MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.[1][2]

A MapReduce program is composed of a *map* procedure (or method), which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a *reduce* method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is a specialization of the *split-apply-combine* strategy for data analysis.[3] It is inspired by the map and reduce functions commonly used in functional programming,[4] although their purpose in the MapReduce framework is not the same as in their original forms.[5] The key contributions of the MapReduce framework are not the actual map and reduce functions (which, for example, resemble the 1995 Message Passing Interface standard's[6] *reduce*[7] and *scatter*[8] operations), but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine. As such, a single-threaded implementation of MapReduce will usually not be faster than a traditional (non-MapReduce) implementation; any gains are usually only seen with multi-threaded implementations on multi-processor hardware.[9] The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play. Optimizing the communication cost is essential to a good MapReduce algorithm.[10]

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been genericized. By 2014, Google was no longer using MapReduce as their primary *big data* processing model,[11] and development on Apache Mahout had moved on to more capable and less disk-oriented mechanisms that incorporated full map and reduce capabilities.[12]

Contents

Overview

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware). Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

A MapReduce framework (or system) is usually composed of three operations (or steps):

1. **Map:** each worker node applies the `map` function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of redundant input data is processed.
2. **Shuffle:** worker nodes redistribute data based on the output keys (produced by the `map` function), such that all data belonging to one key is located on the same worker node.
3. **Reduce:** worker nodes now process each group of output data, per key, in parallel.

MapReduce allows for distributed processing of the map and reduction operations. Maps can be performed in parallel, provided that each mapping operation is independent of the others; in practice, this is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time, or that the reduction function is associative. While this process can often appear inefficient compared to algorithms that are more sequential (because multiple instances of the reduction process must be run), MapReduce can be applied to significantly larger datasets than a single "commodity" server can handle – a large server farm can use MapReduce to sort a petabyte of data in only a few hours.[13] The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data are still available.

Another way to look at MapReduce is as a 5-step parallel and distributed computation:

1. **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key *K1* that each processor would work on, and provides that processor with all the input data associated with that key.
2. **Run the user-provided Map() code** – Map() is run exactly once for each *K1* key, generating output organized by key *K2*.

3. **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the *K2* key each processor should work on, and provides that processor with all the Map-generated data associated with that key.
4. **Run the user-provided Reduce() code** – Reduce() is run exactly once for each *K2* key produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by *K2* to produce the final outcome.

These five steps can be logically thought of as running in sequence – each step starts only after the previous step is completed – although in practice they can be interleaved as long as the final result is not affected.

In many situations, the input data might already be distributed ("sharded") among many different servers, in which case step 1 could sometimes be greatly simplified by assigning Map servers that would process the locally present input data. Similarly, step 3 could sometimes be sped up by assigning Reduce processors that are as close as possible to the Map-generated data they need to process.

Logical view

The *Map* and *Reduce* functions of *MapReduce* are both defined with respect to data structured in (key, value) pairs. *Map* takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

`Map(k1,v1) → list(k2,v2)`

The *Map* function is applied in parallel to every pair (keyed by `k1`) in the input dataset. This produces a list of pairs (keyed by `k2`) for each call. After that, the MapReduce framework collects all pairs with the same key (`k2`) from all lists and groups them together, creating one group for each key.

The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

`Reduce(k2, list (v2)) → list(v3)`

Each *Reduce* call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list.

Thus the MapReduce framework transforms a list of (key, value) pairs into a list of values. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines *all* the values returned by map.

It is necessary but not sufficient to have implementations of the map and reduce abstractions in order to implement MapReduce. Distributed implementations of MapReduce require a means of connecting the processes performing the Map and Reduce phases. This may be a distributed file system. Other options are possible, such as direct streaming from mappers to reducers, or for the mapping processors to serve up their results to reducers that query them.

