Map-Reduce

Marco Mura (mura@di.unipi.it)

2010 March, 31th

This paper is a note from the 2009-2010 course “Strumenti di programmazione per sistemi paralleli e distribuiti” and it’s based by the lessons of Dr. Nicola Tonellotto and his slides.

The pictures in this paper are taken from the course slides by Dr. Nicola Tonellotto.

Introduction

A typical application model we are used to do in computer systems can be represented by a process that takes an input and generates an output.

What changes if we have a very large input (Gigabytes and more) to process and a very large output to generate? A single-process application could run for long time before processing all data.

The common approach to handle this scenario is the “Divide & Conquer” model: we split the input in smaller subsets and then we process these portions with as many workers running in parallel which, each one, produces a single portion of output. Merging all the output portions we can build the final output.

In theory, this is possible and it works perfectly. But in real, this approach throws lots of problems, like

- how do we split the input and what if the input splits is greater than the number of workers?
- how do we aggregate the output?
- how do we coordinate the work and what if the workers need to share data?

And so on…

Just to decide how to design the software to resolve all those problems you may spend more time than the time to write the application code. And maybe, when you start with a new project, you may restart with all this decisions from scratch!

This is certainly a big scale problem.

The MapReduce model

Yahoo and Google, separately of course, looked for some ideas to manage this big scale problem, reducing its complexity under some hypothesis:

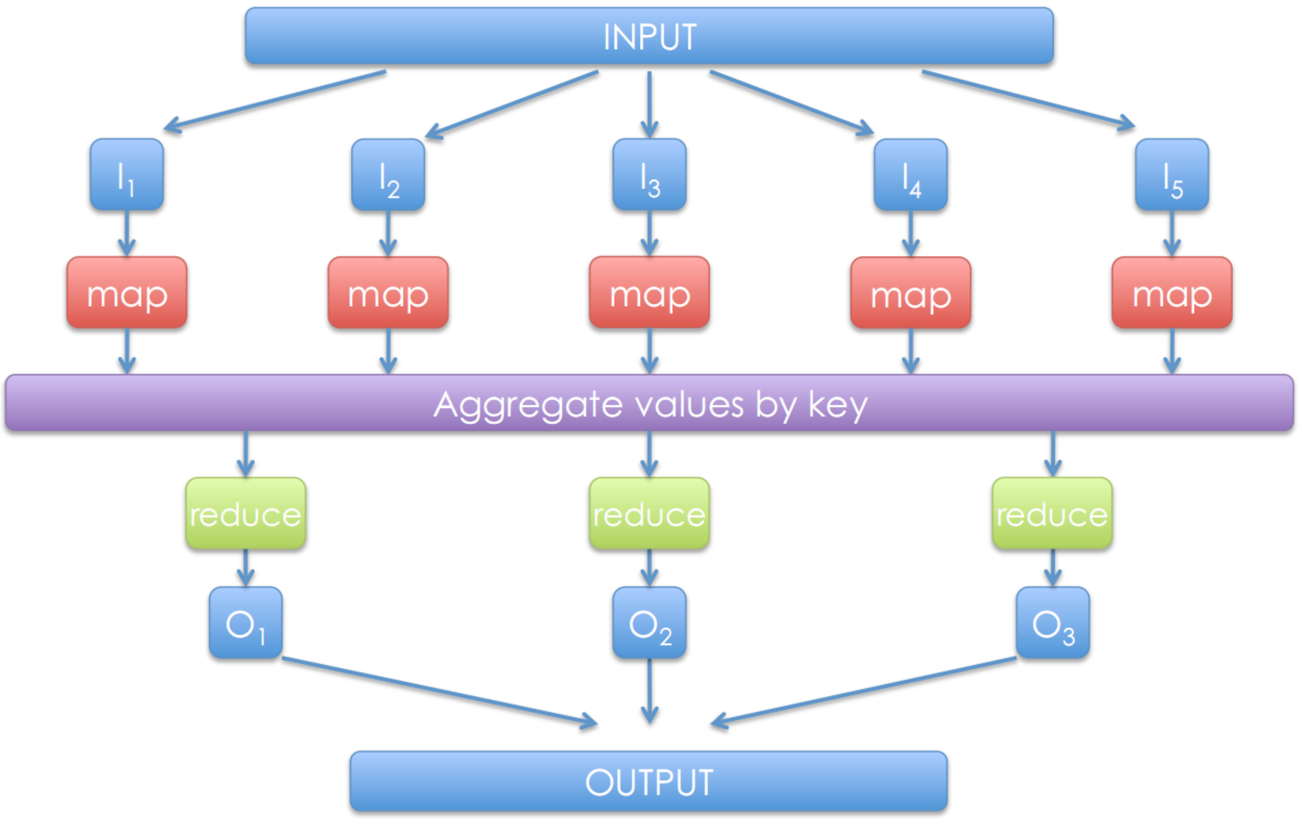
- We don’t need huge processing power, we don’t need to scale up with supercomputers… we can handle all that with low commodity infrastructure and with low end computers, just put a bunch of them into a room and you have all the processing power you need. <u>So, don’t invest money with huge computers but in a large numbers of low cost machines</u>

