Thesis

Implications of Storage Subsystem Interactions on Processing Efficiency in Data Intensive Computing

Submitted by

Hanisha Koneru

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2015

Abstract

Implications of Storage Subsystem Interactions on Processing Efficiency in Data Intensive Computing

Processing frameworks such as MapReduce allow development of programs that operate on voluminous on-disk data. These frameworks typically include support for multiple file/storage subsystems. This decoupling of processing frameworks from the underlying storage subsystem provides a great deal of flexibility in application development. However, as we demonstrate, this flexibility often exacts a price: performance.

Given the data volumes, storage subsystems (such as HDFS, MongoDB, and HBase) disperse datasets over a collection of machines. Storage subsystems manage complexity relating to preservation of consistency, redundancy, failure recovery, throughput, and load balancing. Preserving these properties involve message exchanges between distributed subsystem components, updates to in-memory data structures, data movements, and coordination as datasets are staged and system conditions change. Storage subsystems prioritize these properties differently, leading to vastly different network, disk, memory, and CPU footprints for staging and accessing the same dataset.

This thesis proposes a methodology for comparing and identifying the storage subsystem suited for the processing that is being performed on a dataset. We profile the network I/O, disk I/O, memory, and CPU costs introduced by a storage subsystem during data staging, data processing, and generation of results. We perform this analysis with different storage subsystems and applications with different disk-I/O to CPU processing ratios.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The traditional file systems and databases cannot handle the huge amount of data storage requirements that is common now-a-days. Adding more and more storage to a system is not an optimal solution as there is an increased risk of failure and decrease in access bandwidth. Also, scaling-up is complex and expensive.

A Distributed Storage System stores data in multiple computers. But to an end user, the data appears to be located on a single local disk. The ideal properties in a distributed system are availability, consistency, scalability, transparency and security. Distributed Storage Systems can be broadly divided into two categories - Distributed File Systems and Distributed Databases. Distributed File Systems are very similar to classic file systems in terms of read and write operations with the added advantage of distributing the load of disk space over multiple machines. Though there are no fixed standards, Distribtued Databases can be categorized as Key-Value Storage, Column-Family Storage, Document Storage and Graph Storage. The data access patterns in each of these types of databases is different. There are numerous distributed storage systems available today and each system has its own specific architecture.

Different distributed systems are well suited for different types of applications. For instance, Key-Value databases are best utilized for storing user specific data such as user session data and user preferences but should be avoided when querying the database for specific data values. Where Document-oriented databases show optimal performance for e-commerce platforms and content management systems, Column-Family based databases are optimal for log applications. Graph storage systems are immensely significant for applications that have connected data, specially social networks and recommendation engines. There are numerous implementations available for each of theses categories of databases. But there is no one distributed system that is the winner of all. Depending on the use cases and deployment conditions, one storage system might outperform another and lag behind the same if the conditions change.

It is thus important to choose the right distributed system for an application. Otherwise, the advantages offered by such a system will be limited and might very well be negated by a deterrent in the same system. Performance of a distributed system is application specific. Thus, the distributed system as well as the use cases and environment in which the application needs to be deployed should be thoroughly assessed.

All distributed systems distribute the data among the data nodes in the cluster. An application, whether running on a Key-Value Storage System or a Graph Storage System or any other system, would require to access the data and process it according to the requirements. For this, the distributed components need to interact with each other and also with the client. How and when the data is accessed and processed impacts the performance of the system. By profiling these interactions between the components, we can judge the impact of an underlying storage system on the processing efficiency of the distributed system.

1.1. Research Questions

The research focus of this thesis is to provide answers for the following questions on storage frameworks.

(1) Which aspects of a storage framework's resource utilization profile impacts processing performed on the managed datasets?

(2) How can we identify differences in how storage frameworks preserve properties for managed datasets?

(3) How can we harness knowledge about costs introduced by a storage subsystem to inform suitability for processing performed in managed datasets?

