MapReduce: A Comprehensive Study on Applications, Scope and Challenges

Article *in* International Journal of Advance Research in Computer Science and Management · August 2015

CITATIONS
2

READS
1,752

3 authors, including:

Anurag Sarkar
Northeastern University
8 PUBLICATIONS **69** CITATIONS
SEE PROFILE

Asoke Nath
St. Xavier's College, Kolkata
264 PUBLICATIONS **1,377** CITATIONS
SEE PROFILE

ISSN: 2321-7782 (Online)
Volume 3, Issue 7, July 2015
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

MapReduce: A Comprehensive Study on Applications, Scope and Challenges

Anurag Sarkar[1]
Department of Computer Science
St. Xavier's College (Autonomous)
Kolkata, India

Abir Ghosh[2]
Department of Computer Science
St. Xavier's College (Autonomous)
Kolkata, India

Dr. Asoke Nath[3]
Department of Computer Science
St. Xavier's College (Autonomous)
Kolkata, India

Abstract: ***As the domain of Big Data continues to grow in importance and popularity, there is an increasing need to research techniques that effectively process such massive amounts of data. One such technique is the MapReduce programming model, which is the focus of this paper. We begin with an in-depth discussion of MapReduce including a brief overview and a detailed explanation of its working along with a few example problems. We then discuss its advantages and disadvantages as well as the challenges of using MapReduce and also examine various solutions that have been developed to overcome these challenges. We conclude by presenting the scope of current and future research in the field of MapReduce and also highlight its applications in the domain of bioinformatics.***

Keywords: MapReduce; Big Data; Hadoop; Bioinformatics; Database.

I. INTRODUCTION

BIG DATA

Big Data refers to data which is too vast to be processed by traditional database systems and techniques and hence requires alternative processing methods. Such methods are needed to extract valuable information and data patterns that lie hidden in the massive datasets that constitute Big Data. The challenge of Big Data analytics is thus to process such large amounts of data within a reasonable period of time so that the data can be used to gain insights that may not be discovered using conventional data analysis approaches.

Big Data is usually characterized by the three Vs – Volume, Velocity and Variety.

Volume – This refers to the large amount of information that constitutes Big Data and poses the greatest challenge to Big Data analytics. The massive volume of data calls for scalable storage and distributed querying techniques.

Velocity – This refers to the rapid pace at which large amounts of data flow into an organization as well as how fast the datasets can be processed effectively. Organizations that can quickly utilize this data gain a competitive advantage over those who don't.

Variety – A common characteristic of Big Data is that the source data is varied and does not fall into relational structures. Data may include text information from social networks, image data and data pertaining to e-commerce among others. None of these data sources is inherently suitable for data mining and analysis and requires the use of unique approaches and methods for processing.

Other characteristics of Big Data include:

Variability – The data collected may show signs of inconsistency which severely hampers the process of effectively managing and handling the data.

Veracity – This refers to the quality of the data. Accuracy of analysis is dependent on the authenticity of the data collected.

Complexity – Understandably, big data processing and management is a very complex task owing to the massive volume of data as well as the need to access required data in a time period that is practically feasible.

Big Data finds application in a wide variety of domains. These include:

- Various Government applications which require processing of incredibly vast amounts of data
- Media and advertising – big data analytics is used to gather and process information about millions of individuals and customers
- Technology – Google makes use of Big Data technologies to handle roughly 100 billion searchers per month. Similarly, Amazon processes millions of back-end operations each day.
- Banking – Global banks must handle transactions of millions of customers and worth billions of dollars worldwide each day. Security is a major concern in this domain.
- Science – Large scale scientific experiments make use of Big Data techniques to keep track of and process data in an accurate manner, e.g. Large Hadron Collider data, NASA, etc.

A popular model and framework for handling Big Data is MapReduce and that will be the focus for the remainder of this paper.

MAPREDUCE

MapReduce is a data-parallel programming model pioneered by Google for clusters of machines. It is implemented for processing and generating large datasets to solve a variety of real-world problems and tasks. The computation is specified in terms of a map function and a reduce function, and the underlying runtime system automatically parallelizes the computation across large machine clusters, handles machine failures, and schedules inter-machine communication, making effective use of the network and disks.

It offers the following useful features:

1. The programming model is simple yet expressive. A large number of tasks can be expressed as MapReduce jobs. The model is independent of the underlying storage system and is able to process both structured and unstructured data.

2. It achieves scalability through block-level scheduling. The runtime system automatically splits the input data into even-sized blocks and dynamically schedules the data blocks to the available nodes for processing.

3. It offers fault tolerance whereby only tasks on failed nodes have to be restarted.

Because of its features and benefits, MapReduce finds application in a wide variety of domains:

- large scale machine learning problems
- clustering problems for Google News
- extracting data for reports of popular queries
- extracting properties of Web pages for various purposes

- processing satellite image data
- language model processing for statistical machine translation
- large-scale graph computations
- index building for various search operations
- spam detection
- various data mining applications

There are several implementations of MapReduce but the most popular is Apache Hadoop, an open-source Java implementation. It consists of two layers – a data storage layer called Hadoop Distributed File System (HDFS) and a data processing layer called Hadoop MapReduce Framework, which is an implementation of MapReduce designed for large clusters. HDFS is a block-structured file system managed by a single master node. Each processing job in Hadoop is broken down to as many Map tasks as there are input data blocks and one or more Reduce tasks.