- <u>We don’t move the data. We move the computation where the data is stored!</u> The data is very large (Terabytes) and it’s more efficient to move a small application to run where the data is stored instead of moving the data itself.

- <u>We don’t need random access to the files.</u> Usually, in this kind of application, you read the entire input in sequence and you write the entire output in sequence. So, we don’t need standard file systems which are optimized to random file access; we will need “special” file system optimized to sequence file access. But more than that, it must be distributed to allow the data to be stored along the network.

- We don't need to know the low level details of this implementation. The framework should expose the right level of abstraction (not too high, not too low). You need to write just the business code of the application, you don't care about the reliability, how to manage failures, replications, how to send the data, etc… You just want to specify the input, the output, the code and then run it.

So they though up the MapReduce model, summary described with this schema:



The INPUT block is the entire input data we want to process and this block is splitted in several partitions I_1, I_2…, I_n. For instance, the INPUT block could be a document and every portion I_n could be a single row of the document which contains a list of few words.

The **map** (key$_1$,value$_1$) -> [(key$_2$,value$_2$)] [1]
takes one of those partition, represented by a (key$_1$,value$_1$) pair, and returns a list of (key$_2$,value$_2$) pairs not necessarly related to the (key$_1$,value$_1$) domain. In other words, the map could change the domain taking a string and returning a list of integers. For instance, map could take the row of a document and, for each word it contains, return a list of (word, 1). Note that in this instance the input value is a single row (the input key could be the row number) and the output value is the number 1 (the output key is a word).

The maps output is aggregated by keys: every (k, v_1), (k, v_2)…, (k, v_n) is stored in a single (k, [v_1;v_2;…,v_n]). For instance, for five ("pippo", 1) pair it stores ("pippo", [1;1;1;1;1]).

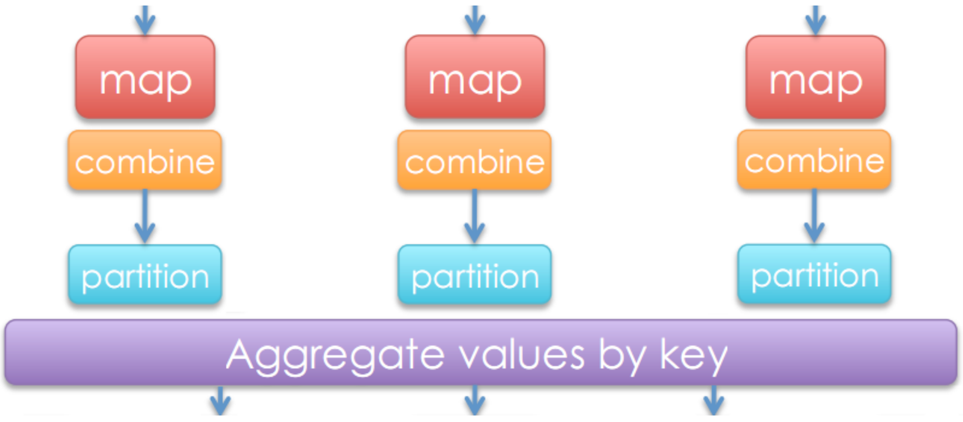
[1] In functional programming, the (T_1,T_2) value is a pair of two types T_1 and T_2. For instance, the type (string, int) represents the instances like ("blabla", 10).
Brackets are used to indicate lists: the [T_1] value is a list of T_1 elements. For instance, [int] represents instances like [1;2;3;4;5].
Consequently, the type [(T_1,T_2)] represents list of pairs; for instance [(string,int)] represents instances like [("a",1); ("b",2); ("c",3); ("d",4)].