Examples

The canonical MapReduce example counts the appearance of each word in a set of documents:[14]

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

Here, each document is split into words, and each word is counted by the *map* function, using the word as the result key. The framework puts together all the pairs with the same key and feeds them to the same call to *reduce*. Thus, this function just needs to sum all of its input values to find the total appearances of that word.

As another example, imagine that for a database of 1.1 billion people, one would like to compute the average number of social contacts a person has according to age. In SQL, such a query could be expressed as:

```
  SELECT age, AVG(contacts)
    FROM social.person
GROUP BY age
ORDER BY age
```

Using MapReduce, the `K1` key values could be the integers 1 through 1100, each representing a batch of 1 million records, the `K2` key value could be a person's age in years, and this computation could be achieved using the following functions:

```
function Map is
    input: integer K1 between 1 and 1100, representing a batch of 1 million social.person
records
    for each social.person record in the K1 batch do
        let Y be the person's age
        let N be the number of contacts the person has
        produce one output record (Y,(N,1))
    repeat
end function

function Reduce is
    input: age (in years) Y
    for each input record (Y,(N,C)) do
        Accumulate in S the sum of N*C
        Accumulate in C_new the sum of C
    repeat
    let A be S/C_new
    produce one output record (Y,(A,C_new))
end function
```

The MapReduce system would line up the 1100 Map processors, and would provide each with its corresponding 1 million input records. The Map step would produce 1.1 billion `(Y,(N,1))` records, with `Y` values ranging between, say, 8 and 103. The MapReduce System would then line up the 96 Reduce processors by performing shuffling operation of the key/value

pairs due to the fact that we need average per age, and provide each with its millions of corresponding input records. The Reduce step would result in the much reduced set of only 96 output records `(Y,A)`, which would be put in the final result file, sorted by `Y`.

The count info in the record is important if the processing is reduced more than one time. If we did not add the count of the records, the computed average would be wrong, for example:

```
-- map output #1: age, quantity of contacts
10, 9
10, 9
10, 9
```

```
-- map output #2: age, quantity of contacts
10, 9
10, 9
```

```
-- map output #3: age, quantity of contacts
10, 10
```

If we reduce files #1 and #2, we will have a new file with an average of 9 contacts for a 10-year-old person ((9+9+9+9+9)/5):

```
-- reduce step #1: age, average of contacts
10, 9
```

If we reduce it with file #3, we lose the count of how many records we've already seen, so we end up with an average of 9.5 contacts for a 10-year-old person ((9+10)/2), which is wrong. The correct answer is $9.\overline{166}$ = 55 / 6 = (9*3+9*2+10*1)/(3+2+1).

Dataflow

The frozen part of the MapReduce framework is a large distributed sort. The hot spots, which the application defines, are:

- an *input reader*
- a *Map* function
- a *partition* function
- a *compare* function
- a *Reduce* function
- an *output writer*

Input reader

The *input reader* divides the input into appropriate size 'splits' (in practice, typically, 64 MB to 128 MB) and the framework assigns one split to each *Map* function. The *input reader* reads data from stable storage (typically, a distributed file system) and generates key/value pairs.

A common example will read a directory full of text files and return each line as a record.

Map function

The *Map* function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other.

If the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.

Partition function

Each *Map* function output is allocated to a particular *reducer* by the application's *partition* function for sharding purposes. The *partition* function is given the key and the number of reducers and returns the index of the desired *reducer*.

A typical default is to hash the key and use the hash value modulo the number of *reducers*. It is important to pick a partition function that gives an approximately uniform distribution of data per shard for load-balancing purposes, otherwise the MapReduce operation can be held up waiting for slow reducers to finish (i.e. the reducers assigned the larger shares of the non-uniformly partitioned data).

Between the map and reduce stages, the data are *shuffled* (parallel-sorted / exchanged between nodes) in order to move the data from the map node that produced them to the shard in which they will be reduced. The shuffle can sometimes take longer than the computation time depending on network bandwidth, CPU speeds, data produced and time taken by map and reduce computations.