The master node, called the JobTracker, accepts jobs from clients, divides these jobs into tasks and assigns the tasks to the worker nodes. Each worker runs a TaskTracker process that manages the tasks currently assigned to that node. Each TaskTracker has a fixed number of slots for executing the tasks.

II. WORKING PRINCIPLE OF MAPREDUCE

A. 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 in the form of two functions: Map() and Reduce().

Map() takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key K and passes them to the Reduce() function.

The Reduce() function accepts an intermediate key K and a set of values for that key. It merges these values to form a (usually) smaller set of values. The intermediate values are supplied to the user's Reduce() function via an iterator. This helps us handle lists of values that are too large to fit in memory.

The following pseudocode demonstrates the Map and Reduce functionality with respect to the Word Count problem.

```
Map(String key, String value)  //key: document name, value: document contents
For each word w in value:
                                    EmitIntermdiate(w, "1")
Reduce(String key, Iterator values)  //key: a word, values: a list of counts
    Int result = 0
    For each v in values:
        result += ParseInt(v)
    Emit(ToString(result))
```

The MapReduce framework guarantees that all values associated with the same key (the word) are brought to the same reducer. The Reducer thus receives all values associated to the keys, sums the values and writes output key-value pairs. In the above example, the key is the word and the value is the number of occurrences.

The Partitioner is in charge of assigning intermediate keys to the Reducer. It is responsible for:

- dividing up the intermediate key-space

- assigning intermediate key/value pairs to reducers

- specifying the task to which an intermediate key/value pair must be copied

The reduce operation cannot start until all mappers have finished. In practice, a common optimization is for the reducers to pull data from mappers as soon as they finish their task.

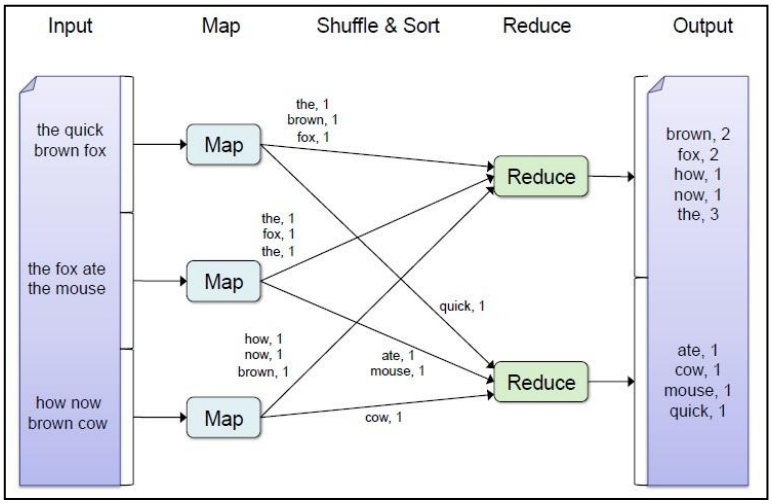

Fig.1. MapReduce implementation of the Word Count Problem

B. Execution Overview

The map tasks are distributed across multiple machines by automatically partitioning the input data into a set of M splits. These splits can be processed in parallel by different machines. Reduce tasks are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g. hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user.

When the user program calls the MapReduce() function, the following occurs:

1. The MapReduce library splits the input files into M pieces (usually 16-64 MB per piece) and starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is the master as previously specified. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks the idle workers and assigns each one either a map or a reduce task.
3. A worker assigned with a map task reads the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined map function. The intermediate key/value pairs produced by the function are buffered in memory.
4. Periodically, these buffered pairs are written to the local disk and partitioned into R regions by the partitioning function. The locations of these pairs are passed back to the master who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified about these locations, it uses remote procedure calls (RPCs) to read the buffered data from the disks of the map workers. When a reduce worker has read all intermediate data for its partition, it sorts it by the intermediate keys to group together all occurrences of the same key. If the amount of intermediate data is too large to fit in the memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user's reduce function. The output of the reduce function is appended to a final output file for this reduce partition.

7. When all map and reduce tasks have completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code. After successful completion, the output of the MapReduce execution is available in the R output files (1 per reduce task).

C. Worker Failures

Due to the presence of multiple machines, it is crucial to deal with machine failures. The master thus periodically pings each worker node. If no response is received within a certain time period, the worker is marked as failed. Any map tasks completed by it are reset back to their initial idle state and thus can be scheduled on other workers. Likewise, any map or reduce task in progress is also reset to idle and can be rescheduled. Map tasks completed by the failed worker must be re-executed because their output is stored on the failed worker's local disk. Completed reduce tasks need not be re-executed as their output is stored in the global file system. When a map task is executed first by A and then later by B because A failed, all workers executing reduce tasks are notified. Any reduce task that has not already read the data from A will read the data from B.

D. Task Granularity

The map phase and reduce phase are subdivided into M and R units respectively. Ideally, M and R should be much larger than the number of worker nodes. Dynamic load balancing is improved if each worker is made to perform many different tasks. Also, recovery from worker failure is faster because the map tasks that have been completed by the failed worker can be spread out across a large number of worker machines.

