ARTICLES

NEWS

VENDORS

- AMD
- Cray
- SGI
- Dell

WHITEPAPERS

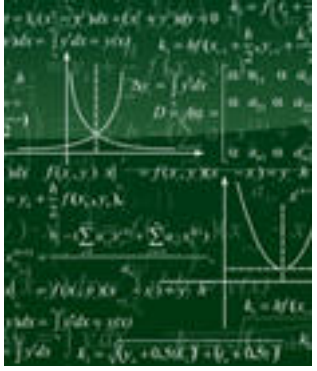
WRITE FOR US

ABOUT US

Giant Data: MapReduce and Hadoop

Thomas Hornung , Martin Przyjaciel-Zablocki , and Alexander Schätzle

Giant volumes of data are nothing unusual in our times of Google and Facebook. In 2010, Facebook sat on top of a mountain of data; just one year later it had grown from 21 to 30 petabytes. If you were to store all of this data on 1TB hard disks and stack them on top of one another, you would have a tower twice as high as the



Enterprises like Google and Facebook use the map–reduce approach to process petabyte-range volumes of data. For some analyses, it is an attractive alternative to SQL databases, and Apache Hadoop exists as an open source implementation.

Empire State building in New York.

Automatically Distributed

This example illustrates that processing and analyzing such data need to take place in a distributed process on multiple machines. However, this kind of processing has always been very complex, and much time is spent solving recurring problems, like processing in parallel, distributing data to the compute nodes, and, in particular, handling errors during processing. To free developers from these repetitive tasks, Google introduced the MapReduce framework.

The idea is based on the understanding that most data-intensive programs used by Google are very similar in terms of their basic concept. On the basis of these common features, Google developed an abstraction layer that splits the data flow into two main phases: the map phase and the reduce phase [1]. In a style similar to functional programming, computations can take place in parallel on multiple computers in the map phase. The same thing also applies to the reduce phase, so that MapReduce applications can be massively parallelized on a computer cluster.

Automatic parallelization of large-scale computations does not explain the popularity of MapReduce in companies such as Adobe, eBay, Twitter, and others. Of course, the Apache Hadoop open source implementation of MapReduce does help, but what is more likely to be important is that Hadoop can be installed on standard hardware and possesses excellent scaling characteristics, making it possible to run a cost efficient MapReduce cluster, which can be dynamically extended by purchasing more computers. An equally attractive option is not having to operate your own MapReduce cluster but accessing cloud capacity instead. Amazon, for example, offers Amazon Elastic MapReduce clusters that adapt dynamically to customer requirements.

MapReduce Framework

The pillar of a MapReduce system is a distributed file system whose basic functionality is easily explained: Large files are split into blocks of equal size, which are distributed across the cluster for storage. Because you always need to consider the failure of the computer in a larger cluster, each block is stored multiple times (typically three times) on different computers.

In the implementation of MapReduce, the user applies an alternating succession of *map* and *reduce* functions to the data. Parallel execution of these functions, and the difficulties that occur in the process, are handled automatically by the framework. An iteration comprises three phases: map, shuffle, and reduce (Figure 1).