Comparison function

The input for each *Reduce* is pulled from the machine where the *Map* ran and sorted using the application's *comparison* function.

Reduce function

The framework calls the application's *Reduce* function once for each unique key in the sorted order. The *Reduce* can iterate through the values that are associated with that key and produce zero or more outputs.

In the word count example, the *Reduce* function takes the input values, sums them and generates a single output of the word and the final sum.

Output writer

The *Output Writer* writes the output of the *Reduce* to the stable storage.

Performance considerations

MapReduce programs are not guaranteed to be fast. The main benefit of this programming model is to exploit the optimized shuffle operation of the platform, and only having to write the *Map* and *Reduce* parts of the program. In practice, the author of a MapReduce program however has to take the shuffle step into consideration; in particular the partition function and the amount of data written by the *Map* function can have a large impact on the performance and scalability. Additional modules such as the *Combiner* function can help to reduce the amount of data written to disk, and transmitted over the network. MapReduce applications can achieve sub-linear speedups under specific circumstances.[15]

When designing a MapReduce algorithm, the author needs to choose a good tradeoff[10] between the computation and the communication costs. Communication cost often dominates the computation cost,[10][15] and many MapReduce implementations are designed to write all communication to distributed storage for crash recovery.

In tuning performance of MapReduce, the complexity of mapping, shuffle, sorting (grouping by the key), and reducing has to be taken into account. The amount of data produced by the mappers is a key parameter that shifts the bulk of the computation cost between mapping and reducing. Reducing includes sorting (grouping of the keys) which has nonlinear complexity. Hence, small partition sizes reduce sorting time, but there is a trade-off because having a large number of reducers may be impractical. The influence of split unit size is marginal (unless chosen particularly badly, say <1MB). The gains from some mappers reading load from local disks, on average, is minor.[16]

For processes that complete quickly, and where the data fits into main memory of a single machine or a small cluster, using a MapReduce framework usually is not effective. Since these frameworks are designed to recover from the loss of whole nodes during the computation, they write interim results to distributed storage. This crash recovery is expensive, and only pays off when the computation involves many computers and a long runtime of the computation. A task that completes in seconds can just be restarted in the case of an error, and the likelihood of at least one machine failing grows quickly with the cluster size. On such problems, implementations keeping all data in memory and simply restarting a computation on node failures or —when the data is small enough— non-distributed solutions will often be faster than a MapReduce system.

Distribution and reliability

MapReduce achieves reliability by parceling out a number of operations on the set of data to each node in the network. Each node is expected to report back periodically with completed work and status updates. If a node falls silent for longer than that interval, the master node (similar to the master server in the Google File System) records the node as dead and sends out the node's assigned work to other nodes. Individual operations use atomic operations for naming file outputs as a check to ensure that there are not parallel conflicting threads running. When files are renamed, it is possible to also copy them to another name in addition to the name of the task (allowing for side-effects).

The reduce operations operate much the same way. Because of their inferior properties with regard to parallel operations, the master node attempts to schedule reduce operations on the same node, or in the same rack as the node holding the data being operated on. This property is desirable as it conserves bandwidth across the backbone network of the datacenter.

Implementations are not necessarily highly reliable. For example, in older versions of Hadoop the *NameNode* was a single point of failure for the distributed filesystem. Later versions of Hadoop have high availability with an active/passive failover for the "NameNode."

Uses

MapReduce is useful in a wide range of applications, including distributed pattern-based searching, distributed sorting, web link-graph reversal, Singular Value Decomposition,[17] web access log stats, inverted index construction, document clustering, machine learning,[18] and statistical machine translation. Moreover, the MapReduce model has been adapted to several computing environments like multi-core and many-core systems,[19][20][21] desktop grids,[22] multi-cluster,[23] volunteer computing environments,[24] dynamic cloud environments,[25] mobile environments,[26] and high-performance computing environments.[27]

At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web. It replaced the old *ad hoc* programs that updated the index and ran the various analyses.[28] Development at Google has since moved on to technologies such as Percolator, FlumeJava[29] and MillWheel that offer streaming operation and updates instead of batch processing, to allow integrating "live" search results without rebuilding the complete index.[30]

MapReduce's stable inputs and outputs are usually stored in a distributed file system. The transient data are usually stored on local disk and fetched remotely by the reducers.

