**Abstract**

While living in rapidly growing data age, it’s not easy to measure the data footprint stored electronically, IDC has estimated it to be of 4.4 zettabytes, which may grow up to 44 zettabytes by 2020. Considering exponential growth of data day-by-day, a sequential system was not able to process this large volume of data within operational time. We had to have a parallel processing platform to process this data. Google has published an approach to analyze this Big Data using MapReduce framework which provided a base for Hadoop distributed computing platform, a top level Apache open source project which is being implemented by many organizations including Facebook, Yahoo, Amazon, Walmart to name a few. In this paper, we have discussed following key points with respect to MapReduce framework implemented in Hadoop open source project: Motivation, Comparison with other parallel processing paradigm, Execution flow, Performance tuning and Industry use cases. Finally, we will conclude the discussion with further research topics in large scale data processing systems.

**1. Introduction**

Over past decade we have witnessed drastic move in data processing and management techniques. Earlier Gigabyte was the norm which has become Terabyte now. With this ever scaling data size, we have developed many parallel processing solutions to transform data into information. One thing is common among them is that most of the problem statements were simple in terms of implementation, it’s just that input dataset was very large for a usual programming approach. Hence as a solution to this, Google developed a new fault tolerant programming framework called MapReduce which completely abstracts parallel/distributed nature of execution from programmer.

Each problem is divided into two phases map and reduce on a very high level. We transform raw data into <key,value> pairs during map phase and based on that we aggregate the result for each key in reduce phase.

In spite the fact that MapReduce covers most of the common problem statements, it is not the best fit for many others, iterative machine learning problem set for example. Further, despite abstraction of parallel nature of execution, programmer needs to consider many infrastructure level tuning factors which may differ for individual problem statement.

**2. Motivation**

Google had many parallel programs running individually implemented in C++, which has commonalities of task decomposition and failure handling, coordination and communication. The goal was to develop a generic framework which provides these services inherently. Programmer needs to code two methods to generate and aggregate <key,value> pairs.

**3. Execution phases of MapReduce program**

MapReduce program reads input file from a distributed storage system which is divided into multiple chunks, does operations in order as mentioned in following. Generally, the master picks idle workers and assigns each one a map or a reduce task according to the stage. Also, all Map and Reduce tasks do not need to be executed at the same time.

**3.1 Map**

In this step, each chunk is assigned to a mapper, a worker which is assigned to a map task, and the mapper applies Map() to each record in the chunk.

**3.2 Sort**

The intermediate outputs produced by the mappers are sorted locally for grouping key-value pairs with the same key.

**3.3 Partition**

After local sort, Combine() is applied to perform pre-aggregation on the grouped key-value pairs which means the communication cost to transfer all the intermediate outputs to reducers will be reduced. After sorting the mapped outputs in local disk of the mappers, they will be partitioned into R, where R is the number of Reduce tasks in the Map-Reduce job. This partitioning is basically done by a hash function, for example: hash(key) mod R. MapReduce scheduler assigns Reduce tasks to workers, once all Map tasks are completed.

**3.4 Shuffle**

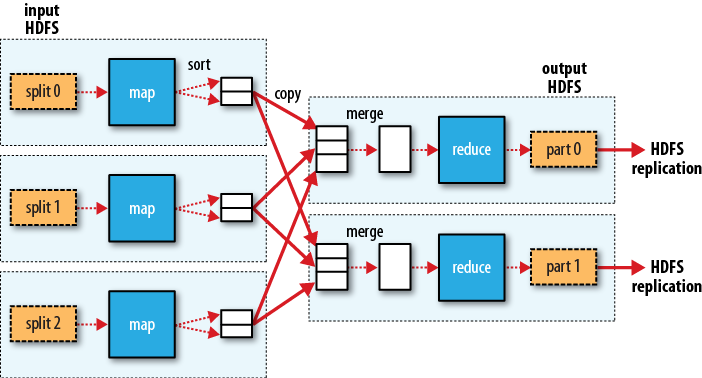
The intermediate results will be shuffled and assigned to reducers. Since all mapped outputs are already partitioned and stored in local disks, each reducer performs the shuffling by pulling its partition of the mapped outputs from mappers. Each record of the mapped outputs will be assigned to only a single reducer by one-to-one shuffling. This data transfer is performed by reducers’ pulling intermediate results.

**3.5 Merge**

A reducer reads the intermediate results and merge them by the intermediate keys. Which means all values of the same key are grouped together. This grouping will be done by external merge-sort.

**3.6 Reduce**

Each reducer applies Reduce function to the intermediate values for each encountered key. The output of reducers are stored and triplicated in HDFS.