1.2. Thesis Contributions

This thesis aims at providing methodologies and benchmarks to assess a storage systems' performance with regards to data management and compare different storage frameworks against applications with varying I/O to CPU processing ratios. There are two key methodologies proposed in this work.

(1) Profiling storage frameworks for several aspects of data management

(2) Assessing suitability of a storage framework for performing analytic tasks with different I/O to CPU processing ratios.

Profiling of storage frameworks is done by analyzing the costs involved in the interactions between the distributed components for different applications. These interactions include message exchanges (network I/O), storage (disk I/O) and processing times. Based on the results of the experiments performed, other storage frameworks can also be evaluated for performance efficiency. For instance, the results on HDFS and HBase presented in this paper can be used to assess the performance of MongoDB on same applications by following the same methodologies and benchmark standards.

1.3. Thesis Organization

The rest of this thesis is organized as follows. Section 2 describes a distributed system and its components. Section 3 explores the different types of distributed storage systems available and their main characteristics. Section 4 contains a survey of related work. Section

5 details out the methodology used for the analysis and the applications examined. Section 6 puts forth the results obtained from the experiments and Section 7 presents the conclusions drawn from this work. We conclude this report with future works in Section 8.

CHAPTER 2

Hadoop MapReduce

Apache Hadoop [1] is the framework for distributed storage and distributed processing. It can handle very large data sets by distributing the data and processing over multiple commodity hardwares. Hadoop is composed of four main modules:

- *Hadoop Common*: This contains all the libraries and utilities required for other Hadoop modules.
- *Hadoop Distributed Storage System (HDFS)*: This is the distributed file storage system. Through HDFS, data is stored over multiple commodity hardwares.
- *Hadoop YARN*: This is the resource-manager for Hadoop. This module manages the resources in Hadoop clusters and schedules the jobs and tasks.
- *Hadoop MapReduce*: This is the programming model which is used for large scale data processing.

Hadoop can be integrated with other frameworks or applications as well. For example, a different distributed storage system can be used in place of HDFS such as MongoDB [9], HBase [4] and Cassandra [10]. Hadoop can be integrated with data processing engines such as Hive, Pig and Spark which render it with additional features. Hive [11] is a data warehouse infrastructure that provides data summarization and ad-hoc querying. Pig [12] is a high-level data-flow language and execution framework for parallel computation. Spark [13] provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation. It is a fast and general compute engine for Hadoop data. There are many other applications which can be incorporated along with Hadoop to enhance its features.

In a relational database, data is organized as a set of one or more tables of rows and columns. Data in a relational database is analyzed using query languages. Structured Query Language (SQL) is the most commonly used query language for these databases. HDFS is a file system and not a real database. Though it provides services for storage and retrieval of data, same as a database, there are no queries involved. Hadoop is more of a data warehousing system. So it requires a framework such as MapReduce to process the data.

MapReduce [5] is capable of processing massive amounts of unstructured data in parallel across a distributed cluster of nodes. A MapReduce program is comprised of two types of procedures: Map tasks and Reduce tasks. Each Map task or Mapper spawns a map function which takes in a value as input and spawns out intermediate key-value pair(s). Each map task is deployed on the node which stores part of the input split data that it is responsible for processing. This maintains data locality resulting in very high aggregate bandwidth across the cluster. If local computations are not possible, the map task is executed on a different cluster node that is closest to the data. In general, this situation is not so common as most distributed systems replicate the data while storing. All map operations are independent of each other and operate in parallel. After generating the key-value tokens, the map functions group the tokens by their key values. The set of all keys is partitioned by either a default partitioner function which uses hashing or by a custom partitioner as defined by the user. The keys are partitioned into same number of boxes as the number of reducers. A Reducer or Reduce task fetches, from all the Mappers, the key-value tokens that it is responsible for and applies the reduce function on them. The reduce tasks also run in parallel but there is no enforced data locality for the reduce tasks. Each Reducer writes the final output as key-value pairs to the file system.

