NativeTask与Hive兼容

NativeTask是Hadoop MapReduce到的高效执行引擎实现，其支持不排序的数据流（这也是性能提升的一部分），经典的MapReduce数据流式需要对记录进行排序的。Hive执行程序中，很多操作都需要对数据进行排序，因此Hive运行在NativeTask中会抛异常。下面分析MapReduce排序操作及NativeTask对排序的支持及改进。

# 1、经典MR排序

MapReduce的执行流程如下图所示：



Map的执行结果先放到内存中，内存不够时，通过spill过程持久化到硬盘中，过程如下：partition->sort->spill。

排序1：在spill到磁盘之前，首先要进行排序操作。

代码执行过程分析如下所示：



Job.setOuputKeyValueClass，决定了Comparator的类，以WordCount为例，输出类为IntWritable，其继承了WritableComparable类，因此在MapOutoutCollector初始化时，Comparator设置为WritableCompartor，其实现了接口方法compare，该comparator在排序时作为比较器。

MapOutputBuffer，被排序的数据结构为kvbuffer:byte[]，该缓冲区大小由mapreduce.task.

io.sort.mb配置。

排序2：map执行结束时，spill到磁盘的文件数一般为多个(sortAndSpill中numSpills++)，Map结束时，需要对Spill的文件进行merge操作，在merge的过程中使用归并算法进行排序。

代码执行过程分析如下所示：



Merger在merge的过程中会使用comparator，通过JobConf.getOutputKeyComparator获取，该方法返回参数为RawComparator。merge的过程是文件流的merge过程，实质上进行归并排序。

排序3:

Map结束后，Reduce启动shuffle阶段，从Map输出文件中获取数据，但是Map一般多个，某个Reduce从多个Map输出文件中获取其所对应的处理数据，这些数据需要merge（同时进行排序）。Reduce分为copy->sort->reduce三个阶段，过程如下所示：



ReduceTask启动Shuffle线程，并初始化MergerManager（用于数据的merge），初始化LocalFetcher，LocalFetcher根据reduceId生成文件位置（用于从Map输出文件读取对应数据），读取数据放入MergerManager的mapOutput中。最后MergerManager对这些数据进行merge操作（也就是sort阶段，归并排序），Merger首先使用ImMermoryMapOut，当内存不够时，使用内存到磁盘的merge操作。最后进行Reduce阶段的数据处理。

总结：在经典MR中，有三个地方涉及到排序操作，首先是Map任务处理时中间结果先放到内存中，内存空间不足需要spill到磁盘中，在spill之前需要排序。第二次排序，在Map任务结束时，需要spill的文件数据较多，因此要进行一次文件归并操作，在这个过程中进行归并排序。第三次排序，再reduce，经过shuffle阶段，从各个Map任务输出文件中获取所需要的数据，由于Map数较多，获取的数据进行一次merge处理，便于下一步的操作。

# 2、NativeTask MR排序

NativeTask改进在MapReduce任务级别能够影响到的方面，具体包括压缩、排序、序列化以及部分shuffle。NativeTask避免序列化和数据拷贝，为了达到最大的吞吐率，在主要数据流中不使用序列化，基本接口为传递内存数组引用的形式，尽量避免数据拷贝。NativeTask增加支持不排序的数据流，不排序的数据流在聚合类应用开销要比保持所有键值对排序的开销要小的多。

NativeTask的核心是增加Map output collector的本地实现，用户通过配置mapreduce.job.

map.output.collector.class来启用，对于shuffle密集型的任务会提高30%以上的性能。基本的思想是增加NativeMapOutputCollector来处理mapper生成的键值对，包括sort,spill（内存数据过多，溢出到磁盘的过程）及IFile的序列化。

NativeMapOutputCollector，使用JNI将键值对及partition value传递到本地端，由于调用JNI有前期开销，因此增加DirectBuffer作为K/V缓存，然后以批方式传递。在本地端，K/V paris放置在Partitioned buffers，sort过程会加快，因为这个排序的数据比较少。两个轻量级的io buffer:ReadBuffer及AppendBuffer 比decorator based java&hadoop io streams，采用轻量级buffer对IFile serialization有好处，下面分析MR执行过程。

类图如下所示：



步骤1：

Map执行的collect处理涉及的类及方法如下：

NativeMapOutputCollectorDelegator#collect->NativeCollectorOnlyHandler#collect ->

BufferPusher#collect->ByteBufferDataWriter#write- >ByteBuffer.write

步骤2：

数据先写到ByteBuffer（NativeBatchProcess.rawOutputBuffer）中，当缓存中的数据写满时，调用ByteBufferDataWrite#flush操作，将数据传到NativeBatchProcessor中进行处理，进行的操作为NativeDataTarget#sendData做进一步的数据处理，过程如下：

NativeBatchProcessor#sendData->NativeBatchProcessor#nativeProcessInput，该方法通过JNI调用C++实现，这也是NativeTask的改进的核心地方。

步骤3：

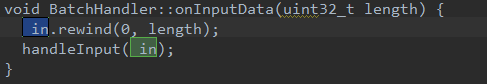
NativeBatchProcessor通过JNI调用BatchHandler.cc中同名方法，其中提供的方法包括：

|  |  |
| --- | --- |
| 方法名 | 描述 |
| setupHandler | 初始化native端的BatchHandler |
| nativeProcessInput | Native side 处理inputBuffer中的数据 |
| nativeFinish | Nativide input过程结束 |
| nativeCommand | 向native side发送control message |
| nativeLoadData | 从native 加载数据 |

collect过程中NativeBatchProcessor#nativeProcessInput ->BatchHandler#xx\_nativeProcessI

-nput ->BatchHandler#onInputData

处理源码如下：



->BatchHandler#handleInput

->MCollectorOutputHandler#handlerInput

->FixSizeContainer#fill

->MapOutputCollector#allocateKVBuffer

->PartitionBucket#allocateKVBuffer->MemoryBlock(....) ->MapOutputCollector#middleSpill

->MapOutputCollector#sortPartitions->ParitionBucket#sort

->PartitionBucket#spill(IFileWriter)

->IFileWriter#write

........

->MapOutputCollector#close->MapOutputCollecotr#finalSpill

->MapOutputCollecotr#sortPartions

->Merger#merge

->IFileWriter#write

在MCollectorOutputHandler定义了缓存\_kvContainer:FixSizeContainer(核心变量char \* buff)