HADOOP MAPREDUCE: AN INTRODUCTION TO MAPREDUCE PROGRAMMING USING JAVA PERSPECTIVE

Anugya Kanswal[1], Mridul Saran[2], Lovekush[3]

[1,2]Department of Computer Science & Engineering Birla Institute of Applied Sciences, Bhimtal, Nainital, Uttarakhand (India)

[3]Department of Computer Science & Engineering Krishna Institute of Engineering and Technology, Ghaziabad Uttar Pradesh (India)

ABSTRACT

The term MapReduce actually refers to two different and tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples. The biggest advantage of using mapreduce is that live data can be queried potentially. MR on key-value stores can be highly efficient at joining data sets that are identically sharded on the join key. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job. Each job is contained in it's own class and should contain static sub-classes for any included mapper, combiner, and reducer functions. These sub-classes aren't required though, and there are several classes to handle these functions included in Hadoop. MapReduce can be implemented in many different languages. In this paper we will discuss implementation of MapReduce framework using JAVA perspective. Google's cluster executes a number of MapReduce programs and MapReduce jobs everyday. Programs are parallelized automatically and are executed on cluster of commodity machines.

I. INTRODUCTION

With a huge amount of devices available in the present time to collect data, such as camera, sensors, microphones etc. every day, we create almost about 2.5 quintillion bytes of data which is a huge explosion in terms of data being collected worldwide. The amount is so large that we can say that about 90% of the data present in the world today has been created alone in last two years.

The term “Big Data” is defined as a large collection of data, too high in terms of volume, velocity and variety that may be unstructured, grow quickly and demand cost-effective, innovative forms of information processing for enhanced insight and decision making.

According to IBM, 80% of data captured today is unstructured, from sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals, to name a few. All of this unstructured data is Big Data.

II. MAP REDUCE

MapReduce is basically a programming model by Google for the purpose of processing data and then generating large data sets with parallel algorithms on clusters.

The framework was first developed as a replacement for earlier used indexing algorithms by Google, thus providing a more efficient way for web page indexing. MapReduce framework is beneficial for beginner deveopers because library routines are used to create parallel programs and the developer need not worry about infra-cluster communication, task monitoring or failure handling processes.

MapReduce runs on a large cluster of commodity machines,is highly scalable and is implemented by multiple programming languages, like Java, C# and C++.

The MapReduce framework usestwo functions namely, the "Map" and "Reduce" functions, that are used in functional programming where a problem is broken down into a set of functions that take inputs and produce outputs, although both the functions work differently in the case of MapReduce programming.

The function "Map" allows different points of the distributed cluster to distribute their work and performs extraction, filtering, and sorting.

The function "Reduce," reduces the final clusters' results into a single output and basically performs a summary operation.

Computational processing occurs on data stored in a file system or within a database, which takes a set of input key values and produces a set of output key values.

The MapReduce framework has a very good mechanism of fault tolerance, where periodic reports from each node in the cluster are expected after the completion of work. Further, the MapReduce infrastructure or framework coordinates and controls the distributed servers, running the various tasks in parallel and managing data transfers between the different parts of the system thus, eliminating redundancy and providing overall management.

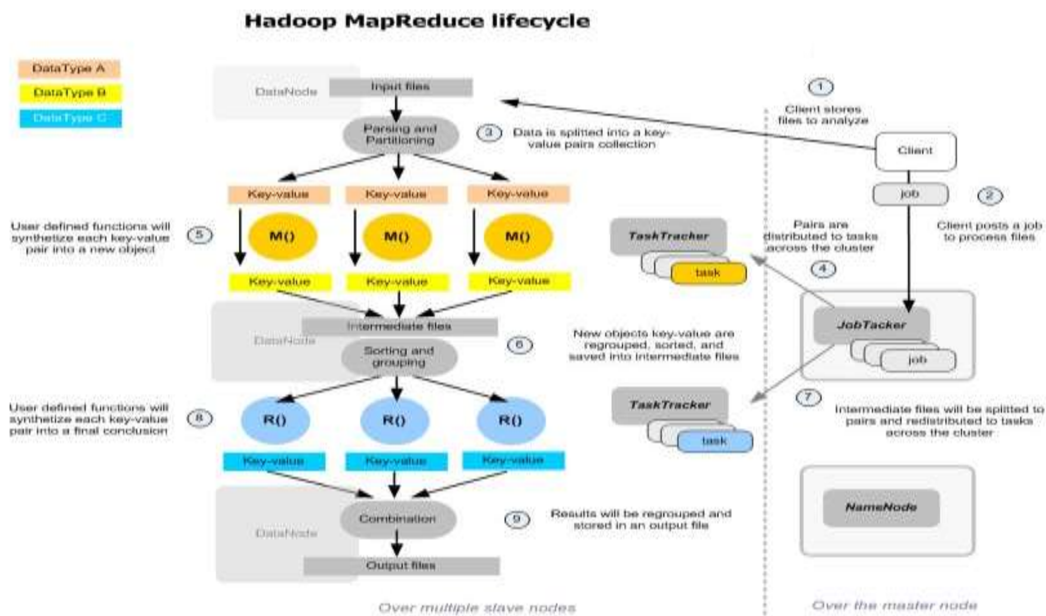
Google's cluster executes a number of MapReduce programs and MapReduce jobs everyday. Programs are parallelized automatically and are executed on cluster of commodity machines. The partioning of the input data, scheduling of the execution across machines and failure handling reqiresintermachinecommunivation. Programmers with low experience with parallel and distributed systems can use the resources of a large distributed systems without any difficulty.