Then the **reduce** (key$_2$, [values]) -> (key$_3$, value$_3$)
takes a single (key$_2$, [values]) pair to produce a single (key$_3$, value$_3$) pair. Usually, but not necessarily, the key$_3$ is the key$_2$ itself. For instance, the reduce could take the ("pippo", [1;1;1;1;1]) pair to produce ("pippo", 5).

Finally, the OUTPUT block is built merging all the reduce outputs.

In this model, programmers need to specify just the **map** and the **reduce** function. All the rest is done by the runtime, which handles scheduling, the data distribution, synchronization, errors, faults, etc…

Actually, the programmer must specify other two functions other than map and reduce ones, but they are not part of the Map-Reduce model: the **combine** and the **partition** functions.



The **combine** and the **partition** functions are useful to optimize the works, avoiding the transmission of useless (replicated) data and to help the load balancing.

The **combine** [(key, value)] -> (key, [values])
is called after the single map, before sending the data to the aggregator, to minimize the number of data sent. It works like a mini-reducer that combines pairs with same key with a single pair with the list of values. For instance, if the single map produces three ("pippo", 1), the **combine** combines all that pairs in a single ("pippo", [1;1;1]) one. In this instance, I send one key and three values instead of three keys and three values, resulting in a saving of transmission time.

The **partition** (key, num_reducers) -> reducer
takes a key and the number of reducers and returns which reducer will handle the specified key. With this function the programmer can balance the load specifying which reducers must process the keys. For instance, if we have two reducers and four keys "a", "b", "c" and "d", with the size |a| = 1Tb, |b| = 10 bytes, |c| = 10Gb and |d| = 1Kb, we could prefer to send "a" and "b" in the first partition and "c" and "d" in the second one, to balance the load as the best we can.

The MapReduce term can refer to some different things:
- it can refer to the MapReduce model we have just seen,
- or it can refer to its execution framework (the runtime),
- or it can refers to a specific MapReduce implementation, like MapReduce: an implementation of MapReduce in C++ by Google, used internally and not released to the public.
Hadoop is an open source implementation inspired by Google's MapReduce, it's written in Java and it's used (in production) by Yahoo who is the main contributor to the project.

Distributed file systems

Now we're going to see where and how the data is stored in a MapReduce model.

The data is not moved along the network to be sent to the node that is running the computation; on the contrary the computation is moved in the node which is storing the required data. So, in a MapReduce model any node (low-end computers ideally) performs both calculations and storage works.

To implement this model, MapReduce uses a Distributed File System. We will see two implementations of Distributed File System: the Google one called GFS (Google File System) used by their MapReduce, and the HDFS (Hadoop Distributed File System) used by Hadoop. They are very similar each other, but with some differences that we will see during the DFS explanation.

Note that the Google File System is not released to the public, so what we know about it comes from public documents released by Google itself. You can find official information about GFS here: http://labs.google.com/papers/gfs-sosp2003.pdf .