E. Backup Tasks

A straggler is a machine that takes an excessively long time to complete a map or reduce task. This leads to lengthening the total time required to complete a MapReduce operation. In order to solve this problem, when a MapReduce operation is close to being completed, the master node schedules backup executions of the remaining tasks. The task is marked as completed when any one of the executions completes.

F. Performance

There are 5 key factors that affect the performance of MapReduce. These are:

1. Grouping Schemes

The default grouping algorithm is a sort-merge algorithm and few interfaces are offered to modify the default behavior. This is because the designers felt that data grouping is a difficult task and the framework should hide such complexity. However, sort-merge is not optimal for certain tasks (such as aggregation, equal-join) and thus other schemes are often needed.

2. I/O Modes

There are 2 modes for reading data from the underlying storage system – direct I/O and streaming I/O.

In direct I/O, the data is read from a local disk where as in streaming I/O, it is read from another running process through inter-process communication (IPC). Direct I/O is more efficient when data is retrieved from the local node and thus finds use in most parallel database systems. Streaming I/O on the other hand allows the MapReduce execution engine to read data from any processes such as distributed file system processes and database instances.

3. Parsing

This is the process of converting the raw data retrieved from the storage system into the key/value pairs required for processing. Apache Hadoop employs two parsing schemes – immutable and mutable. The immutable scheme parses raw data into immutable read-only Java objects with a unique object being created for each record. The mutable scheme reuses a single mutable Java object for parsing all records. The immutable scheme is thus slower than the mutable scheme as a large number of objects are required, the creation of which incurs overhead on the CPUs.

4. Indexing

Indexing can be used to significantly speed up data processing.

5. Scheduling

MapReduce makes use of a runtime scheduling scheme. The scheduler assigns the data blocks to the available nodes for processing. This scheme however increases runtime cost and can slow down the execution of the MapReduce job. Thus, parallel database systems make use of a compile-time scheduling scheme. When a query is submitted, the optimizer generates a distributed query plan for all the nodes. Thus, there is no scheduling cost after the distributed query plan is generated. However, the runtime scheme allows MapReduce to be scalable and be able to dynamically adjust resources during the execution of a job.

A detailed discussion of the influence of the above factors on the performance of MapReduce is provided in [5].

III. Example Implementations

A. Word Count

MapReduce is usually demonstrated with the word counting problem. The pseudocode for the map and reduce functions required to solve the problem are given below.

```
def map(line):
    foreach word in line.split():
        output(word,1)
def reduce(key, values):
    output(key, sum(values))
```

We may also make use of a combiner function as below:

```
def combiner(key, values):
    output(key, sum(values))
```

Combiners are similar to reduce functions however they are not passed all the values for a given key: rather, a combiner outputs a value that summarizes the input values it was passed. Combiners are typically used to perform map-side 'pre-aggregation' which lessens the amount of network traffic required between the map and reduce steps.

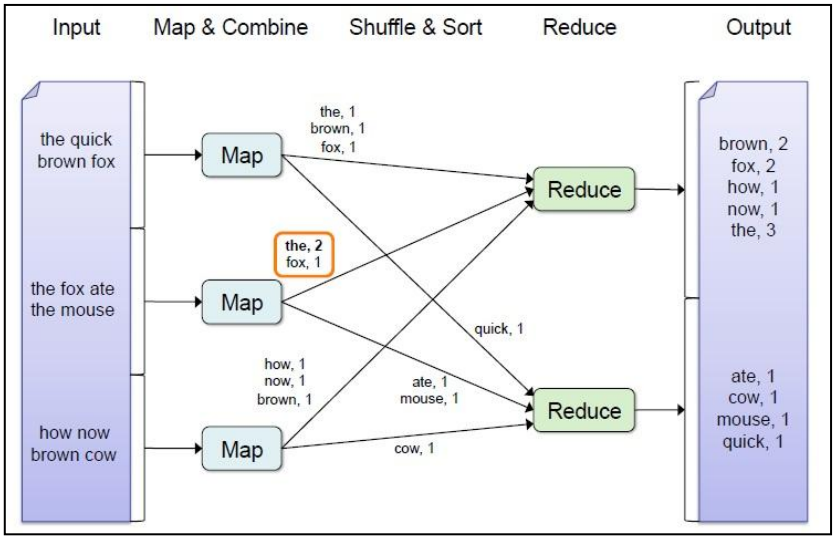

Fig. 2: MapReduce implementation of Word Count with Combiner

Snippets of sample Java and Python implementations of the above pseudocode are shown below.

1. Word Count in Java

Map

```
String line = value.toString();
StringTokenizer itr = new StringTokenizer(line);
while(itr.hasMoreTokens())
    output.collect(new text(itr.nextToken()), ONE);
```

Reduce

```
int sum = 0;
while(values.hasNext())
        sum += values.next().get();
output.collect(key, new IntWritable(sum));
```

2. Word Count in Python

Map

```
import sys
for line in sys.stdin:
        for word in line.split():
                print(word.lower() + "\t" + 1)
```

Reduce

```
import sys
counts = {}
for line in sys.stdin:
        word, count = line.split("\t")
```



```
        dict[word]  = dict.get(word,0) + int(count)

for word, count in counts:

        print(word.lower() + "\t" + 1)
```

B. Inverted Index

The inverted index problem looks at creating an index of the words that appear in a group of files, indicating for each word the list of files in which that word can be found. A MapReduce implementation for the problem is given below.