CHAPTER 3

Distributed File/Storage Systems

We all know that the amount of data in the world is exploding. According to International Data Corporation (IDC) [14], a prominent market research, analysis and advisory firm, the world's information now approximately doubles about every year and a half. In 2011, there was a total of 1.8 zettabytes of data generated and stored world-wide and in 2012, this number was 2.8 zettabytes. IDC predicts that by the year 2020 the world would have generated 40 zettabytes of digital data. This statistics show the growth in amount of data and, along with it, storage needs. When dealing with huge data, typically in petabytes or higher, it is not feasible to store all the data in one disk. For one, there is high risk of losing all the data in case of disk failure. Also, the access bandwidth would decrease with the increase in storage. So rather than scaling-up, the better solution is to scale-out. This means that instead of building and investing in large, expensive and complex hardware with more and more storage capacity, it is better to add low cost commodity hardware to increase storage capacity incrementally. In other words, we need to distribute the storage to different machines/ disks instead of overloading one machine.

A very simple definition of Distributed Storage is storing data in multiple computers or in computers that are geographically dispersed. It offers the advantages of centralized storage with the scalability and cost base of local storage. Centralized Storage or Storage-Area Network (SAN) [15] is a dedicated high-speed network with multiple storage units that communicate with each other and give access to multiple users simultaneously. Users of a distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations. Though there are numerous distributed storage systems such as Hadoop Distributed Storage System (HDFS), Google File System (GFS) [6], Cassandra [10], HBase [4] and MongoDB [9], they all try to implement the following basic design principles:

Availability: Server crashes or hardware failures should ideally not affect the data accessibility in a distributed system. For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response [16]. A system should be have high availability as the costs of down-time can be huge in these cases. One possible and most widely used way to achieve this is using *Data Replication*. Multiple copies of the data are maintained on different systems. So even if one system fails temporarily or permanently, data can be fetched from a replicated server.

Consistency: Consistency means all nodes should see the same data at the same time. If the data has been replicated, as is the case in almost all distributed systems, then all the replicas should be consistent with each other. If a change has been made to a data item on one machine, it takes some time to propagate the change to all the other replicas. During this short time, the copies of the data will be different from each other. But eventually the change will be propagated to all the copies and the system will attain consistency. This is known as *eventual consistency*. Various distributed systems tackle consistency in different ways as there is a trade-off between availability, consistency and speed.

Scalability: A distributed system should be able to handle the addition of users and resources without suffering a prominent increase in complexity or loss of performance. Scalabilty is important as its preferable to scale-out than scale-up, if required.

Security: Due to the multiple systems and users being involved, security threats become even more prevalent in distributed systems. As such, it becomes very significant to prohibit illegal access of data and protect systems from malicious attacks. This is done using different methods of authentication, authorization and encryption.

Transparency: Transparency hides the fact that resources are physically distributed across multiple systems. All aspects of the distributed system design such as access, location, replication, concurrency and failure should be invisible to the user. To the user, the whole distributed system should appear as a single system.

3.1. Distributed Database Storage Models

Distributed Storage Systems can be broadly divided into four categories based on their design and access patterns: Key-Value Storage, Document Storage, Column Family Storage and Graph Storage. Most of the current distributed storage systems will fall into one of these four categories. All these storage models provide different ways of accessing data.

3.1.1. Key-Value Storage Model

SQL, the Structured Query Language is the standard language for relational databases. It can handle structured data efficiently and can execute arbitrary and highly dynamic queries. But the focus is shifting more and more towards object oriented way of thinking. Data to be stored is not being modeled for a database but for logical integrity so as to help in understanding the software infrastructures. An object-based storage manages data as objects, where an object typically includes the data, some variable amount of metadata and a unique identifier. For object-based storage, the ability of SQL to process arbitrary dynamic queries is rendered useless. As such SQL becomes inefficient for such systems and we need a different model to manage data storage. In response to these changes, new ways to store data have emerged that allow data to be more unstructured, schema-less and grouped together more logically and naturally.