III. HADOOP MAPREDUCE

Hadoop MapReduce (Hadoop Map/Reduce), a part of the Apache's Hadoop project, is basically a software framework used for the distributed processing of data sets(very large in terms of size) on clusters of commodity hardware. The framework manages scheduling of tasks, monitoring them and re-execution in case of failure.

As per the Apache Software Foundation, the main objective of Map/Reduce is to divide the input data set into independent parts that can be processed in a parallel manner. The Hadoop Map Reduce framework firstly, sorts the outputs of the map function, which are then provided as input to the reduce tasks. Both the input and the output of the functions are stored in a file system.

3.1. Architecture and Working



Prior to executing the Mapper function, the master node partitions the input into smaller sub problems which are then distributed to worker nodes. There is a single JobTracker for the cluster. Each datanode on the cluster has a TaskTracker and thus there are multiple TaskTrackers — possibly dozen, hundreds, or thousands, depending on the number of nodes in the cluster. The JobTracker runs on the master node, and a TaskTracker works on each data (or worker) node. Worker nodes may themselves act as master nodes in that they in turn may partition the sub-problem into even smaller sub-problems. In the Reduce step the master node takes the answers from all of the Mapper sub-problems and combines them in such a way as to get the output that solves the problem. Hadoop MR generally provides multiple Reduce tasks to finalize the output. A unified result — one output file — is not a requirement for the output Hadoop MR, but depending on the user's wish only one Reduce task may run. With large problems, user's preference may be the output files themselves to be distributed across the HDFS, as the output of one MR Job may be the input to a separate and later MR Job or another job. This chaining of jobs can be done with Oozie or other job coordination frameworks.

A Map function reads an input file — or, generally, a split of a file (in the form of a block or blocks of a file or files) — creates a series of key / value pairs, processes each pair, and generates zero or more output key / value pairs. Map functions do not require any order in the input data and do not provide or require dependencies from one record of data to another. The output of a Map function is written to the local file system as only one copy is needed — it is input to a Reduce function and just a temporary set of data. The Combiner and Partition functions fit between a Map function and a Reduce function. The Map function potentially outputs a large number of key / value pairs. For the Reduce function to work, all key / value pairs for a particular key value must be sent to the same Reducer.The Partition function decides to which Reducer a particular key / value pair is to be sent. The Partition function is given a key and the number of Reducers. It uses some process, typically a hash function, or a modulo function, that converts the key value into a reducer index value. Whatever function is used to generate the Reducer selection index, you want it to attempt to evenly distribute the data across all reducer datanodes. There is a default partitioning function but user has the option of coding his/her own. Since all records emitted by all of the Map functions are transferred to Reducer datanodes, and since Hadoop processing typically works

with large amounts of data, there remains a lot of netwtraffic between the nodes in the cluster. This has a negative effect on the overall performance of the job. Two slowest parts of processing within an HDFS cluster are: disk access on a datanode (i.e., reading the data blocks / splits) and inter datanode transmission of data. The Combiner function is used to lessen the network traffic. The Combiner is optional since it is not applicable in all use cases. Normally Hadoop MR applications are written in Java. But there is an API that allows the Map and Reduce functions to be written in any language — such as Perl or Ruby — that can read from standard input and write to standard output. Hadoop Streaming uses UNIX standard streams to interface between Hadoop and your non-Java program. In addition, Hadoop Pipes is used to interface between C++ and Hadoop MR. Sockets are used as the communication mechanism between the TaskTracker and the C++ Map or Reduce functions.