Input: Records with format (filename, text)

Output: List of files containing each word

```
def map(filename, text):

        foreach word in text.split():

                output(word, filename)

def reduce(word, filenames):

        output(word, sort(filenames))
```

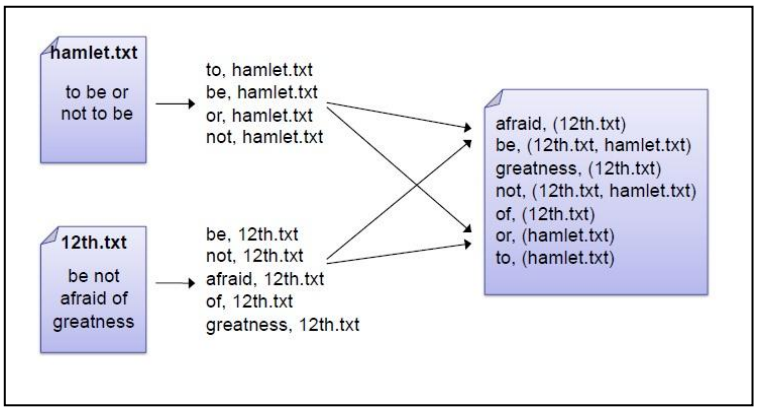

Fig 3: Inverted Index using MapReduce

C. Most Popular Words

Input: Records with format (filename, text)

Output: The 100 words that appear in the most number of files

A MapReduce implementation of this problem would proceed in 2 stages:

1. Create an inverted index that outputs (word, list(file)) records as described above.

2. Map each (word, list(file)) to (count, word) and then sort these records on the count.

```
def map(word, list(file)):

        output(size(list), word)

def reduce(count, word):

        sort(word)

        output(first 100 words after sorting)
```


IV. Advantages, Disadvantages and Improvements

Though MapReduce has been hailed as a novel way of processing big data, it has also been criticized for its inefficiency with respect to parallel data processing in comparison with Database Management Systems (DBMS). Studies have shown a tradeoff between fault tolerance and efficiency. MapReduce increases the fault tolerance of long-time analysis through regular checkpoints of completed tasks and via data replication. However, frequent I/O operations required for fault tolerance tend to reduce efficiency. Parallel DBMS on the other hand aims at efficiency instead of fault tolerance.

The advantages and disadvantages to using MapReduce are shaping the future trends and research scope of the field. We provide an overview of the pros and cons of the MapReduce framework before delving into a discussion about current and future research into improving it.

A. Advantages

1. The MapReduce model is simple but expressive. A programmer defines a job with only the Map and Reduce functions, and does not have to specify the physical distribution of the job across the nodes.

2. MapReduce does not have any dependency on the data model and schema. A programmer can work with unstructured data more easily than they do with a conventional DBMS.

3. MapReduce is independent from the underlying storage layers. Thus, it can work with different storage layers such as BigTable and various others.

4. It is highly fault tolerant and also highly scalable.

B. Disadvantages

1. MapReduce by itself does not provide support for high-level languages like SQL in DBMS nor for any query optimization technique.

2. MapReduce also does not support schemas. Thus, the framework parses each data record at reading input, leading to performance degradation.

3. MapReduce offers ease of use with a simple abstraction, but in a fixed dataflow. Hence, several complex algorithms are difficult to implement with only map and reduce functions. Additionally, algorithms that require multiple inputs are not well supported since the MapReduce data flow is designed to read a single input and produce a single output.

4. Since the primary goals of MapReduce are scalability and fault tolerance, its operations are not optimized for efficient I/O. Also, map and reduce are both blocking operations. A transition to the next stage cannot be done until all the tasks of the previous stage are completed. There is no support for pipelined execution. Additionally, MapReduce does not have execution plans and does not optimize plans as DBMS does in order to minimize data transfer across nodes. Thus, its performance is often poorer than that of DBMS. The MapReduce framework also suffers from a latency problem due to batch processing. All the inputs for a job must be prepared in advance for processing.

5. MapReduce is a relatively new technology, having been pioneered by Google in 2004. This is in contrast to the concept of DBMS which has been around for roughly forty years.

C. Improvements

The following section provides a brief survey of the ways in which many of the above problems are being solved through various improvements and enhancements.

1. High Level Languages

Microsoft SCOPE, Apache Pig and Apache Hive all support declarative query languages for the MapReduce framework. They allow query independence from program logic, query reuse and automatic query optimization.

- SCOPE is similar to SQL but consists of C# expressions.
- Pig is an open-source project that supports ad-hoc analysis of big data. It consists of a high-level data flow language known as Pig Latin and a framework for its execution.
- Hive is also an open-source project but aims at offering data warehouse solutions on top of Hadoop, supporting ad-hoc queries with an SQL-like language called HiveQL.

2. Support for Schemas

Though MapReduce itself offers no schema support, data formats such as XML, JSON (JavaScript Object Notation), Apache's Thrift and Google's Protocol Buffers can be used for checking data integrity. The formats are self-describing and support an irregular and nested data model rather than the classic relational model. A drawback of this model is that the data size may grow since data contains the schema information within itself.