In a Key-Value store [17], data is stored as a collection of key and value pairs, such that each possible key appears just once in the collection. The data structures in a relational database is a pre-defined series of tables with fields of defined data types and all the records have the same format. Key-Value stores provide more flexibility in modeling the data. There is no defined structure and each record may have different fields associated with it. Since the data is indexed using a single unique key, retrieval can be random. This means that to retrieve one specific data or range of data, the whole collection need not be accessed. The system can directly lookup the required record(s) using the key.

Some of the most popular distributed Key-Value storage systems are Redis [22], Amazon's Dynamo [23] and Cassandra [10].

3.1.2. Document Storage Model

Where SQL is good for structured data and Key-Value stores perform well for unstructured data, Document-oriented Database [18] are effective for semi-structured data. Though they can be treated as a sub-class of the Key-Value storage system, they have gained much popularity and are used as extensively. Document-based storage systems also have metadata and a unique identifier associated with each data item, same as Key-Value stores. The difference between the two is in the processing of the data. In Key-Value store, data is considered to be inherently opaque to the database as their is no structure to the data. In Document-based stores, the data has an internal structure which is useful while extracting the metadata for further optimization. This internal structure is comparable to an XML document structure. The retrieval process is similar to that of Key-Value store where the key is used to lookup the record. Typically, the keys are indexed and the indexes are used to speed up the retrieval. In addition, utilizing the document structure of the data, only specific parts of the document can also be retrieved.

MongoDB [9], Microsoft's DocumentDB [24] and CouchDB [21] are few examples of D5ocument-oriented storage systems.

3.1.3. Column-Family Storage Model

In a Column-Family Storage System [19], data tables are stored as sections of columns of data instead of the general way as rows of data. Columns which will be usually accessed together are grouped together into one column family. Each column is a part of one and only one column family. This is done to avoid duplication of data. Row-based storage systems can efficiently return an entire row data. As such they are not suitable for operations covering the entire data-set. For instance, to retrieve records that meet some condition on a column, the entire data-set would have to be combed through to get those specific rows of data. The same operation in a column-family storage system would access only one specific column and then return all the rows matching the specified condition.

In this storage system, all values of a Column-Family are serialized together, then the values of the next column-family and so on. Data is indexed by row key, column key and timestamp. The data retrieval in such systems in performed column-wise. It is more efficient when data processing requires many rows but only a subset of columns. Google's BigTable [25] and Apache's HBase [4] are the most popular distributed systems implementing Column-Family storage. We will analyze HBase in more detail in the next sections.

3.1.4. Graph Storage Model

With the recent advances in machine learning and data mining, Graph Storage [20] is becoming more and more popular. Graph databases are based on graph theory and have three entities: nodes, properties and edges. Nodes are analogous to entities or objects. Properties are pertinent information related to the nodes. Edges are connections from nodes to nodes or nodes to properties and represent the relationship between the two. Each node, property and edge in the system is represented by a unique identifier. Every node comprises of a set of outgoing and/or incoming edges and a set of properties expressed as key-value pairs and each edge has a starting and/or ending node and a set of properties. Graph storages are useful for mining data from social media as they can efficiently analyze connections and relations between entities. Graph storage systems do not require index look-ups. The edges act as direct pointers between adjacent nodes.

A few examples of distributed graph storage systems are FlockDB [26], InfiniteGraph [27] and OrientDB [28].

3.2. Distributed File Systems

We discussed different types of distributed storage models and their design principles. There is another class of distributed storage which is Distributed File Systems. These are network file systems where the server is distributed over multiple machines which may or may not be geographically dispersed. It also allows access to files from multiple hosts. It can be seen as a distributed implementation of the classical model of a file system, where multiple users share files and storage resources.

The difference between distributed file systems and distributed storage systems is analogous to the difference between a local file system and a local mySQL database. In distributed file systems, files are broken down into smaller chunks of files, if needed, and then distributed over all the nodes in the system to be stored in their local storage.