Let's talk about the features of the Distributed File System, and the main ideas which the DFS is based on:

- It is highly fault tolerance.
  Think about hardware faults: it happens. Imagine that every day there is a probability of 1/1000 that your computer has a hardware failure (just to clarify, numbers are made up to make this example). If you have 10.000 computers in your network, it is very likely that one of them has a hardware failure today. So, the DFS need to be highly fault tolerance to prevent losing data when any kind of failure (hardware or software) happens.

- High throughput.
  We don't need that jobs have a fast response time (or low latency) but we want to maximize the throughput of the file system running lot of jobs in a short time.

- Suitable for applications with large data sets.
  It's very important to reduce the time of reading the entire (huge) file instead of optimizing the read time of a single record. We are not interested in low latency data access or in small files management.

- Streaming access to file system data.
  The DFS is optimized to read and write sequentially huge files. At most, to append data to an existing file, but not to rewrite it. This is not suitable for random file accesses (reading data from random locations), replacing data into files, etc.

- Can be built out of commodity hardware.
  We want to build a DFS using low-end computers, low cost hardware. We don't want to spend money to buy reliable but expensive infrastructure, high-performing and so on.

Let's see how the data is organized and managed into a generic Distributed File System.

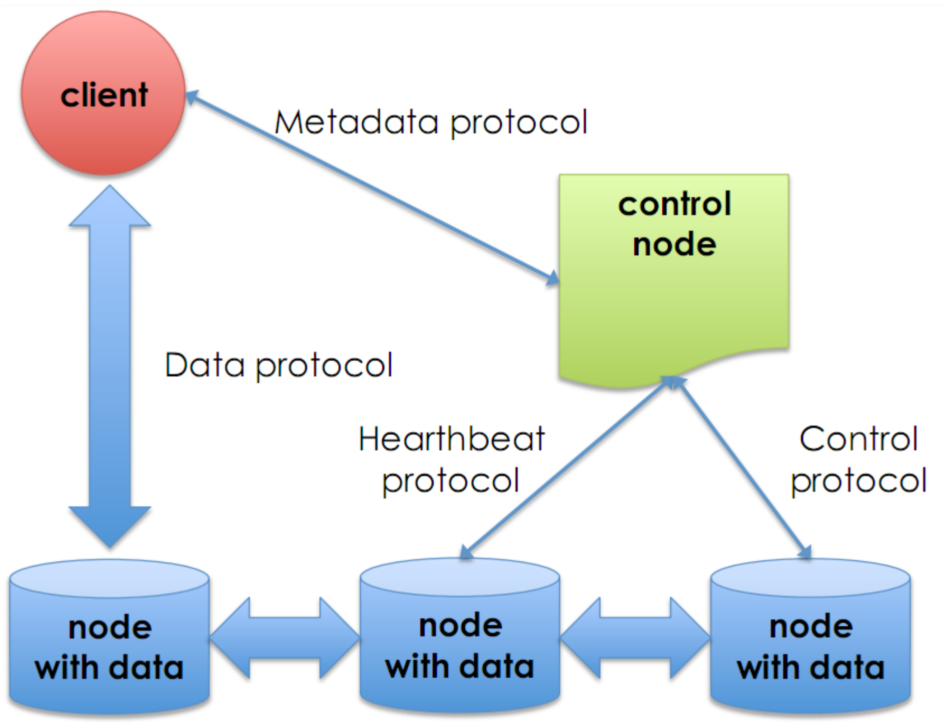
Just to introduce the topic, think about the local file system on your hard disk. When you read a file, and you read it byte by byte, the operating system buffer the operation reading a block of some Kbytes (file system blocks) to increment the performance, either because the hard disk seek time is high and we want to avoid to pay it for each byte (so you will find the next one on the memory), and because your application usually don't read sparse bytes but it's likely to read the next byte after read the current one. In few words, when a file read operation is done, the data is read block by block to increment the performance.

Reading block by block is done in Distributed File Systems too, but blocks in DFS are very much larger that in normal hard disks: they are about 64 Mbytes!
That's not a problem because files in DFS are read entirely and sequentially, so they want to minimize the impact of the disk seek time as more as possible.