Criticism

Lack of novelty

David DeWitt and Michael Stonebraker, computer scientists specializing in parallel databases and shared-nothing architectures, have been critical of the breadth of problems that MapReduce can be used for.[31] They called its interface too low-level and questioned whether it really represents the paradigm shift its proponents have claimed it is.[32] They challenged the MapReduce proponents' claims of novelty, citing Teradata as an example of prior art that has existed for over two decades. They also compared MapReduce programmers to CODASYL programmers, noting both are "writing in a low-level language performing low-level record manipulation."[32] MapReduce's use of input files and lack of schema support prevents the performance improvements enabled by common database system features such as B-trees and hash partitioning, though projects such as Pig (or PigLatin), Sawzall, Apache Hive,[33] YSmart (https://archive.is/20121214201610/http://ysmart.cse.ohio-state.edu/),[34] HBase[35] and Bigtable[35][36] are addressing some of these problems.

Greg Jorgensen wrote an article rejecting these views.[37] Jorgensen asserts that DeWitt and Stonebraker's entire analysis is groundless as MapReduce was never designed nor intended to be used as a database.

DeWitt and Stonebraker have subsequently published a detailed benchmark study in 2009 comparing performance of Hadoop's MapReduce and RDBMS approaches on several specific problems.[38] They concluded that relational databases offer real advantages for many kinds of data use, especially on complex processing or where the data is used across an enterprise, but that MapReduce may be easier for users to adopt for simple or one-time processing tasks.

Google has been granted a patent on MapReduce.[39] However, there have been claims that this patent should not have been granted because MapReduce is too similar to existing products. For example, map and reduce functionality can be very easily implemented in Oracle's PL/SQL database oriented language[40] or is supported for developers transparently in distributed database architectures such as Clusterpoint XML database[41] or MongoDB NoSQL database.[42]

Restricted programming framework

MapReduce tasks must be written as acyclic dataflow programs, i.e. a stateless mapper followed by a stateless reducer, that are executed by a batch job scheduler. This paradigm makes repeated querying of datasets difficult and imposes limitations that are felt in fields such as machine learning, where iterative algorithms that revisit a single working set multiple times are the norm.[43]

Conferences and users groups

- The First International Workshop on MapReduce and its Applications (MAPREDUCE'10) (https://web.archive.org/web/20100114053209/http://graal.ens-lyon.fr/mapreduce/) was held in June 2010 with the HPDC conference and OGF'29 meeting in Chicago, IL.
- MapReduce Users Groups (http://mapreduce.meetup.com/) around the world.

See also

- Homomorphism lemma

Implementations of MapReduce

- Apache Hadoop
- Apache CouchDB
- Disco Project (http://discoproject.org/)
- Infinispan
- Riak