Google File System [6], Hadoop Distributed File System [2], Sun Network File System [30], Andrew File System [29] are few of the numerous distributed files systems developed. We will discuss more about Hadoop Distributed File System in the upcoming sections.

3.3. Choosing a Suitable Storage System

In the previous sections, we have seen how many different types of storage systems are available today. We have also established that as the data storage or processing needs grow, it is preferable to use distributed storage and computing rather than putting the whole load on a single machine. But how do we decide which storage system to use and which computing technique to use? We cannot generalize that Column-Family storage systems are better than Key-Value storage systems or that Distributed File Systems outperform Document storage systems. The performance of storage systems is dependent on the application. For different applications, different systems might perform better. For instance, if suppose we need to access only a very small subset of the data, then some distributed storage system would be a better choice than a distributed file system which allows only linear sequential access but if all the data needs to be accessed, then a distributed file system might outperform the rest.

The design of the application also plays an important role in performance of the system. While designing applications, the architecture of the underlying storage subsystem must also be considered as it would effect the interactions between the components for data retrieval and processing. For instance, to do text processing on a text document where each word in the document has to be processed individually, retrieving a line or paragraph at once would still be more optimal than retrieving each word separately. Data retrieval has overhead charges associated with it.

To answer the question of which storage system would be ideal for a specific type of application, we compare their performances against different types of applications. Each application is designed to optimize the performance In this work, we have done a performance analysis of HDFS and HBase on two types of applications: I/O bound and compute bound.

For each of the storage systems, applications have been designed to optimize performance on that storage system, keeping the basic algorithm intact.

CHAPTER 4

Related Work

HDFS and HBase, have both been developed by the Apache Software Foundation. HDFS has been inspired by and is comparable to Google's GFS [6] and was designed to store large files across multiple geo-diverse machines. Similarly, HBase is a distributed non-relational database modeled after Google's Bigtable [25]. Both HDFS and HBase are open-source projects. Along with storage systems, we need distributed computing to utilize the distributed nature of the data storage. This is achieved using MapReduce [5], an open-source programming model developed by Google Inc. Apache also developed it's own distributed computing programming model, based on the same idea, called Hadoop MapReduce [31].

MapReduce [5] provides an infrastructure for executing large-scale data processing jobs on multiple cluster machines. It exploits the processing capacity of the computing clusters by distributing the work load over the clusters. Partitioning of input data, scheduling of executions across the cluster nodes, coordinating between the nodes and node failures is all handled by MapReduce. It is a framework that provides automatic parallelization and distribution of large-scale computations. Hadoop MapReduce was developed on the same concepts.

The primary purpose of HDFS is to store very large data-sets and to stream those data-sets to user applications at high bandwidths. By distributing the storage across multiple cluster nodes, scalability can be achieved at economical rates. In 2006, when Hadoop was first developed and deployed at Yahoo!, it span over 25000 servers and stored 25 petabytes of data. As per IDC's most recent report around 32 percentage of enterprises have deployed Hadoop and another 36 percentage are planning to do so.

Shvachko, Kuang, Radia and Chansler [2] give a detailed description of HDFS and its architecture. The HDFS namespace follows hierarchical structure of files and directories. Files are split into large blocks (by default, 128 megabytes) and each data block is replicated at multiple DataNodes (by default, three). The NameNode is responsible for maintaining the namespace metadata and mapping of file blocks to DataNodes. Blocks in DataNodes are stored in the native file system of the local host. For user applications to interact with HDFS, they require HDFS Client, which acts as the interface between users and HDFS. For reading a file from HDFS, the HDFS Client requests locations of the corresponding data blocks from the NameNode. The NameNode sends the list of all DataNodes hosting replicas of the data blocks. The HDFS Client then reads the data directly from the DataNode closest to it. For writing a file to HDFS, the HDFS Client splits the file into data blocks and, for each data block, requests the NameNode to assign DataNodes to host the replicas. The Client then establishes a pipeline to the required DataNodes and sends the data blocks. Each DataNode sends periodic *heartbeats* to check in with the NameNode and also informs the NameNode about the data blocks that it hosts.