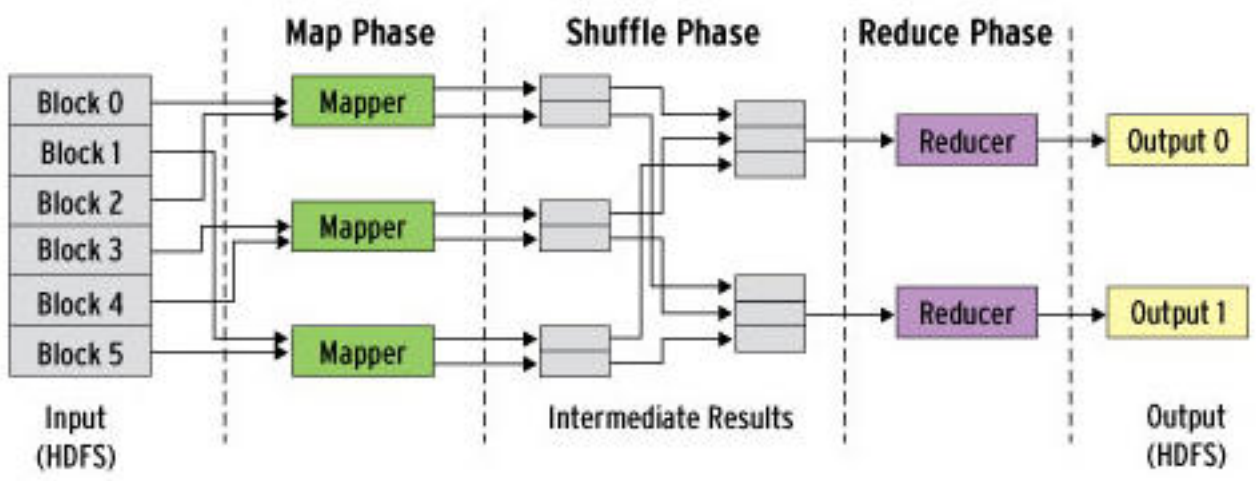


Figure 1: The MapReduce framework breaks down data processing into map, shuffle, and reduce phases. Processing is mainly in parallel on multiple compute nodes.

Map Phase

The map phase applies the *map* function to all input. For this to happen, mappers are launched on all computers in the cluster, whose task it is to process the blocks in the input file that are stored locally on the computer. In other words, computations take place where the data is stored (data locality). Because no dependencies exist between the various mappers, they can work in parallel and independently of one another. If a computer in the cluster fails, the last or not yet computed map results can be recalculated on another computer that possesses a replica of the corresponding block.

A mapper processes the contents of a block line by line, interpreting each line as a key-value pair. The actual *map* function is called individually for each of these pairs and creates an arbitrarily large list of new key-value pairs from it:

```
map(key, value) -> List(key', value')
```

Shuffle Phase

The shuffle phase sorts the resulting pairs from the map phase locally by their keys, after which, MapReduce assigns them to a reducer according to their keys. The framework makes sure all pairs with the same key are assigned to the same reducer. Because the output from the map phase can be distributed arbitrarily across the cluster, the output from the map phase needs to be transferred across the network to the correct producers in the shuffle phase. Because of this, it is normal for large volumes of data to cross the network in this step.

Reduce Phase

The reducer finally collates all the pairs with the same key and creates a sorted list from the values. The key and the sorted list of values provides the input for the *reduce* function.

The *reduce* function typically compresses the list of values to create a shorter list – for example, by aggregating the values. Commonly, it returns a single value as its output. Generally speaking, the *reduce* function creates an arbitrarily large list of key-value pairs, just like the *map* function:

```
reduce(key, List(values)) -> List(key', value')
```

The output from the reduce phase can, if needed, be used as the input for another map–reduce iteration.

Example: Search Engine

A web search engine is a good example for the use of MapReduce. For a system like this, it is particularly important to be able to compute the relevance of the page on the web as accurately as possible. One of many criteria is the number of other pages that link to one page on the web. Viewed in a simple way, this assumption is also the basic idea behind the page rank algorithm that Google uses to evaluate the relevance of a page on the web.

For this to happen, Google continuously searches the web for new information and stores the links between the pages in doing so. If you consider the number of pages and links on the web, it quickly becomes clear why computing the page rank algorithm was one of the first applications for MapReduce at Google.

