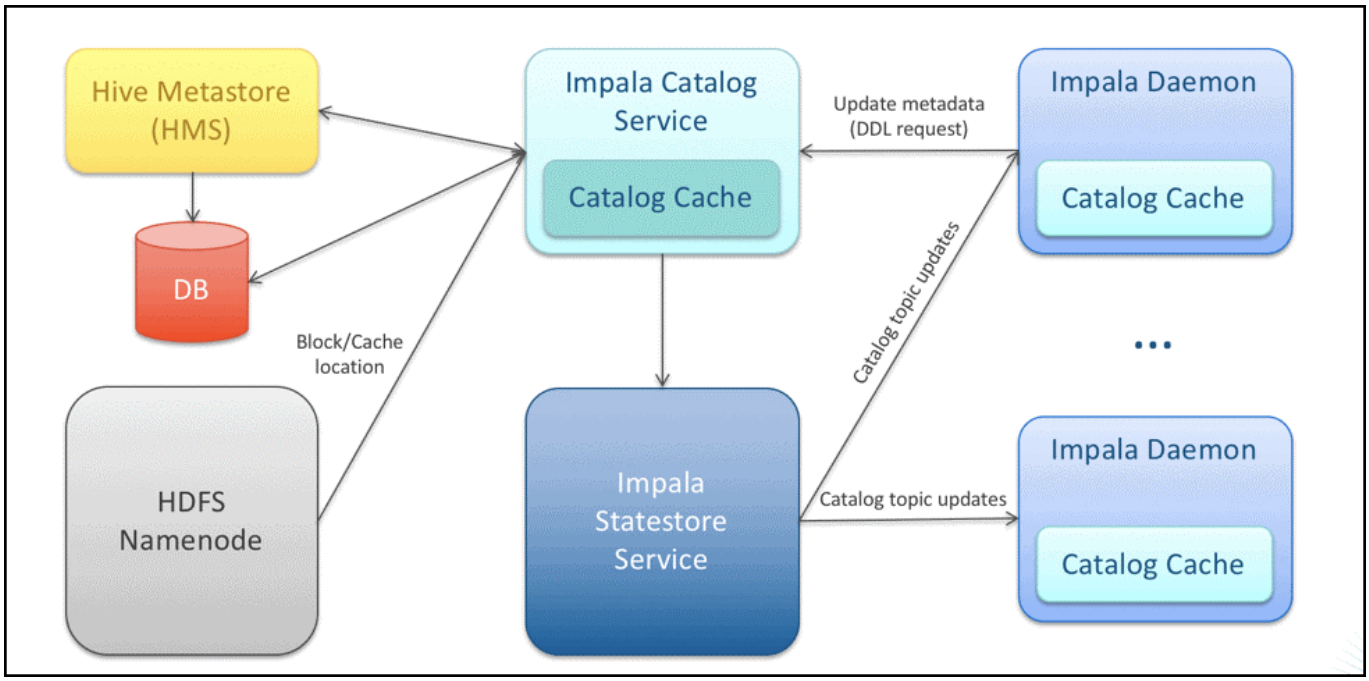
```
hdfs dfs -ls -R | awk '$1 !~ /^d/ && $5 == "0" { print $8 }' | xargs -n100 hdfs dfs –rm
```

这个命令将把这些文件移动到.Trash位置（HDFS回收站开启的情况下），一旦Trash清理策略生效，这些文件也将随之删除。

注意：如果有应用程序对这些文件有依赖性，删除这些文件可能会导致应用程序失败。

Impala-对catalog的影响

Impala是一个AD HOC引擎，它将HDFS namespace信息缓存在服务中，以实现更快速的元数据访问。下面是一个架构图，详细介绍了Impala如何缓存HDFS元数据。



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

与namenode管理HDFS文件元数据类似，Impala需要在Catalog中也维护一份元数据。下表描述了这些元数据及其估计的平均内存使用量。

对象的内存使用量

对象	内存用量
Table	5KB
Partition	2KB
Column	100B
Incremental Stats	400B* (per column per partition)
File	750B
File Block	300B

最高可以预估1.4KB/列/分区

例如：如果有1000个表，每个分区有200个表，每个分区有10个文件，那么Impala catalog的大小至少是：（不包括表统计信息和表列信息）。

#tables * 5KB + #partitions * 2KB + #files * 750B + #file_blocks * 300B = 5MB + 400MB + 1.5GB + 600MB = ~ 2.5GB

