The Hadoop Framework

Nils Braden

University of Applied Sciences Gießen-Friedberg
Wiesenstraße 14
35390 Gießen
nils.braden@mni.fh-giessen.de

Abstract. The Hadoop Framework offers an approach to large-scale computing on a big number of nodes in a cluster. It is based on the MapReduce programming model and provides the programmer with any essential means to reliably distribute work without having to handle the upcoming problems of the distribution.
This paper provides a beginners approach to distributed parallel programming with the Hadoop Framework by explaining the necessary parts of the framework and giving an introduction on MapReduce.

Keywords: MapReduce, Cluster, Rack-Based-Computing, Hadoop, Data processing, Distributed programming

1 Introduction

With growing amounts of data comes the need to analyze the data. As stated by Anand Rajaraman "More data usually beats better algorithms"[1] so if the used algorithms are sufficiently evolved the best way to get better results is to use the algorithms with more data.

As of 2009 the Internet Archive stored around 2 petabytes of data and was growing at a rate 20 terabytes per month [6] and Facebook was hosting approximately 10 billion photos, taking up one petabyte of storage [3].

These amounts of data raise the problems of computing power and storage. If there are ten terabytes of data to be read and they have to be read from five nodes it will take about 330 minutes. If the data is spread across 500 nodes it is just going to take five minutes when reading at an average rate of 100 megabytes per second. If we would start to transfer the data over the network to other nodes the time would increase by some orders of magnitude.

Hadoop adresses these problems by distributing the work and the data over thousands of machines. The workload is spread evenly accross the cluster and there are several techniques to be explained which ensure fault-tolerance and data-integrity. It also ensures that only a small amount of the data is transferred over the network on processing time so it suits best in cases were data is written once and often read.

[1] See http://anand.typepad.com/datawocky/2008/03/more-data-usual.html

The MapReduce programming model allows for the programmer to work on an abstract level without having to tinker with any cluster-specific details, concurrent use of ressources, communication and data flow. The two functions, map and reduce, receive incoming data, modify it and send it to an output channel.

Comparison to RDBMS

Relational database management systems (RDBMS) are often compared to Hadoop but in fact are more like the opposite. Both systems got different strengths. While RDBMS can hardly store more then 1 terabyte of data, using Hadoop just starts to make sense with several terabytes. Hadoop shines when reading the whole data is necessary while RDBMS are using indexes and caches for random access to access small portions of the data.

Another big difference are constraints which are non-existent in Hadoop but impossible to think away from database systems. By leaving this ballast behind, Hadoop has one serious advantage over traditional systems: Hadoop scales well. If the number of nodes is doubled, the computing time for a job is roughly half of the original time which is impossible to achieve with specialized and expensive database servers.

Hadoop operates well on semi-structured or unstructured data by allowing the programmer to chose the input keys and values. For processing a textfile one would probably chose the line number as key and the line as value so there would be no need to find and check any key inside the actual data.

Comparison to Grid-Computing

The terms “Grid-Computing” and “High Performance Computing” often refer to quite different fields of computing. What matters here is the fact that large-scale computing on lots of different nodes is not a new idea but has in fact been done for years. The general approach is to distribute the work over the nodes but leave the data on one central storage which works well for compute-intensive tasks but becomes the bottleneck for tasks which require access to the whole dataset.

Hadoop follows a different approach by trying to locate the work-units on the same nodes as the necessary data for this unit is hosted because in a data center the network bandwidth becomes the most precious ressource if the amount of data is large.

The framework takes responsibility for coordination of the processes. It is complicated to check for dead nodes and differentiate between nodes which are just making slow progress, nodes that are not reachable via network at the moment but may become available later and nodes which stopped working.

2 The Hadoop Framework

This section contains a general overview of the components and their association in the Hadoop Framework without getting into the depths of what exactly they

do. The Hadoop Framework consists of several modules which provide different parts of the necessary functionality to distribute tasks and data across a cluster. These modules are explained in the sections 3 and 4 while this section covers general information about the framework, the communication and the topology of a cluster.