3. Flexible Data Flow

Several algorithms are difficult to implement in the form of map and reduce functions. Some must maintain global state information during processing but MapReduce does not maintain such information during execution. Looping requires state information for both execution and termination. Thus, MapReduce reads the same data iteratively and materializes intermediate results in local disks for each iteration, thereby needing lots of I/O operations and excessive computations.

HaLoop, Twister and Pregel are systems that support looping in MapReduce. The first two avoid reading unnecessary data repeatedly by identifying and maintaining invariant data during iterations. Pregel is mainly concerned with processing graph data which usually requires several iterations and implements a programming model inspired by the Bulk Synchronous Parallel (BSP) model.

Clustera, Dryad and Nephele/PACT are three other systems that allow for more flexible dataflows than MapReduce. Clustera is a cluster management system that handles different types of jobs in which each job can be connected to form a directed acyclic graph (DAG) or a pipeline to perform complex computations. Dryad is a distributed data-parallel tool that lets us design and execute dataflow graphs and allows us to define how to shuffle intermediate data. Nephele/PACT is another parallel execution engine and programming model. The PACT model extends MapReduce to support dataflows that are more flexible. Nephele transforms a PACT program into a physical DAG and then executes it across different nodes. Thus, all these systems add flexibility to dataflows in MapReduce. A thorough discussion on these systems can be found in [6].

4. Blocking Operations

Map and Reduce functions are blocking operations meaning all tasks must be completed before moving forward to the next stage or job. This is because MapReduce relies on external merge sort for grouping intermediate results. This property causes performance degradation and makes online processing hard to support.

In response to this problem, MapReduce Online has been designed to support online aggregation and continuous queries in MapReduce. To support pipelining between tasks, the MapReduce architecture is modified by making the mappers push their data, which is temporarily stored on local disks, to reducers periodically in the same MapReduce job. For an in-depth study of MapReduce Online, please refer to [5].

5. I/O Optimization

The cost of I/O operations in MapReduce can be reduced by employing index structures, column-oriented storage, and data compression. Hadoop++ offers an index-structured file format to improve the I/O cost of Apache Hadoop. However, since it has to build an index for each file partition during the data loading stage, the loading time is increased significantly. If the input data is processed just once, the additional cost of building index may not be justified. HadoopDB also benefits from database indexes by leveraging DBMS as storage in each node.

6. Scheduling

MapReduce uses a block-level runtime scheduling scheme with speculative execution. A separate Map task is created to process a single data block. Thus, a node which completes its task early is assigned more tasks. Tasks on a straggler node are redundantly executed on other idle nodes.

The Hadoop scheduler implements the speculative task scheduling with a simple heuristic method which compares the progress of each task to the average progress. Tasks with the lowest progress when compared to the average are re-executed. However, this heuristic method is not suitable in a heterogeneous environment where different nodes have different computing power and a node whose task progresses further than others may still be the last if its computing power is inferior to that of others. Longest Approximate Time to End (LATE) scheduling has been devised to improve the response time of Hadoop in such heterogeneous environments. This scheme estimates the task progress with the progress rate, instead of the simple progress score.

7. Joins

Join is a popular operation in conventional databases but is not well supported by MapReduce. As MapReduce is concerned with processing a single input, supporting joins that require more than two inputs has been an open issue. Within MapReduce, join methods may be classified as either map-side join or reduce-side join.

A common map-side join is the map-merge join which is similar in working to the sort-merge join of DBMS. In map-merge join, first, two input relations are partitioned and then sorted on the join keys. Second, the mappers read the inputs and merge them. Another map-side join method is the broadcast join which can be used when one of the relations is small in size. This smaller relation is broadcast to every mapper and stored in memory. This reduces the cost of I/O for moving and sorting both relations. In-memory hash tables are used to store the smaller relation and to locate matches by means of table lookups with values from the other relation.

The most common reduce-side join is the repartition join. Here, each mapper tags each row of both relations to identify which relation the rows come from. Then, the rows of those keys that have the same key value are copied to the same reducer during shuffling. Lastly, each reducer joins the rows on the basis of key equality. This process is similar to hash-join in DBMS. More information about the application of joins in MapReduce can be found in [6].

8. Energy Issues

In the modern data-center computing area, the issue of energy consumption is crucial. In general, the energy cost of data centers accounts for about a quarter of the total monthly operating cost and thus it is essential to develop energy efficient solutions to work with the nodes in the data center.

One solution is the Covering-Set approach which designates in advance some nodes that keep a replica of each data block and other nodes that are powered down during periods of low utilization. Since the dedicated nodes always have more than one replica of the data, all data across the nodes is accessible except in the event of multiple node failures. Another solution is the All-In scheme which attempts to conserve energy in an all or nothing fashion. Here, all MapReduce jobs are queued up until a

predefined threshold is reached. If it is exceeded, all nodes in the cluster run to finish all the jobs and after completion, all the nodes are powered down until enough new jobs have been queued up.

V. CHALLENGES AND SOLUTIONS

A. Performance

Though MapReduce is lauded for its scalability, fault tolerance and ability to process Big Data, it can take several hours to execute queries. Thus it may be many orders of magnitude slower than modern DBMS and prevent interactive analysis. A large portion of query execution time is spent in task initialization, scheduling, coordination and monitoring.