**4. Performance issues and area of improvements**

* Output of map tasks will be written on local disc first and then transferred over the network to the node running reduce task unless reduce task is launched on the same machine. This shuffle process involves lot of I/O. We can reduce that by using relevant compression techniques while transferring data over network.
* Use of combiner class wherever applicable, aggregates result locally which reduces network I/O during shuffle phase.
* Mapper/Reducer tasks in multiple of available task slots avoid any delay caused by a small fraction of tasks running at the last.

**5. MapReduce vs Parallel Databases**

Many researchers suggested, DBMSs are not suited for solving extremely large data processing tasks. According to Pavlo et al’s comparison, Hadoop is 2∼50 times slower than parallel DBMS except in the case of data loading. It is argued by Anderson et al that the current Hadoop system is scalable, but achieves very low efficiency per node, which means previous studies on high performance systems often made by concentrating on scalability without efficiency. The lack of efficiency is basically due to many issues such as performance, total cost of ownership and energy. The cost for constructing and maintaining a cluster was considered, MapReduce would not be a proper solution. Parallel DBMS considers efficiency instead of fault tolerance. Because DBMS actively uses pipelining intermediate results between query operators, when a failure happens, it can cause a potential danger that a large amount of operations need be redone.

The MapReduce framework executes its tasks based on runtime scheduling scheme and does not build any execution plan wherein DBMS generates a query plan tree for execution.

**Advantages of Map-Reduce**

**Simple and easy to use:** MapReduce model is simple. With MapReduce, a programmer defines his job with only Map and Reduce functions, without specifying physical distribution of the job across nodes.

**Flexible:** MapReduce, unlike DBMS model, does not have any dependency on data model and schema. With MapReduce a programmer can deal with irregular or unstructured data more easily than they do with DBMS.

**Independent of the storage:** MapReduce is basically independent from storage layers. Thus, MapReduce can work with different storage layers.

**Fault tolerance:** MapReduce is highly fault-tolerant. For example, MapReduce can continue to work in spite of failures.

**High scalability:** The best advantage of using MapReduce is high scalability. For example, with YARN resource negotiator Hadoop could scale out more than 10,000 nodes.

**Limitations of Map-Reduce**

**No high-level language:** There is no high-level language like SQL in DBMS or any query optimization technique in MapReduce. Developers should code their Map and Reduce functions.

**No schema and index:** There is no schema and index in MapReduce. Which means processing eliminates the benefits of data modeling. Thus, MapReduce parse each items at reading input and transform them into data objects for data processing, which decrease performance.

**A Single fixed dataflow:** MapReduce provides simple abstraction in a fixed dataflow. Therefore, many complex algorithms are hard to implement with Map and Reduce. Additionally, some algorithms that require multiple inputs are not supported because the dataflow of MapReduce read a single input and generate a single output.

**Low efficiency:** With fault-tolerance and scalability as its primary goals, MapReduce operations are not always optimized for I/O efficiency. In addition, Map and Reduce are blocking operations. Moreover, block-level restarts, a one-to-one shuffling strategy. MapReduce does not have specific execution plans and does not optimize plans like DBMS does to minimize data transfer across nodes. MapReduce framework has a latency problem that comes from batch processing nature.

**6. Industry use cases**

**Finance:** The Finance sector is one of the major users of high computing. One of the primary use case is in risk modelling, to solve the question for banks to evaluate customers and markets better than legacy systems.

**Healthcare:** Healthcare industry has a long list of compute intensive applications as in for curing diseases, reducing medical cost, predicting and managing epidemics and maintaining the quality of human life by keeping track of large scale health index and metrics.

**Telecom:** While telecom users are growing every day, practically it is not possible to manage and process usage and billing data in real time without transforming serial algorithms into parallel one. Additionally, telecom companies are using distributed computing for applications like predicting support calls, customer segmentation and clustering to name a few.

**Retail:** Now-a-days e-commerce is transforming the buying behavior of customers like never before. Further to that big giants are receiving millions of transactions from their Brick-and-mortar outlets. For them, it is quite obvious to use parallel computing infrastructure to predict sales, real time feedback of promotional offers, identify loyalty customers, market basket analysis to name a few

**Manufacturing:** Real time sensor data processing for production infrastructure is the most common use case for manufacturing industry

**7. Further Research**

* Storing map task output into in-memory will improve performance of iterative processes, machine learning algorithms for example.
* Communicating with cluster resource manager and launching tasks only in multiple of available slots will avoid any delay caused by fraction of tasks running lastly.
* Stream oriented MapReduce flow
* Optimizer to prepare an execution plan

**References**

[1] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[2] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, Bongki Moon: Parallel Data Processing with MapReduce: A Survey https://www2.cs.arizona.edu/~bkmoon/papers/sigmodrec11.pdf

[3] Hadoop the definitive guide – by Tom White, O’Reilly publications