References

1. "Google spotlights data center inner workings" (http://news.cnet.com/8301-10784_3-9955184-7.html). *cnet.com*. 30 May 2008.
2. "MapReduce: Simplified Data Processing on Large Clusters" (http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf) (PDF). *googleusercontent.com*.
3. Wickham, Hadley (2011). "The split-apply-combine strategy for data analysis". *Journal of Statistical Software*. **40**: 1–29. doi:10.18637/jss.v040.i01 (https://doi.org/10.18637%2Fjss.v040.i01).
4. "Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages." - "MapReduce: Simplified Data Processing on Large Clusters" (http://research.google.com/archive/mapreduce.html), by Jeffrey Dean and Sanjay Ghemawat; from Google Research
5. Lämmel, R. (2008). "Google's Map *Reduce* programming model — Revisited". *Science of Computer Programming*. **70**: 1–30. doi:10.1016/j.scico.2007.07.001 (https://doi.org/10.1016%2Fj.scico.2007.07.001).
6. http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm MPI 2 standard
7. "MPI Reduce and Allreduce · MPI Tutorial" (http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/). *mpitutorial.com*.
8. "Performing Parallel Rank with MPI · MPI Tutorial" (http://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/). *mpitutorial.com*.
9. "MongoDB: Terrible MapReduce Performance" (https://stackoverflow.com/questions/3947889/mongodb-terrible-mapreduce-performance). Stack Overflow. October 16, 2010. "The MapReduce implementation in MongoDB has little to do with map reduce apparently. Because for all I read, it is single threaded, while map reduce is meant to be used

do with map reduce apparently. Because for all I read, it is single-threaded, while map-reduce is meant to be used highly parallel on a cluster. ... MongoDB MapReduce is single threaded on a single server..."

10. Ullman, J. D. (2012). "Designing good MapReduce algorithms" (http://xrds.acm.org/article.cfm?aid=2331053). *XRDS: Crossroads, the ACM Magazine for Students*. **19**: 30. doi:10.1145/2331042.2331053 (https://doi.org/10.1145%2F2331042.2331053). (Subscription required (help)). Cite uses deprecated parameter | `subscription=` (help)
11. Sverdlik, Yevgeniy (2014-06-25). "Google Dumps MapReduce in Favor of New Hyper-Scale Analytics System" (http://www.datacenterknowledge.com/archives/2014/06/25/google-dumps-mapreduce-favor-new-hyper-scale-analytics-system/). *Data Center Knowledge*. Retrieved 2015-10-25. ""We don't really use MapReduce anymore" [Urs Hölzle, senior vice president of technical infrastructure at Google]"
12. Harris, Derrick (2014-03-27). "Apache Mahout, Hadoop's original machine learning project, is moving on from MapReduce" (https://gigaom.com/2014/03/27/apache-mahout-hadoops-original-machine-learning-project-is-moving-on-from-mapreduce/). *Gigaom*. Retrieved 2015-09-24. "Apache Mahout [...] is joining the exodus away from MapReduce."