The MapReduce framework performs operations exclusively on <key, value>pairs,i.e., for the framework the input that is given to the job is a set of <key, value> pairs and the output of the job comes as a set of <key, value> pairs as well.

The key and value classes need to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

MapReduce job performs input and output in following manner:

(input) <k1, v1> -> map -><k2, v2> -> combine -><k2, v2> -> reduce -><k3, v3> (output)

IV. MAPREDUCE PROGRAMMING WITH JAVA

Applications typically implement the Mapper and Reducer interfaces to provide the map and reduce methods. These form the core of the job.

4.1. Mapper

Mapper maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

The Hadoop Map Reduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Overall, Mapper implementations are passed the Job for the job via the Job.setMapperClass(Class) method. The framework then calls map(WritableComparable, Writable, Context) for each key/value pair in the InputSplit for that task. Applications can then override the cleanup(Context) method to perform any required cleanup.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to context.write(WritableComparable, Writable).

Applications can use the Counter to report its statistics.

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the Reducer(s) to determine the final output. Users can control the grouping by specifying a Comparator via Job.setGroupingComparatorClass(Class).

The Mapper outputs are sorted and then partitioned per Reducer. The total number of partitions is the same as the number of reduce tasks for the job. Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.

Users can optionally specify a combiner, via Job.setCombinerClass(Class), to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.

The intermediate, sorted outputs are always stored in a simple (key-len, key, value-len, value) format. Applications can control if, and how, the intermediate outputs are to be compressed and the CompressionCodec to be used via the Configuration.

4.1.2. How Many Maps?

The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute.

Thus, if you expect 10TB of input data and have a blocksize of 128MB, you'll end up with 82,000 maps, unless Configuration.set(MRJobConfig.NUM_MAPS, int) (which only provides a hint to the framework) is used to set it even higher.

4.2. Reducer

Reducer reduces a set of intermediate values which share a key to a smaller set ofvalues. The number of reduces for the job is set by the user via Job.setNumReduceTasks(int).

Overall, Reducer implementations are passed the Job for the job via the Job.setReducerClass(Class) method and can override it to initialize themselves. The framework then calls reduce(WritableComparable, Iterable<Writable>, Context) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the cleanup(Context)method to perform any required cleanup.

Reducer has 3 primary phases: **shuffle**, **sort** and **reduce**.

Shuffle

Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

Sort

The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

Secondary Sort

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a Comparator viaJob.setSortComparatorClass(Class). Since Job.setGroupingComparatorClass(Class) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

Reduce

In this phase the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each <key, (list

of values)> pair in the grouped inputs.

The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable).

Applications can use the Counter to report its statistics. The output of the Reducer is *not sorted*.

4.2.1. How Many Reduces?

The right number of reduces seems to be 0.95 or 1.75 multiplied by (*<no. of nodes> * <no. of maximum containersper node>*).

With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

4.2.2. Reducer NONE

It is legal to set the number of reduce-tasks to *zero* if no reduction is desired.

In this case the outputs of the map-tasks go directly to the FileSystem, into the output path set by FileOutputFormat.setOutputPath(Job, Path). The framework does not sort the map-outputs before writing them out to the FileSystem.

4.3. Partitioner

Partitioner partitions the key space.

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive the partition, typically by a *hash function*. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.

HashPartitioner is the default Partitioner.

4.4. Counter

Counter is a facility for MapReduce applications to report its statistics. Mapper and Reducer implementations can use the Counter to report statistics.

Hadoop MapReduce comes bundled with a library of generally useful mappers, reducers, and partitioners.

4.5. Job Configuration

Job represents a MapReduce job configuration.

Job is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution. The framework tries to faithfully execute the job as described by Job, however:

Some configuration parameters may have been marked as final by administrators (see Final Parameters) and hence cannot be altered.

While some job parameters are straight-forward to set (e.g. Job.setNumReduceTasks(int)), other parameters interact subtly with the rest of the framework and/or job configuration and are more complex to set (e.g. Configuration.set(JobContext.NUM_MAPS, int)).

Job is typically used to specify the Mapper, combiner (ifany), Partitioner, Reducer, InputFormat, OutputFormat implementations. FileInputFormat indicates the set of input files (FileInputFormat.setInputPaths(Job, Path...)/ FileInputFormat.addInputPath(Job, Path)) and (FileInputFormat.setInputPaths(Job, String...)/ FileInputFormat.addInputPaths(Job, String))and where the output files should be written (FileOutputFormat.setOutputPath(Path)).