This block abstraction, of course, allows files to be larger than a block (remember, we have files of Terabytes). Files are composed by several blocks not necessarily stored on the same node. And blocks are replicated in different nodes, to implement the fault tolerance.

How the files can be read?

DFS file access has a common characteristic with the FTP protocol. In FTP there are two channels: one channel for the commands (ls, cd, get…) and one channel for the data stream. In DFS we have two channels too, one to ask for a file and one for the file stream.



The client opens the data channel with a control node, or Name node, which regulates file access to the clients. This is a single point of failure of the architecture: there is only one control node! Implementations (GFS and HDFS) have their solution to recover from a control node failure.

The control node knows all the data nodes which contain the requested file blocks, including all the replicas. So the control node gives to the client the names of the data nodes to contact to operate with the file and the client opens another channel with the required data node to transmit/receive the data.

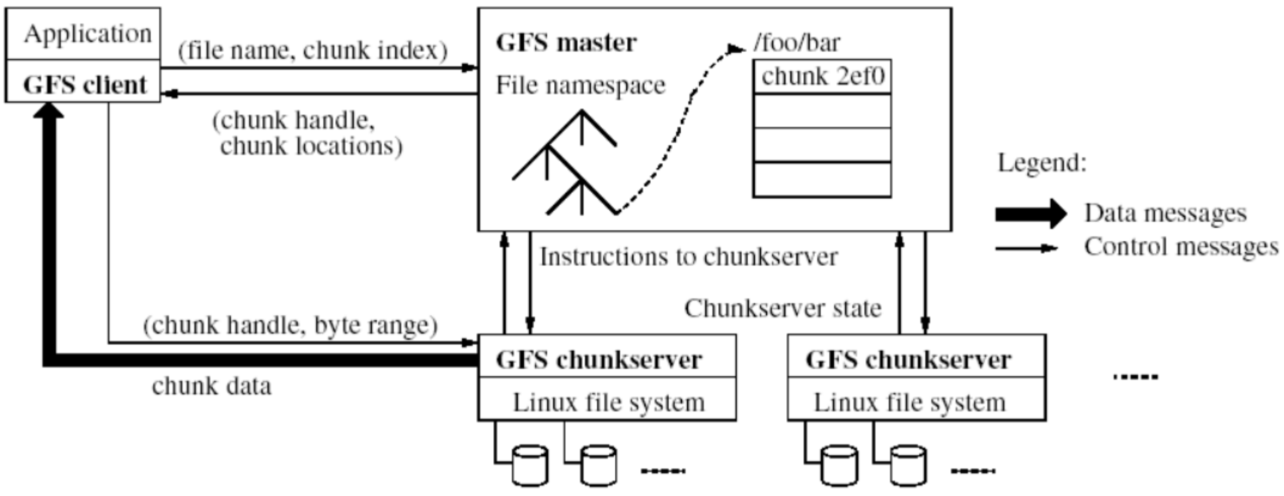
The control node also checks periodically the reliability of all data nodes and, if a node fails, activates the procedure to rebuild another replica of the blocks lost in the failed node.

The Google File System

Now we will see one specific implementation of a Distributed File System: the Google File System. Sometime we will see some difference between GFS and the Hadoop DFS.

In the Google scenario, they have few files of very large size (typically many Gbytes). They write the entire file once, and then usually they only read it. When they need to update a file, simply they append new data into the file (never rewrite). [2]
They have many sequential file reads, just few random file reads. There is no caching in reading and they don't need it because of the block size.
They don't have common multiple file writes, but sometimes it happens and they support it.

In the Google File System, the file blocks are called chunks and they are sized 64 Mbytes.
They have a simple centralized management with one Master Server (the name node) which maintains the metadata for all file chunks stored in all chunkservers (the data nodes). Each chunk is replicated in at least three chunkservers.
There isn't any data caching in the chunkservers; the only thing cached is the metadata in the master server.
The file access is implemented with custom API, it's not POSIX-compliant. They have *create*, *delete*, *open*, close, *read*, *write* functions with different semantic compared with POSIX, plus two additional operation: *snapshot* which is a control operation, and *record-append* which is the operation to append to the end of a file.