13. Czajkowski, Grzegorz; Marián Dvorský; Jerry Zhao; Michael Conley. "Sorting Petabytes with MapReduce – The Next Episode" (http://googleresearch.blogspot.com/2011/09/sorting-petabytes-with-mapreduce-next.html). Retrieved 7 April 2014.
14. "Example: Count word occurrences" (http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0004.html). Google Research. Retrieved September 18, 2013.
15. Senger, Hermes; Gil-Costa, Veronica; Arantes, Luciana; Marcondes, Cesar A. C.; Marín, Mauricio; Sato, Liria M.; da Silva, Fabrício A.B. (2015-01-01). "BSP cost and scalability analysis for MapReduce operations". *Concurrency and Computation: Practice and Experience*. **28** (8): 2503–2527. doi:10.1002/cpe.3628 (https://doi.org/10.1002%2Fcpe.3628). ISSN 1532-0634 (https://www.worldcat.org/issn/1532-0634).
16. Berlińska, Joanna; Drozdowski, Maciej (2010-12-01). "Scheduling divisible MapReduce computations" (http://www.sciencedirect.com/science/article/pii/S0743731510002698). *Journal of Parallel and Distributed Computing*. **71** (3): 450–459. doi:10.1016/j.jpdc.2010.12.004 (https://doi.org/10.1016%2Fj.jpdc.2010.12.004). Retrieved 2016-01-14.
17. Bosagh Zadeh, Reza; Carlsson, Gunnar. "Dimension Independent Matrix Square Using MapReduce" (http://stanford.edu/~rezab/papers/dimsum.pdf) (PDF). Retrieved 12 July 2014.
18. Ng, Andrew Y.; Bradski, Gary; Chu, Cheng-Tao; Olukotun, Kunle; Kim, Sang Kyun; Lin, Yi-An; Yu, YuanYuan (2006). "Map-Reduce for Machine Learning on Multicore" (http://www.willowgarage.com/map-reduce-machine-learning-multicore). NIPS 2006.
19. Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G.; Kozyrakis, C. (2007). "Evaluating MapReduce for Multi-core and Multiprocessor Systems". *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. p. 13. CiteSeerX 10.1.1.220.8210 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.8210). doi:10.1109/HPCA.2007.346181 (https://doi.org/10.1109%2FHPCA.2007.346181). ISBN 978-1-4244-0804-7.
20. He, B.; Fang, W.; Luo, Q.; Govindaraju, N. K.; Wang, T. (2008). "Mars: a MapReduce framework on graphics processors" (http://wenbin.org/doc/papers/Wenbin08PACT.pdf) (PDF). *Proceedings of the 17th international conference on Parallel architectures and compilation techniques – PACT '08*. p. 260. doi:10.1145/1454115.1454152 (https://doi.org/10.1145%2F1454115.1454152). ISBN 9781605582825.
21. Chen, R.; Chen, H.; Zang, B. (2010). "Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling". *Proceedings of the 19th international conference on Parallel architectures and compilation techniques – PACT '10*. p. 523. doi:10.1145/1854273.1854337 (https://doi.org/10.1145%2F1854273.1854337). ISBN 9781450301787.
22. Tang, B.; Moca, M.; Chevalier, S.; He, H.; Fedak, G. (2010). "Towards MapReduce for Desktop Grid Computing" (http://graal.ens-lyon.fr/~gfedak/papers/xtremmapreduce.3pgcic10.pdf) (PDF). *2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. p. 193. CiteSeerX 10.1.1.671.2763 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.671.2763). doi:10.1109/3PGCIC.2010.33 (https://doi.org/10.1109%2F3PGCIC.2010.33). ISBN 978-1-4244-8538-3.