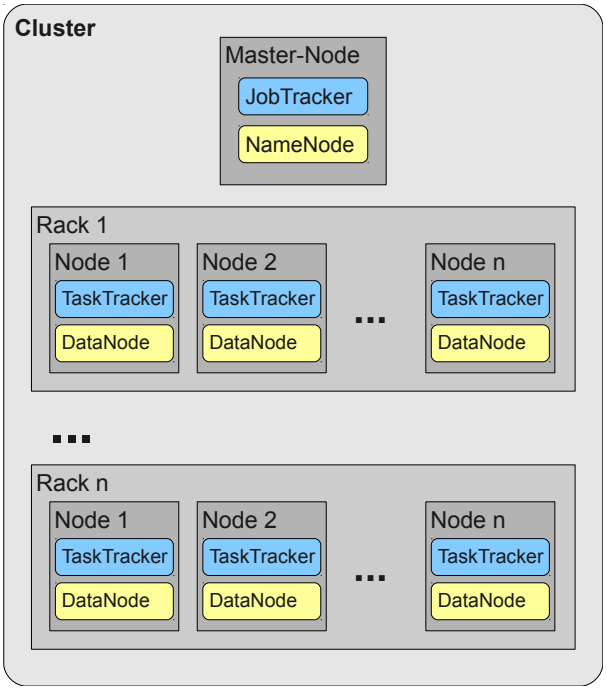

Fig. 1. Abstract view on a Cluster. A cluster consists of several nodes organized into racks, each node running a TaskTracker for the MapReduce tasks and a DataNode for the distributed storage system. One special node, the Master-Node runs the JobTracker and the NameNode which are organizing the distribution of tasks and data.

Hadoop Framework

The Hadoop Framework is written in Java but can run MapReduce programs expressed in various languages, e.g. Ruby, Python and C++. This functionality is implemented using Unix standard streams, the input for the Map function is streamed to the program which implements it, no matter which language it is implemented in as long as it can read and write from standard input and to standard output. Another way to achieve this is with the "Hadoop Pipes", an interface to C++ which uses sockets for the communication with the Map and Reduce functions.

The central parts of the framework are in the Common module which contains generally availabe interfaces and tools to support the usage of the other modules. These modules include MapReduce, a framework to distribute the processing of large data sets on compute clusters, and HDFS, the Hadoop Distributed Filesystem which enables large clusters to save and access data with high throughput.

Communication

In every cluster running Hadoop there is one node addressed as the "Master-Node". This node constitutes a single point of failure which is why Hadoop is not a high-availability system.

Communication from outside of the cluster is completely handled by the Master-Node, because it keeps all the necessary information about the cluster, usage of the nodes, disk-space and distribution of files. Every MapReduce task is sent to the MasterNode where the JobTracker, the component that manages the MapReduce jobs, and the NameNode, which is responsible for everything concerning the filesystem, are running. The JobTracker communicates with TaskTrackers on the nodes which receive work units and send reports about their status back to the JobTracker. The work of the JobTrackers is explained in detail in section 3 and the general layout of the communicating modules is shown in figure 1 on page 3. The NameNode is explained in section 4.

Cluster

A Hadoop-Cluster typically consists of 10 to 5000 nodes and is subdivided into racks by a two-level network topology[3] as shown in figure 1 on page 3. The frameworks' configuration should include information about the toplogy to allow the underlying filesytem the use of the location for its replication strategy. Also it allows the framework to place MapReduce tasks at least near the data they operate on if it happens to be impossible to place the task at the same machine because that way the "not-so-precious" in-rack bandwith is used.

3 MapReduce

The MapReduce module of the framework and its underlying technique of splitting the work in a map- and a reduce-phase are explained in detail in this section. The examples mentioned here are more thoroughly clarified in section 5.

Programming Model

The following quote from [1] summarizes the programming model:

> The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.

The division into the two phases, Map and Reduce, permits the framework to distribute the work without respect to the amount of data or the number of nodes operating in parallel. Every Map operation is self-contained and the Reduce tasks get all output data for one key so there is no overlapping in the work of the Reduce tasks. The following sections explain the Map and Reduce phase in more detail.

Map

The Map function as used by MapReduce is provided with a list of key-value pairs as input data and returns a list of key-value pairs as its' output. The keys and values from the input do not need to be of the same type as the output. All values are then grouped and sorted by the key and sent to the Reduce function.

It is optionally possible to call a user-defined Combine function before the data is sent to minimize the usage of bandwith. The Combine function is mostly the same as the Reduce function but is only applied to the output of one Map task at once. In the word-count example in section 5, a combiner would sum up the counts for every word so instead of sending lots of pairs of the form (*word*, 1) it would just send one pair for each word.

Conventionally the Map function in functional programming languages takes a list of values and applies a mapping function to every value. For example if there is a function $square\, x = x * x$ and we call $map\, square\, [1, 2, 3, 4]$ we get $[1, 4, 9, 16]$ as the result. As stated in [2],

> there is no trivial correspondence between MapReduce's MAP & REDUCE and what is normally called map & reduce in functional programming. Also, the relationships between MAP and map are different from those between REDUCE and reduce.