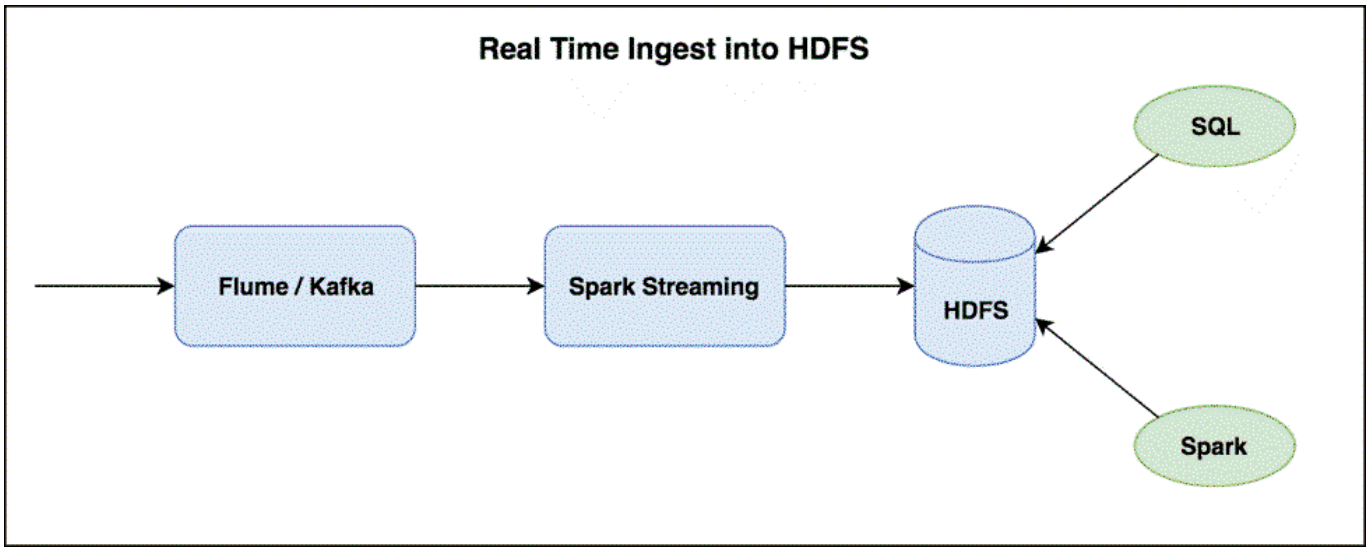
Impala目录大小越大，内存占用就越大。不建议在HMS的Hive/Impala中使用大的元数据，因为它需要跟踪更多的文件，会导致：

- 更长的元数据加载时间
- 更长的StateStore topic更新时间
- DDL语句操作缓慢
- 更长的查询计划分配时间

小文件是如何产生的

流式数据处理（spark streaming/flink等流式计算框架）

流式或者bacth的数据计算，最终可能会一段时间内产生大量的小文件。对于流式数据的近乎实时的要求，小的timewindow（每隔几分钟或几个小时）数量较小，就会生产很多小文件。比如下图的流式处理模式：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

拥有大量map/reduce的任务

MapReduce任务，如果有大量的map和reduce task，在HDFS上生成的文件基本上与map数量（对于Map-Only作业）或reduce数量（对于MapReduce作业）成正比。大量的reducer没有足够的数据被写到HDFS上，会把结果集稀释成很小的文件，因为每个reducer只写一个文件。按照同样的思路，数据倾斜也会产生类似的效果，即大部分数据被路由到一个或几个reduce，让其他的reduce写的数据很少，导致文件变小。

过度分区表

过度分区表是指每个分区的数据量很小（<256 MB）的Hive表。Hive Metastore Server (HMS) API 调用开销会随着表拥有的分区数量而增加。这反过来会导致性能下降。在这种情况下，应该考虑表的分区设计并减少分区粒度。

Spark过度并行化

在Spark作业中，根据写任务中提到的分区数量，每个分区会写一个新文件。这类似于MapReduce框架中的每个reduce任务都会创建一个新文件。Spark分区越多，写入的文件就越多。控制分区的数量来减少小文件的生成。

文件格式和压缩

出于小文件治理的目的，我们更推荐使用非TexFile的序列化存储方法。

识别出小文件

FSImage和fsck

因为NameNode存储了所有与文件相关的元数据，所以它将整个命名空间保存在内存中，而fsim

age是NameNode的本地本机文件系统中的持久化记录。因此，我们可以通过分析fsimage来找出文件的元信息。fsimage中可用的字段有：

Path, Replication, ModificationTime, AccessTime, PreferredBlockSize, BlocksCount, FileSize, NSQUOTA, DSQUOTA, Permission, UserName, GroupName