23. Luo, Y.; Guo, Z.; Sun, Y.; Plale, B.; Qiu, J.; Li, W. (2011). "A Hierarchical Framework for Cross-Domain MapReduce Execution" (http://yuanluo.net/publications/LUO_ECMLS2011.pdf) (PDF). *Proceedings of the second international workshop on Emerging computational methods for the life sciences (ECMLS '11)*. CiteSeerX 10.1.1.364.9898 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.364.9898). doi:10.1145/1996023.1996026 (https://doi.org/10.1145%2F1996023.1996026). ISBN 978-1-4503-0702-4.
24. Lin, H.; Ma, X.; Archuleta, J.; Feng, W. C.; Gardner, M.; Zhang, Z. (2010). "MOON: MapReduce On Opportunistic eNvironments" (http://eprints.cs.vt.edu/archive/00001089/01/moon.pdf) (PDF). *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing – HPDC '10*. p. 95. doi:10.1145/1851476.1851489 (https://doi.org/10.1145%2F1851476.1851489). ISBN 9781605589428.
25. Marozzo, F.; Talia, D.; Trunfio, P. (2012). "P2P-MapReduce: Parallel data processing in dynamic Cloud environments" (http://grid.deis.unical.it/papers/pdf/MarozzoTaliaTrunfioJCSS2012.pdf) (PDF). *Journal of Computer and System Sciences*. **78** (5): 1382–1402. doi:10.1016/j.jcss.2011.12.021 (https://doi.org/10.1016%2Fj.jcss.2011.12.021).
26. Dou, A.; Kalogeraki, V.; Gunopulos, D.; Mielikainen, T.; Tuulos, V. H. (2010). "Misco: a MapReduce framework for mobile systems". *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments – PETRA '10*. p. 1. doi:10.1145/1839294.1839332 (https://doi.org/10.1145%2F1839294.1839332). ISBN 9781450300711.
27. Wang, Yandong; Goldstone, Robin; Yu, Weikuan; Wang, Teng (October 2014). *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. pp. 799–808. doi:10.1109/IPDPS.2014.87 (https://doi.org/10.1109%2FIPDPS.2014.87). ISBN 978-1-4799-3800-1.
28. "How Google Works" (http://www.baselinemag.com/c/a/Infrastructure/How-Google-Works-1/5). baselinemag.com. "As of October, Google was running about 3,000 computing jobs per day through MapReduce, representing thousands of machine-days, according to a presentation by Dean. Among other things, these batch routines analyze the latest Web pages and update Google's indexes."
29. Chambers, Craig; Raniwala, Ashish; Perry, Frances; Adams, Stephen; Henry, Robert R.; Bradshaw, Robert; Weizenbaum, Nathan (1 January 2010). *FlumeJava: Easy, Efficient Data-parallel Pipelines* (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35650.pdf) (PDF). *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 363–375. doi:10.1145/1806596.1806638 (https://doi.org/10.1145%2F1806596.1806638). ISBN 9781450300193. Retrieved 4 August 2016.
30. Peng, D., & Dabek, F. (2010, October). Large-scale Incremental Processing Using Distributed Transactions and Notifications. In OSDI (Vol. 10, pp. 1-15).
31. "Database Experts Jump the MapReduce Shark" (http://typicalprogrammer.com/relational-database-experts-jump-the-mapreduce-shark).
32. David DeWitt; Michael Stonebraker. "MapReduce: A major step backwards" (http://craig-henderson.blogspot.com/2009/11/dewitt-and-stonebrakers-mapreduce-major.html). craig-henderson.blogspot.com. Retrieved 2008-08-27.
33. "Apache Hive – Index of – Apache Software Foundation" (https://cwiki.apache.org/confluence/display/Hive/Home).
34. Rubao Lee; Tian Luo; Yin Huai; Fusheng Wang; Yongqiang He; Xiaodong Zhang. "YSmart: Yet Another SQL-to-MapReduce Translator" (http://www.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-11-7.pdf) (PDF).
35. "HBase – HBase Home – Apache Software Foundation" (http://hbase.apache.org/).
36. "Bigtable: A Distributed Storage System for Structured Data" (http://research.google.com/archive/bigtable-osdi06.pdf) (PDF).
37. Greg Jorgensen. "Relational Database Experts Jump The MapReduce Shark" (http://typicalprogrammer.com/relational-database-experts-jump-the-mapreduce-shark). typicalprogrammer.com. Retrieved 2009-11-11.
38. Pavlo, Andrew; Paulson, Erik; Rasin, Alexander; Abadi, Daniel J.; DeWitt, Deavid J.; Madden, Samuel; Stonebraker, Michael. "A Comparison of Approaches to Large-Scale Data Analysis" (http://database.cs.brown.edu/projects/mapreduce-vs-dbms). Brown University. Retrieved 2010-01-11.
39. "United States Patent: 7650331 - System and method for efficient large-scale data processing" (http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,650,331.PN.&OS=PN/7,650,331&RS=PN/7,650,331). *uspto.gov*.

40. Curt Monash. "More patent nonsense — Google MapReduce" (http://www.dbms2.com/2010/02/11/google-mapreduce-patent/). dbms2.com. Retrieved 2010-03-07.
41. "Clusterpoint XML database" (https://web.archive.org/web/20140328121334/http://www.clusterpoint.com/). clusterpoint.com. Archived from the original (http://www.clusterpoint.com) on 2014-03-28.
42. "MongoDB NoSQL database" (http://www.mongodb.org). 10gen.com.
43. Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael; Shenker, Scott; Stoica, Ion (June 2010). *Spark: Cluster Computing with Working Sets*. HotCloud 2010.

External links