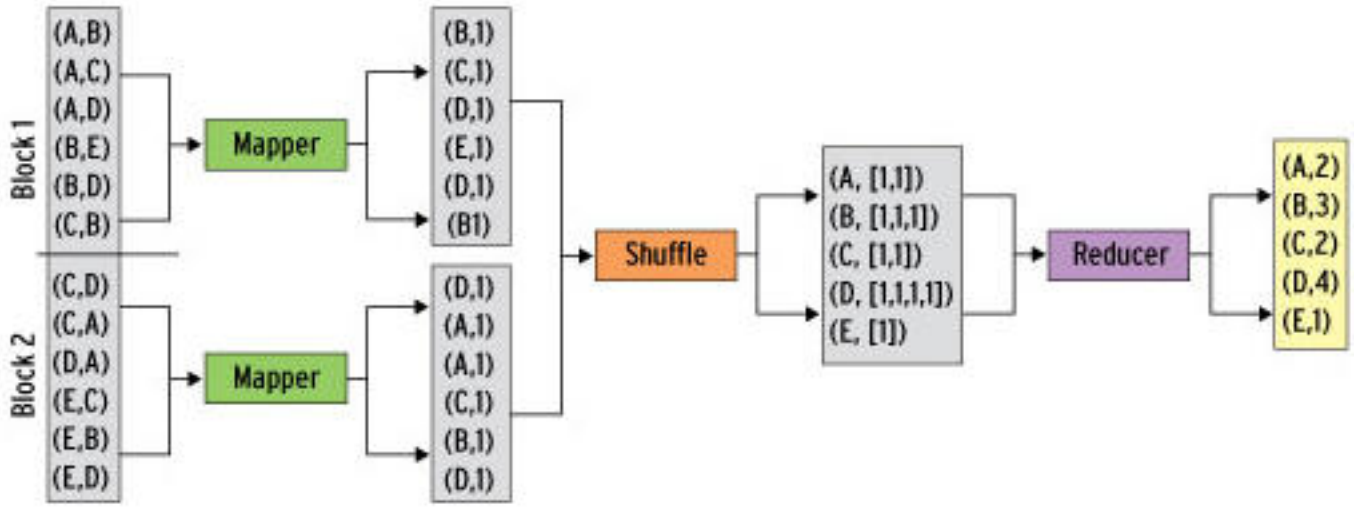


Figure 2: Schematic process of a map–reduce computation.

Figure 2 is a schematic for computing the number of incoming links for a page with the map–reduce method. The input comprises sets of (*X*, *Y*) pairs, each of which corresponds to a link from page *X* to page *Y*. It is subdivided into two blocks of six entries each – of course, in real life, the input would comprise far more blocks.

The framework assigns a mapper to each block of input, and the mapper runs the *map* function against every entry in the block. To count the number of pages that link to a specific page, it is useful to use the link target (the second value in the input pair) as a key for the output from the map function; all pairs with the same key will be collated downstream. The output value *1* from the *map* function indicates a link to the corresponding page:

```
method map(source,target)
  emit(target,1)
end
```

The shuffle phase collates all the output with identical keys from the map phase, sorts the output, and distributes it to the reducers. Consequently, for each linked page there is precisely one pair with the corresponding page as the key and a list of *1*s as the value. The reducer then applies the *reduce* function to each of these pairs. In this case, it simply needs to add up the number of *1*s in the list (pseudocode):

```
method Reduce(target,counts[c1,c2,...])
  sum <- 0
  for all c in counts[c1,c2,...] do
    sum <- sum + c
  end
  emit(target,sum)
end
```

If you take a look at the schematic, you quickly see that a mapper can create multiple key-value pairs for one page. The output from the mapper for block 1, for example, contains the (*B*, *1*) pair twice because there are two links to page *B* in block *1*.

Another thing you notice is that the values are not aggregated until they reach the reducer. The Hadoop framework uses a combiner, which prepares the output from a mapper before sending it across the network to the reducer, thus reducing the volume of data that needs to be transferred.

In the example here, the combiner can simply correlate all of the output from one mapper and compute the

total (Figure 3).