Picture taken from the referenced paper

[2] In Hadoop there isn't any support for modifying an existing file. Once written, it can only be read.

The GFS Architecture is very similar to the general one we have seen before. The client asks to the master server for a filename and a chunk index, and then the master searches the file chunks using a file namespace. Then it returns to the client the locations where the chunk can be found.

Now the client knows which chunkservers have the desired chunk, so the client asks the chunk to one of them and the chunkserver sends the data to it. The chunkservers simply uses standard linux file system to store the chunks.

The client now has received the chunk and asks to the master server for the next chunk. And so on…

The master server responsibilities, other than store and manage the metadata, are to check periodically the reliability of the chunkservers, to get information about their status and the replica management: creating/deleting/monitoring the chunks between the chunkservers.

A single master server, as we have already seen, creates a single point of failure in the architecture.
The GFS solution to this problem is storing all operations in a log file into the distributed file system, so replicated in several places. In case of Master server failure, another computer can reload the operation log and restore the master server functionalities.[3]
This doesn't mean that current writings are resumed without losses; it simply restores the functionalities with a consistent metadata.

A single master server could create a scalability bottleneck too, but this is reduced by the chunk size.
With a big chunk size, clients make less requests (and less frequently) to the master and the master need to store less information in the metadata.

What is stored in the metadata?
Basically there are filenames and chunk names, the mapping from files to chunks and the location of every replica for every chunk. The metadata is kept in memory, to a faster scanning, so the filenames are compressed.

These are the main steps for a file read:

- The client sends to the master the request Read(filename)

- The master replies with (chunk ID, chunk version, replicas locations)
  the replicas locations contains all the locations where the chunk is stored.

- The client select the first chunkserver of the list (that is the closest one) and send to it
  Read(chunkID, byte range)

- The chunkserver replies with the data.

In case of error during the last step, the client picks the next chunkserver on the list and tries with it. So the client uses the replicas locations to try one by one all replicas until the transfer is done without

[3] The Hadoop solution uses a secondary master server, with the same functionalities of the main one, which periodically performs a metadata checkpoint.

error. The list must be sorted by closeness, with an IP-based inspection for example.
In GFS the sorting of the replica locations is done by the client, in HDFS by the control node.

These are the main steps for a file write, in Google File System:

- The client sends to the master the request to write a file (appending)

- The master server checks if a chunkserver in the replicas is already holding the lease for the current chunk. Otherwise, the master elects one of them as the primary.
  The master returns to the client the location of the primary chunkserver and the locations of the others.

- The client pushes the data to be written to all the chunkservers (the primary and the secondaries). They receive the data and keep it in memory; they don't write it immediately on the file.

- When all the data is received by all chunkservers, the client asks to the primary chunkserver to write the data into the file.

- The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients. The primary applies the write to its own local state in serial number order.

- If the write is done without errors, the primary forwards the write request and order to all secondary replicas to write the data.

- The secondary replicas reply to the primary indicating that they have completed the operation. Then the primary replies to the client the end of the operation.

- In case of error, the write is failed and the client must handle the inconsistencies (retry or rollback)

The steps for a write in Hadoop Distributed File System are very similar to the GFS ones, but

- The client, instead pushing the data to all the data servers, pushes the data to the closest one only, which forwards it to the second one, and so on. So the data in pushed in pipeline to all the data servers.

- In case of error, if at least one replica has written without errors, the system asynchronously rebuilds all the replicas.

- For now, Hadoop supports only writing of new files. Append is not supported.

For now, there isn't any security level implemented. Anyone can read any files, or delete it, without any security check. Anyway, they are going to introduce some access control mechanism like the one you can find in the linux file system.