(*MAP* is meant to address the MapReduce's Map function while *map* stands for the map function as known from functional programming. The same goes for the Reduce function.)

Reduce

The Reduce function in the context of MapReduce is invoked with the intermediate output of the Map functions, grouped and sorted by the key. These values are typically merged together and a possibly smaller amount of key-value pairs is produced as output.

In functional programming languages the Reduce function takes a function as first argument and a list of elements as the second. Then the elements are recursively combined by applying the function to the first argument and the result of combining the rest. An example would be the calculation of the sum in the following way: We have a function $add\, x = x+1$ and call $reduce\, add\, [1, 2, 3, 4]$, as a result we get the sum of 10.

Data flow

When a MapReduce job is started in a Hadoop cluster

> the number of reduce tasks is equal to the number of reducers specified by the programmer. The number of map tasks, on the other hand, depends on many factors: the number of mappers specified by the programmer serves as a hint to the execution framework, but the actual number of tasks depends on both the number of input files and the number of HDFS data blocks occupied by those files.

(from [4])

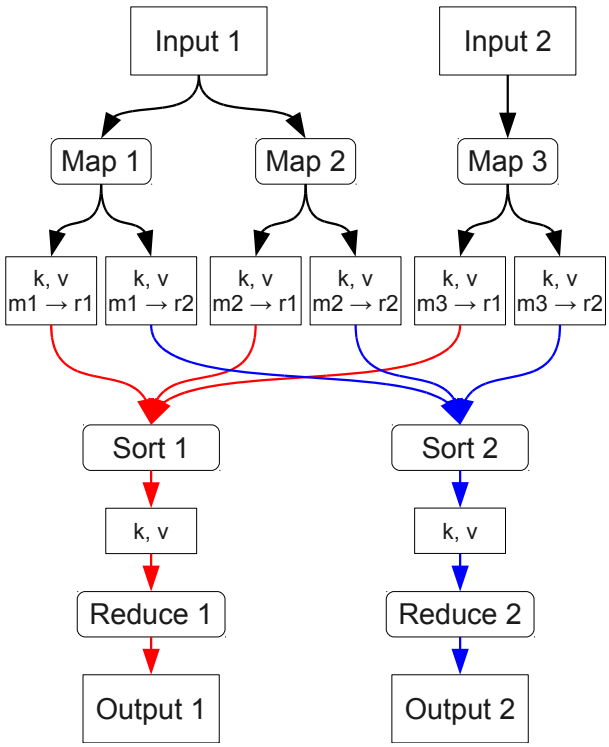

Fig. 2. Data flow in a Hadoop Cluster. The input data is passed to the Map jobs which apply their function and output key-value pairs grouped by the key and sorted. Every Reducer gets the pairs for one key and outputs the processed data into the distributed storage.

For some tasks only one Reduce invocation is necessary, for example if the result of a computation is only a sum of a lot of input values. The input data normally is already in HDFS, the Hadoop Distributed Filesystem which will be

explained in section 4, so the Map tasks are assigned to the machines which hold the relevant data. This method is used to reduce the utilized bandwith and is often referred to as "data locality" or "location awareness". The general data flow for two input blocks, three Map tasks and two Reduce tasks is shown in figure 2 on page 6.

The Map tasks apply their function to the input data and write it to their local storage so the Reduce tasks have to fetch it when they need the data. Before the data is written to disk it is implicitly sorted and grouped by the key as stated above. The data is written locally because of the anticipatory scheduling of tasks as explained in 2. If several Map tasks worked on the same data the reduce tasks would receive every information several times which would consume a lot of bandwith.

After the Reduce tasks have finished their work the output is normally written to the distributed storage as it may be the same size or even bigger than the input data and therefore cannot be safely stored to local storage.

4 Hadoop Distributed Filesystem

The Hadoop Distributed Filesystem is designed to be fault-tolerant in an environment with thousands of nodes which are likely to fail and to provide high throughput access to data [7]. This section explains the advantages of the filesystem in association with the MapReduce module.

General

In a distributed system with a large number of nodes of commodity hardware failure is the biggest problem to cope with. HDFS approaches this by replicating the data to three nodes by default, one of which is to be in another rack to make sure the data is available when a network link fails and a complete rack goes offline.

The filesystem allows for files with a size of tens of petabytes but restricts the usage of the files to streaming-access because the "time to read the full dataset (or large portions of it) is more important then latency in reading the first record." (from [3])

Other filesystems such as the CloudStore[2] or the Amazon Simple Storage Service[3] can be used instead of HDFS too, the framework provides an abstract class and several implementations for different storage systems.