Additionally, MapReduce does not by default support data pipelining or the overlapping of the map and reduce phases. Intensive disk I/O along with data materialization for fault tolerance also adds to the overall execution time. Common optimization techniques such as those used by database systems and query optimizers may find application in improving the performance of MapReduce. These optimizations include operator pipelining, online aggregation, indexing and sorting, data co-location, allowing early approximate query results, skew mitigation, reuse of previously computed results and work sharing. These are briefly discussed below.

1. Operator Pipelining and Online Aggregation:

MapReduce Online [5] is an extension to Apache Hadoop. It improves performance by supporting online aggregation and stream processing and also improves resource utilization. It enables pipelining between operators while guaranteeing fault-tolerance. Pipelining is implemented between tasks and between jobs. A drawback of this technique is that it nullifies combiners. To solve this issue, MapReduce Online buffers intermediate data till a threshold limit, applies the combiner function and spills it to disk. As a result, early results of the jobs can be computed, making approximate answers to queries available to users. This technique is known as online aggregation and produces useful early results quicker than the final results. By applying the reduce function to the data seen by the reducer so far, the system can produce an early snapshot of the final results.

2. Approximate Results:

The EARL Library is an extension to Apache Hadoop which allows incremental computations of early results by employing sampling and bootstrapping. To implement EARL, first, pipelining was enabled between the mappers and reducers, similar to MapReduce Online, in order to let the reducers start processing data as soon as they became idle. Then, the mappers were kept active and reused instead of being restarted for each iteration. This modification saved a lot of setup time. Finally, a channel was established between the mappers and reducers for communication, so that the termination condition could easily be tested.

3. Indexing and Sorting:

Lack of schemas and data indexing can lead to long query runtimes. As mentioned before, Hadoop++ is an extension to Hadoop that supports indexing. HAIL is another extension that supports inexpensive index creation on Hadoop data attributes to reduce execution times in MapReduce. More information on Hadoop++ and HAIL can be found in [7].

4. Work Sharing:

Another extension to Hadoop is MRShare which exploits sharing opportunities among different MapReduce jobs. It converts a batch of queries into a new batch by providing the optimal grouping of queries to maximize sharing opportunities.

5. Data Reuse:

ReStore is an extension to Apache Pig and stores and reuses the intermediate results of scripts originating from complete MapReduce jobs or sub-jobs. The input to ReStore is a workflow of MapReduce jobs. ReStore maintains a repository where it

stores the outputs of jobs along with the physical execution plan, the filename of the output in the Distributed File System and runtime statistics about the MapReduce job that generated the output. The system contains a plan matcher and rewriter which looks for possible matches in the repository and rewrites the workflow of the jobs to make use of stored data. Guidelines for implementing repository garbage collection in the future have been proposed.

6. Skew Mitigation:

SkewTune is a Hadoop extension that offers mechanisms to detect stragglers and repartition the input data that they are yet to process. SkewTune uses Late Skew Detection to determine when a task should be regarded as a straggler and avoid false positives and unnecessary overhead.

7. Data Co-Location:

The CoHadoop extension allows applications to control where the data is stored by stating which of the files are related and need to be processed together. CoHadoop then tries to co-locate these files to improve efficiency. Whereas the Hadoop Distributed File System (HDFS) uses a random placement policy for load-balancing, CoHadoop allows applications to store all copies of related files together by setting a new file property known as the locator. This is an integer and files with the same locator are stored in the same set of HDFS DataNodes.

B. Programming Model

Developing efficient MapReduce programs warrants advanced programming skills and a thorough understanding of the system architecture. A criticism of this model is that it is too 'low-level' compared to languages like SQL, since common data analysis tasks consist of processing multiple datasets and relational operations which are difficult to implement in MapReduce (for e.g. joins). Another criticism is the model's 'batch' nature. Data has to be uploaded to the file system and when the same dataset needs to be analyzed several times, it must be read each time. Additionally, the steps of computation are fixed and all applications must adhere to the map-shuffle-sort-reduce sequence. Complex analysis queries are achieved by chaining multiple jobs i.e. having the outputs of one job be the inputs for the next job. This however makes the model unsuitable for certain varieties of algorithms. Machine learning and graph processing often require iterative computations. Since MapReduce operators are stateless, iterative algorithms in MapReduce require manual management of state and chaining.

Several extensions have been developed to overcome some of these problems associated with the MapReduce model:

1. High Level Languages:

High level languages are less prone to bugs and maintenance and thus reduce costs as well as the amount of coding required. Apache Pig, as discussed previously, is a high-level system that comprises of a declarative scripting language called PigLatin and an execution engine that enables parallel execution of data flows on top of Hadoop. Pig hides the complexity of the MapReduce programming model and lets users write SQL-like scripts.

Hive, also discussed previously, is another such high level system. Hive was developed by Facebook and offers data warehousing in addition to abstraction. It allows easy storage, summarization and querying of large amounts of data. The high level language HiveQL lets users express queries in an SQL-like manner.

Jaql is a query language designed for querying semi-structured data using the JSON format. It is extensible and supports parallelism using Hadoop.

Cascading is a Java application framework that helps development of data processing applications on Hadoop. It provides a Java API for defining complex dataflows and abstracts the concepts of Map and Reduce and introduces the concept of flows, which consist of a data source, reusable pipes that perform operations on the data and data sinks.