通常可以采用以下方法来解析fsimage

拷贝Namenode数据目录下的fsimage文件到其他目录，然后执行：

hdfs oiv -p Delimited -delimiter "|" -t /tmp/tmpdir/ -i fsimage_copy -o fsimage.out

关于hdfs oiv命令的使用，可以查看useage。

另一种方法是使用 fsck命令扫描当前的HDFS目录并保存扫描后的信息。

注意：在大型集群中，考虑生产环境的稳定性，不建议使用fsck命令，因为它会带来额外的开销。

如何处理小文件

提前规避

1.流式写入

调整流式写入的时间窗口是一个不错的选择，如果业务对实时性要求很高，那么可以根据数据类型（非结构化vs结构化）、append/update频率和数据使用模式（随机读取vs聚合），HBase和Kudu是存储层的更好选择。对于已经存在的小文件，也可以设置定期的Job对这些文件进行压缩、合并，以减少文件量和文件数量。

2.过度分区表

在决定分区的粒度时，要考虑到每个分区的数据量。为有大文件的分区做计划（用Parquet的话，约256MB或更大），即使这意味着有较少的粒度分区，例如每月而不是每天的分区。对于数据量小的表（几百MB），可以考虑创建一个非分区表。

3.Spark过度并行化

在Spark中向HDFS写入数据时，在向磁盘写入数据前要重新分区或聚合分区。这些语句中定义的

分区数量将决定输出文件的数量。强烈建议检查Spark作业的输出，并验证创建的文件数量和实现的吞吐量。

4.使用工具进行压缩

hadoop本身提供merge命令，当然用户也可以自行编写工具实现。

5.使用Hive对数据进行压缩

如果你有一个现有的Hive表有大量的小文件，那么可以通过以下设置来重写这个表（parquet格式）。关于Hive压缩可以查阅其他文档获取更详细的信息。

```
set hive.exec.compress.output=true;
set parquet.compression=snappy;
set hive.merge.mapfiles=true;
set hive.merge.mapredfiles=true;
set hive.merge.mapredfiles=true;
set hive.merge.smallfiles.avgsize=134217728; --128M
set hive.merge.size.per.task = 268435456; --256M
set hive.optiming.sort.dynamic.partition = true;
set parquet.blocksize= 268435456; --256M
set dfs.block.size=268435456; --256M
```

关于Hive配置属性的详细信息可以在Apache
Hive官方页面上找到，这里再提供一些重新数据的方法。

CREATE TABLE AS SELECT（CTAS）对于一个非分区表会比较方便。

你也可以先运行CREATE TABLE LIKE (CTL)来复制一个表结构，然后使用INSERT OVERWRITE SELECT语句将数据从源表加载数据到目标表。

注意：如果在没有定义静态分区名的情况下插入数据，需要在Hive中启用非严格的动态分区模式，可以通过设置

```
hive.exec.dynamic.partition.mode=non-strict
```

分区列必须是选择语句中的最后一列，这样动态分区才能在这种情况下工作。此外，也可以直接使用mapred.reduce.tasks设置来配置reduce的数量。创建的文件数量将等于使用的减速器数量。设置一个最佳的减速器值取决于写入的数据量。

总结

提前规避要好于事后补救，在任务开发以及表设计的前期尽可能考虑小文件问题尤为重要，因为事后修改任务或者修改表带来业务失败的概率会大得多。此外，小文件治理也是一个长期的过程，对于一个生产集群，定期的进行小文件治理是必要的。

网易数据资产中心将小文件数量或比例转化成指数关联到库，表，目录上。用户可以根据库，表，目录等信息发现小文件产生的任务，对小文件的产生进行追本溯源，然后通过调整任务参数等手段从源头进行治理。

追本溯源可以从源头切断小文件的产生，但是这个是比较理想化的情况，现实中存在很多场景无法在源头进行调整。网易数据资产中心也提供了定期触发的小文件合并策略，在策略识别到小文件过多的表或者目录上进行小文件合并。对于已经产生了很多小文件的表或目录提供主动合并的手段将小文件进行合并。

本文原文：蚂蚁绊倒大象？不起眼的小文件竟拖了Hadoop大佬的后腿