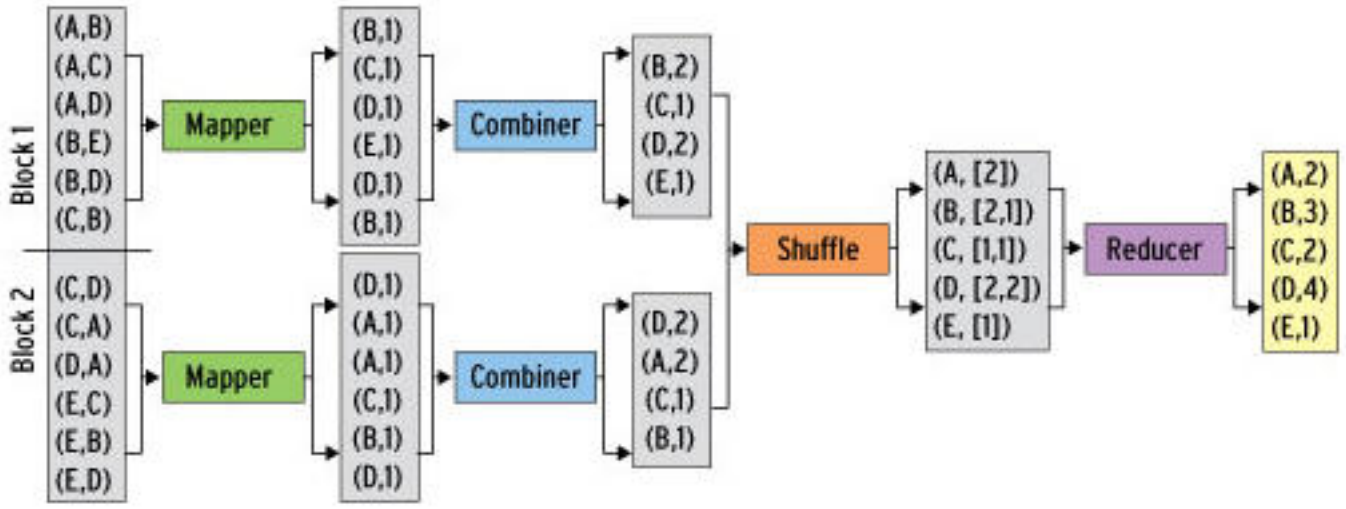


Figure 3: The use of a combiner makes sense for arithmetic operations in particular.

Combiners are useful, but they can't always be integrated into the process meaningfully; thus, they are thus optional. The user has to decide in each use case whether the combiner will improve the process or not.

Hadoop

The Apache Hadoop Project is an open source implementation of Google's distributed filesystem (Google File System, GFS [2]) and the MapReduce Framework. One of the most important supporters promoting this project over the years is Yahoo!. Today, many other well-known enterprises, such as Facebook and IBM, as well as an active community, contribute to ongoing development. Through the years, many associative projects have arisen to add functionality by extending the classic framework.

Hadoop Ecosystem

The Hadoop Ecosystem contains numerous extensions that cover a variety of application scenarios.

- **Pig** is a system developed by Yahoo! to facilitate data analysis in the MapReduce framework. Queries are written in the data transfer motion language Pig Latin, which prefers an incremental and procedural style compared with the declarative approach of SQL. Pig Latin programs can be translated automatically into a series of map–reduce iterations, removing the need for the developer to implement *map* and *reduce* functions manually.
- **Hive** is a data warehouse developed by Facebook. In contrast to Pig Latin, the Hive query language follows the declarative style of SQL. Hive automatically maps to the MapReduce framework at execution time.
- **HBase** is a column-oriented NoSQL database based on HDFS. In typical NoSQL database style, HBase is useful for random read/write access in contrast to HDFS.

Distributed Filesystem

The Hadoop Distributed Filesystem (HDFS) follows the pattern laid down by GFS. The architecture follows the classic master-slave principle, wherein one computer in the cluster (Name Node) takes care of management and the other computers (Data Nodes) take care restoring the data blocks. The data block is stored on multiple computers, improving both resilience to failure and data locality, taking into account that network bandwidth is a scarce resource in a large cluster.

To take some of the load off the network, Hadoop distributes computations in the map phase to the computers so that the greatest amount of data possible can be read locally. HDFS was designed and developed in

particular for efficient support of write once/read many access patterns with large files. This also explains why the developers set much store by fast data throughput, although this has a negative effect on latency. It is only logical that changes to store files are not supported.

Hadoop MapReduce

The MapReduce implementation in Hadoop also conforms to the master-slave architecture. The Job Tracker coordinates the sequence and assigns subtasks to the individual task trackers. A task tracker can handle the role of a mapper as well as that of a reducer.

The most important characteristic of MapReduce is the linear scalability of the framework. Simply put, the computational duration of a MapReduce application can be reduced by half (approximately) by doubling the size of the cluster. In practical terms, the genuine scaling benefits will depend on many factors, such as the nature of the problem you are solving.

Besides scalability, Hadoop possesses other characteristics that facilitate the development of distributed applications. For example, it automatically captures hardware failures and reruns subtasks that have not completed. It also performs subtasks multiple times toward end of computations to prevent a single outlier from unnecessarily slowing down the entire process (speculative execution).