2. Domain Specific Systems:

Iterative algorithms are common in domains such as machine learning and graph processing. The HaLoop extension offers built-in support for developing iterative applications in MapReduce. It provides a mechanism to cache and index invariant data between iterations thus reducing communication costs. It extends the Hadoop API enabling users to easily define loops and termination conditions.

Certain jobs need to be run repeatedly with slightly different inputs. Such computations lead to inefficiency and redundancy. To solve this, such MapReduce applications must store and use state across multiple runs. However, as MapReduce was not designed to reuse intermediate results, developing such applications is error-prone and complex. Incoop is a Hadoop extension that provides a transparent way to support incremental computations via three modifications:

- Inc-HDFS: a modified HDFS which splits data depending on the content of the file rather than size, thereby offering ways to identify similarities between datasets and thus opportunities for data reuse

- Contraction Phase: a computation phase before the reduce phase to control task granularity, using the idea of combiners to decompose the reduce task into a hierarchy of smaller tasks.

- Memoization-Aware Scheduler: a scheduler that considers data locality of previously computed results while also employing a work-stealing algorithm. It attempts to schedule tasks on nodes that have data which can be reused. When a node runs out of work, the scheduler locates the node with the largest task queue and delegates a task to the idle node.

C. Configuration & Automation

There are several configuration parameters with respect to a MapReduce cluster that affect performance. These include the number of parallel tasks, the size of the file blocks and the replication factor. Proper tuning of these parameters is essential to achieving high performance while configuring them improperly may lead to resource underutilization and inefficient execution.

As with solutions to other issues, there are several extensions which help with respect to configuration and tuning. These are:

1. Starfish:

This extension is a self-tuning system which dynamically configures system properties depending on user input and the workload. It performs tuning on three levels. At the job level, it employs a Just-in-Time (JIT) Optimizer to choose efficient execution techniques, a Profiler to learn performance models and create job profiles and a Sampler to collect information about input, intermediate and output data and help the Profiler create approximate models. At the workflow level, it uses a Workflow-Aware Scheduler which exploits data locality at the workflow level. Finally, at the workload level, it uses the Workload Optimizer for data flow sharing, materialization of intermediate results for reuse and reorganizing jobs inside a batch.

2. Disk I/O Minimization:

Sailfish is a Hadoop extension that offers opportunities for auto-tuning such as dynamically setting the number of reducers and handling skew of intermediate data. It also improves performance by reducing disk I/O due to transfers of intermediate data.

3. Data-Aware Optimization:

Manimal is an auto-optimization framework for MapReduce. It applies well-known query optimization techniques by performing static analysis of compiled code. The system's analyzer examines the user code and sends the optimization descriptors to the optimizer. The optimizer uses this information and the pre-computed indexes to choose an optimized execution plan. The new plan is then executed in the standard map-shuffle-reduce fashion.

VI. Conclusion and Future Trends

The MapReduce programming model has been instrumental in data processing and data analysis applications in the field of Big Data. This is accomplished through MapReduce's support of scalability, efficiency and fault tolerance. However, this means that the framework must make tradeoffs between these features and performance optimization. Improving upon and extending the MapReduce model to implement such optimizations and improve overall performance while not compromising on the features that have made it successful has been an important area of research and will prove to be so in the future.

A. Bioinformatics Applications

Bioinformatics is an interdisciplinary domain that is concerned with the development of software tools and methods for analyzing, processing and understanding biological data. It combines the fields of computer science, mathematics, statistics and engineering in order to process biological data.

Presently, MapReduce has found great application in this field. Research in bioinformatics involves analysis of massive datasets which will only grow as more data is compiled in the years to come. Thus, there is an increasing reliance on technologies like MapReduce and Hadoop for fault-tolerant and parallelized data analysis. We provide a brief review of applications of MapReduce in bioinformatics.

1. Next-Generation DNA Sequencing

- The CloudBurst software [11], among the first to make use of Hadoop in the field of bioinformatics, maps next generation short read sequencing data to a reference genome for SNP discovery and genotyping.
- The Crossbow software [12], which also makes use of Hadoop, performs whole genome resequencing analysis and SNP genotyping from short reads. It is a cloud-computing tool that executes in parallel.
- Myrna [13] is a cloud computing pipeline for calculating differential gene expression in large RNA-Seq datasets. It can be executed on the cloud using Amazon Elastic MapReduce, on any Hadoop cluster, or even on a stand-alone computer without using Hadoop.
- PeakRanger [14] is another Hadoop application. It splits the job by chromosomes to exploit the chromosome-level independence (CLI) of ChIP-seq data sets. In this case, 'map-then-reduce' is converted to 'split-by-chromosome-then-call-peaks' with the chromosomes being used as keys.
- Quake [15] has been developed to correct sequencing errors. It uses a Hadoop cluster to count k-mers and also to sum together the partial counts on the individual machines by using a version of the word count algorithm described previously.
- Genome Analysis Toolkit (GATK) – This is a structured programming framework that simplifies the development of efficient analysis tools for next-generation DNA sequencing by making use of the MapReduce programming model.

2. Other Bioinformatics Domains

- The MapReduce-like middleware Hydra has been used to support parameter sweep parallelism and data parallelism in bioinformatics workflows [16]
- MapReduce has also been used to implement pattern finding algorithms for analyzing complex prescription compatibility networks.
- Phylogenetic applications [17] have also been designed by making use of the MapReduce framework.