Optionally, Job is used to specify other advanced facets of the job such as the Comparator to be used, files to be put in the DistributedCache, whether intermediate and/or job outputs are to be compressed (and how), whether job tasks can be executed in a *speculative* manner (setMapSpeculativeExecution(boolean))/ setReduceSpeculativeExecution(boolean)), maximum number of attempts per task (setMaxMapAttempts(int)/ setMaxReduceAttempts(int)) etc.

Of course, users can use Configuration.set(String, String)/ Configuration.get(String) to set/get arbitrary parameters needed by applications. However, use the DistributedCache for large amounts of (read-only) data.

4.6. Task Execution & Environment

The MRAppMaster executes the Mapper/Reducer *task* as a child process in a separate jvm.

The child-task inherits the environment of the parent MRAppMaster. The user can specify additional options to the child-jvm via the mapreduce.{map|reduce}.java.opts and configuration parameter in the Job such as non-standard paths for the run-time linker to search shared libraries via -Djava.library.path=<> etc. If the mapreduce.{map|reduce}.java.opts parameters contains the symbol *@taskid@* it is interpolated with value of taskid of the MapReduce task.

Here is an example with multiple arguments and substitutions, showing jvm GC logging, and start of a passwordless JVM JMX agent so that it can connect with jconsole and the likes to watch child memory, threads and get thread dumps. It also sets the maximum heap-size of the map and reduce child jvm to 512MB & 1024MB respectively. It also adds an additional path to the java.library.path of the child-jvm.

```
<property>
    <name>mapreduce.map.java.opts</name>
    <value>
        -Xmx512M -Djava.library.path=/home/mycompany/lib -verbose:gc -
        Xloggc:/tmp/@taskid@.gc -Dcom.sun.management.jmxremote.authenticate=false -
        Dcom.sun.management.jmxremote.ssl=false
    </value>
</property>
<property>
    <name>mapreduce.reduce.java.opts</name>
    <value>
        -Xmx1024M -Djava.library.path=/home/mycompany/lib -verbose:gc -
        Xloggc:/tmp/@taskid@.gc -Dcom.sun.management.jmxremote.authenticate=false -
        Dcom.sun.management.jmxremote.ssl=false
    </value>
```

</property>

V. CONCLUSION

There is a clean abstraction for programmers.Hadoop provides “automatic” way to parallelize programs and distribute work which is similar to an assembler-like language for Hadoop ,it is fault-tolerant and if a process has a high number of intermediate results, faster network cards will help improve performance efficiency. Map and Reduce functions run in parallel and processing is done where the data resides.The transfer of data is minimized. If a datanode has a problem, then the processing is moved to another datanode. By default, a block of data get replicated to two additional different datanodes, for a total of three copies of each block. The three copies of each block are kept on two different racks, just in case one of the racks fail. The biggest advantage of running MapReduce on the key-value stores is that you can (potentially) query live data. For environments who do analytics/reporting on live data - that's a big win and a necessity. A 'full scan' (the most common pattern for map-reduce) is most likely to be much faster with raw file system access. for one - map-reduce against file systems more effectively decouples computation from storage than key-value servers do and is much easier to scale out.

REFERENCES

[1] Hadoop, http://hadoop.apache.org/mapreduce/.

[2] D. Abadi et al. Column-Oriented Database Systems. PVDLB, 2(2):1664–1665, 2009.

[3] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In EDBT, pages 99–110, 2010.

[4] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist, “X. 509 proxy certificates for dynamic delegation,” in 3rd annual PKI R&D workshop, vol. 14, 2004.

[5] I. Foster and C. Kesselman, “Globus: A meta computing infrastructure toolkit,” International Journal of High Performance Computing Applications, vol. 11, no. 2, pp. 115–128, 1997.

[6] G. Alliance, “Globus toolkit,” Online at: http://www.globus.org/toolkit/about.html, 2006.

[7] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin,W. Smith, and S. Tuecke, “A resource management architecture for metacomputing systems,” in Job Scheduling Strategies for Parallel Processing. Springer, 1998, pp. 62–82.

[8] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, “Condor-g: A computation management agent for multi-institutional grids,” Cluster Computing, vol. 5, no. 3, pp. 237–246, 2002.

[9] A. Shoshani, A. Sim, and J. Gu, “Storage resource managers: Middleware components for grid storage,” in NASA Conference Publication. NASA; 1998, 2002, pp. 209–224.