Filesystem Details

The filesystem uses "Blocks" as the smalles unit of data that can be read or written. The size of a block defaults to 64 megabytes which is comparatively big

[2] see http://kosmosfs.sourceforge.net/
[3] Amazon S3, see http://aws.amazon.com/s3/

but ensures that a Map task does not need to read data from other machines because one block can not be spread out over several nodes. From this it follows that for a cluster of 1000 nodes you need at least 1000 blocks of input data for a task to occupy every node in the cluster.

In case of failure of a node the Master-Node asks the holders of the remaining copies of the lost-blocks to replicate the blocks again to bring the replication factor back to the desired level. This is necessary because

> Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional.

(from [7])

HDFS Structure

An Hadoop cluster running HDFS as a filesystem always has a central NameNode as shown in figure 1 on page 3 which handles the data distribution and any operations on the data. It communicates with the DataNode which is running on every connected node and performing the disk access. Every DataNode periodically reports a list of its blocks to the NameNode so the NameNode does not need to write this information to disk all the time. The NameNode just persistently stores information about the filesystem-tree and the metadata of the files.

The DataNodes "are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode."[8] The read- and write-requests from clients are managed by the NameNode but executed by the DataNodes because the NameNode has the filesystems' metadata and knows which nodes store the requested blocks. The user data is never going through the NameNode because this would quickly saturate its' network connection.

5 Examples

This section lists some use-cases for the convenient utilisation of the MapReduce programming model.

Counting Words

This is the most commonly used example to explain the usage of the Map and Reduce functions. It takes the input text and sums up counts for every appearing word. Optionally the Reduce function can be installed as a Combiner class to save bandwith. It sums up all counts of one map task before the result is sent.

```
virtual void Map(const MapInput& input) {
  const string& text = input.value();
  const int n = text.size();
  for (int i = 0; i < n; ) {
    // Skip past leading whitespace
    while ((i < n) && isspace(text[i]))
      i++;

    // Find word end
    int start = i;
    while ((i < n) && !isspace(text[i]))
      i++;

    if (start < i)
      Emit(text.substr(start,i-start),"1");
  }
}
```

```
virtual void Reduce(ReduceInput* input) {
  // Iterate over all entries with the
  // same key and add the values
  int64 value = 0;
  while (!input->done()) {
    value += StringToInt(input->value());
    input->NextValue();
  }

  // Emit sum for input->key()
  Emit(IntToString(value));
}
```

Sorting of large Datasets

Sorting in Hadoop normally is an implicit merge-sort which is done by the framework and does not need to be implemented. In the examples supplied with the Hadoop source code the Map and Reduce functions are left empty, there is only a special partitioning function which tries to find evenly distributed partitions based on a sample of a small subset of the keys.

The sorting takes place on the nodes which perform the (empty) mapping function and the result of this process is merged together by the reducers.

Inverted Indexing

An inverted index is an index data structure which in this example maps words to document IDs, one of the obviously common operations google needs to do

to provide the searchengine with new content. In [1] the method is explained by the following quote:

> The map function parses each document, and emits a sequence of $(word, ID)$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $(word, list(ID)$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Two-way Join

As explained in [5] the two-way join of two relations $R(A, B)$ and $S(B, C)$ can be expressed as a MapReduce task by the following steps:

1. Mappers read input from the files containing R and S
2. Tuples (a, b) from R are mapped to key-value pairs with key b and value (a, R)
3. Tuples (b, c) from S are mapped to key-value pairs with key b and value (c, S)
4. The Reducers combine tuples from R and S that have a common B-Value
5. The combined tuples are written to disk

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '2004), 137–150 (2004)
2. Lämmel, R. Google's MapReduce programming model – Revisited. Sci. Comput. Program., vol. 68, issue 3, 208–237 (2007)
3. White, T.: Hadoop: The Definitive Guide. Second Edition, Yahoo Press, (2009)
4. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce (Synthesis Lectures on Human Language Technologies). Morgan and Claypool Publishers (2010)
5. Afrati, F. N., Ullman, J. D. Optimizing joins in a map-reduce environment. 13th International Conference on Extending Database Technology, 99–110 (2010)
6. Internet Archive http://www.archive.org/about/faqs.php (Retrieved: 2010-12-09)
7. Apache Hadoop http://hadoop.apache.org/ (Retrieved: 2010-12-12)
8. Apache Hadoop http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html (Retrieved: 2010-12-12)