Other research topics to consider with respect to MapReduce are summarized below:

1. Incorporating Database Techniques:

Unlike traditional applications, MapReduce is data-intensive rather than computation-intensive. Hence, it is crucial to minimize disk I/O and communication. Thus, incorporating traditional database techniques, such as materialization of intermediate results, indexing and caching, into the MapReduce framework is an important area of research. Some of these extensions have been discussed in previous sections.

2. Relaxing Fault Tolerance

The original MapReduce design pioneered by Google considered clusters consisting of hundreds and thousands of machines. In this setting, since failures are commonplace, strict fault tolerance methods had to be implemented. But with the release of Apache Hadoop and other MapReduce implementations, the MapReduce framework has been employed by organizations smaller than Google who do not use nearly as many clusters as Google, and hence, suffer from much smaller rates of failure. For such smaller organizations, Google's strict fault tolerance mechanisms may be relaxed in order to achieve higher performance. This issue of fault tolerance and performance tradeoffs needs to be extensively studied in the future. Future implementations could make it possible to choose one of fault tolerance or performance over the other based on the situation.

3. Standards and Benchmarks

Different implementations of Hadoop are evaluated using different datasets and applications and there is a lack of a standard benchmark or workload for comparing various implementations and extensions. Future research needs to define what a typical MapReduce workload should be.

4. More Sophisticated High-Level Features

Although many high-level features and declarative abstractions have been incorporated into MapReduce, as seen in previous sections, MapReduce is still far from supporting interactive analysis capabilities. Though it has been augmented with several high level attributes, further benefits may be reaped with more sophisticated query optimization techniques. Mechanisms such as approximate answers and data reuse need to be studied further.

ACKNOWLEDGEMENT

The authors are grateful to the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata for giving opportunity to do research work in Big Data. AN is also grateful to Dr. Fr. Felix Raj, Principal of St. Xavier's College (Autonomous), Kolkata for giving constant encouragement to do research work in the department.

References

1. M. Zaharia, "Introduction to MapReduce and Hadoop," UC Berkeley RAD Lab.
2. J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, vol. 51, no.1, January 2008.
3. D. Jiang, B.C. Ooi, L. Shi and S. Wu, "The performance of MapReduce: an in-depth study."
4. Siddaraju, Sowmya C.L., Rashmi K. and Rahul M., "Efficient analysis of Big Data using MapReduce framework," International Journal of Recent Development in Engineering and Technology (IJRDET), vol. 2 issue 6, June 2014.
5. T. Condie et al.., "MapReduce Online."
6. K.H. Lee, Y.J. Lee, H. Choi, Y.D. Chung and B. Moon, "Parallel data processing with MapReduce: A survey," SIGMOD Record, vol. 40 no.4, December 2011.
7. V. Kalavri and V. Vlassov, "MapReduce: Limitations, Optimizations and Open Issues."
8. E. Dumbill, "What is Big Data? An introduction to the Big Data landscape." Retrieved from https://beta.oreilly.com/ideas/what-is-big-data, 2012.
9. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," BMC Bioinformatics 2010 11(Suppl 12): S1.
10. A. Tiwari, "MapReduce and Hadoop algorithms in bioinformatics papers." Retrieved from http://abhishek-tiwari.com/post/mapreduce-and-hadoop-algorithms-in-bioinformatics-papers, 2012.

11. M.C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," Bioinformatics (2009) 25 (11): 1363-1369.
12. B. Langmead, M.C. Schatz, J. Lin, M. Pop and S.L. Salzberg, "Searching for SNPs with cloud computing," Genome Biology 2009, 10:R134.
13. B. Langmead, K.D. Hansen and J.T. Leek, "Cloud-scale RNA-sequencing differential expression analysis with Myrna," Genome Biology 2010, 11:R83.
14. X. Feng, R. Grossman and L. Stein, "PeakRanger: A cloud-enabled peak caller for ChIP-seq data," BMC Bioinformatics 2011, 12:139.
15. D.R. Kelley, M.C. Schatz and S.L. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," Genome Biology 2010, 11:R116.
16. F. Coutinho et al., "Data parallelism in bioinformatics workflows using Hydra," HPDC '10, June 20-25, 2010.
17. S.J. Matthews and T.L. Williams, "MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees," BMC Bioinformatics 2010, 11(Suppl 1):S15.

AUTHOR(S) PROFILE

Anurag Sarkar, is a student of M.Sc. Computer Science, St. Xavier's College (Autonomous), Kolkata. He is currently doing research work in machine learning and big data.

Abir Ghosh, is a student of M.Sc. Computer Science, St. Xavier's College. He is currently doing research work in the field of big data.

Dr. Asoke Nath, is Associate Professor in department of Computer Science, St. Xavier's College (Autonomous), Kolkata. Apart from his teaching assignment he is involved with various researches in cryptography and network security, Visual Cryptography, Steganography, Mathematical modelling of Social Networks, Green Computing, Big data analytics, MOOCs, Quantum computing, e-learning. He has already published more than 145 papers in reputed Journals and conference proceedings. Dr. Nath is the life member of MIR Labs (USA), CSI Kolkata Chapter.