Automatic parallelization of execution by Hadoop doesn't mean that developers no longer need to worry about the process. On the contrary, you need to split the problem you want to solve into a fairly strict sequence of map and reduce phases, which is often very difficult or even impossible. This relatively inflexible schema is also one of the main criticisms leveled at MapReduce.

MapReduce vs. SQL

Managing and analyzing large volumes of data is the classic domain of relational databases, which use SQL as a declarative query language. Does the widespread use of MapReduce now mean that relational databases are superfluous? You can only answer this question on a case-by-case basis. The following aspects will provide orientation:

- **Data volume:** Map reduce is suitable for very large volumes of data that go well beyond the processing capacity of a single computer. To make comparable strides with relational databases, you also need to parallelize query processing. Although possible, this approach will not typically scale in a linear way as you increase the number of computers used.
- **Access patterns:** One of the ideas behind MapReduce is that processing data sequentially in large blocks optimizes the bleed rate. In contrast, queries that only relate to a part of the data can be answered more efficiently with the help of indices in relational databases. If you wanted to answer a query of this kind with MapReduce, you would need to read the entire data record.
- **Data representation**: One assumption of relational databases make is that the data possesses an inherent structure (a schema). Users leveraged this structure to achieve what, in effect, is redundancy-free storage data in various tables. What this means for queries, however, is that they often need to combine information from various tables. MapReduce does not support the schema concept but leaves it to the user to convert the data in the map phase to the form required for the reduce phase. The benefit of this is that MapReduce supports more universal usage than a relational database.
- **Ad hoc queries**: One of the major strengths of relational databases is the declarative query language SQL. In MapReduce, the programmer has to solve each task individually.

This list could go on. One important point for a decision in an enterprise environment will certainly be the

question of sustainability and longevity of Hadoop. Relational databases are firmly established, and many companies possess sufficient know-how to cover their own requirements sufficiently. What you can expect, however, is coexistence of the two approaches, which supplement one another.

Hadoop in Production Use

Besides the official Hadoop distribution by Apache, the Cloudera distribution including Apache Hadoop (CDH) is also widespread. The Cloudera website includes downloadable preconfigured system images for various virtual machines.

- **Tutorials**: For a comprehensive introduction to Hadoop, check out Tom White's book, *Hadoop: The Definitive Guide* [3]. The Cloudera website also includes step-by-step examples with small records.
- **Programming language**: Hadoop is implemented in Java, which is why the *map* and *reduce* functions are typically also implemented in Java; however, developers also have the option of using other languages such as Python or C++.
- **Java API**: A new MapReduce API for Java was introduced in Hadoop version 0.20. The classes belonging to the legacy API are available in the *org.apache.hadoop.mapred.** Java package; the new ones are in *org.apache.hadoop.mapreduce.**. Many of the examples available still use the old API. Nevertheless newcomers are advised to adopt the new interface at the outset; the MapReduce program cannot use both versions.

Conclusions

From Google's original idea of reducing the complexity of distributed applications, Apache Hadoop has established a rich Ecosystem of versatile tools and data processing. In particular, because of its excellent scaling properties, built-in error torrents, and many useful automation features, Apache Hadoop is something that many enterprises and research groups would not want to do without in their daily work.

Of course, this doesn't mean that classic data processing systems like relational databases are no longer needed, but when handling the increasingly large volumes of digital knowledge available in the world, scaling systems like Hadoop will continue to gain importance.

Info

[1] Dean, J., S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *In*: *OSDI '04: 6th Symposium on Operating Systems Design and Implementation* (USENIX and ACM SIGOPS, 2004), pp. 137-150

[2] Ghemawat, S., H. Gobioff, and S. T. Leung: "The Google File System". *In*: M. L. Scott and L. L. Peterson, eds. *Symposium on Operating Systems Principles (SOSP)* (ACM, 2003), pp. 29-43

[3] White, T. *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly, 2012.

The Authors

Thomas Hornung, Martin Przyjaciel-Zablocki, and Alexander Schätzle are members of the scientific staff at the University of Freiburg computer science department, where they research parallel processing of semantic data.