Hadoop Distributed File System has write-once read-many access semantics. Shafer, Rixner and Cox [32] analyzed the interactions between Hadoop and HDFS, and described the performance bottlenecks in the filesystem. Due to the structural implementation of Hadoop causing delays in scheduling new MapReduce tasks, HDFS is not utilized as efficiently as possible. In HDFS, the access pattern for disk is periodic and prefetching is not employed. HDFS is written in Java and hence some performance-enhancing features of the native platform are not exploited. As such, the HDFS implementation runs less efficiently and has higher processor usage than would otherwise be necessary. Also, though HDFS is portable, its performance is highly dependent on the behavior of underlying software layers. These performance bottlenecks can be overcome by improving I/O scheduling, adding pipelining and prefetching to task scheduling and HDFS clients, pre-allocating file space on disk, and modifying or eliminating the local filesystem [32].

HBase [4] is a fault-tolerant, highly scalable, NoSQL distributed database built on top of HDFS. HBase only contains information about the storage locations of the data blocks whereas the actual data resides in HDFS. HDFS, because of its file system structure, can allow only linear access to data. On the other hand, HBase can be used for real-time random read and write operations on large data-sets. HBase also employs master-slave architecture, similar to HDFS. HBase has a master server (HMaster), analogous to the NameNode of HDFS, and numerous data servers (HRegionServers) as slaves, similar to the DataNodes in HDFS. Scheduling and management of resources is done by Zookeeper [3]. Each row in a HBase table has a unique sorting key and arbitrary number of columns. The table cells are versioned by timestamp which is assigned during insertion. Because of this, a column can have several versions for the same row key. Each cell has four identifiers: Table name, Row-Key, Column-Family and Column name, and Timestamp. HBase tables are split into regions according to row keys and distributed among the HRegionServers. Regions are further divided vertically by column-families and stored as files in HDFS.

Parallel databases [33] seek to improve the performance of relational database management systems through parallelization of various operations involved with a database such as loading data and evaluating queries. Both MapReduce and Parallel databases process data in parallel using multiple systems. Where MapReduce jobs express the problem in terms of map and reduce functions, Parallel databases organize their data in the relational data model using collections of tables. These two systems are competing and complimentary to each other. Parallel databases scored high on performance and MapReduce is more flexible in handling unstructured data. In [34], Mchome describes how data analysis is affected by data organization and querying structure in both Parallel databases and MapReduce programming models.

Before choosing between the different distributed storage models as the ideal system for a particular application, it is recommended to analyze whether a distributed system would be ideal for the application or a parallel database. McClean, Conceicao and O'Halloran [34] provide a high level comparison between MapReduce and Parallel databases and present a selection criteria to choose between the two for a particular application. MapReduce performs better for unstructured data and Parallel databases for structured data. Also, the cost for MapReduce beats the cost for using Parallel databases at enterprise level.

A comparative analysis of GFS and HDFS is done by Vijayakumari, Kirankumar and Rao [8]. They provide the similarities and dissimilarities between the two in terms of architecture and properties such as scalability, security, cache management, communication, replication strategy, etc. HDFS and GFS are similar in more ways than not. Both have a cluster based master-slave architecture, have hierarchical file structure and support batch processing. GFS uses TCP for communications and HDFS uses RPC based protocol on top of TCP/IP. GFS uses Bigtable as its database and HDFS uses HBase. These are some of the similarities and dissimilarities between the two distributed file systems. HDFS, being an open-source project, is more popular and widely used in different enterprises, whereas GFS is owned and used by Google Inc.

In 2010, Facebook shifted from Cassandra, it's in-house built distributed database, to HBase. The reasons, as indicated by Borthakur, Sarma, Gray, Muthukkaruppan, Spiegelberg et al. [35], for moving away their MySQL-based architecture was that it is difficult and not optimal to scale some workloads because of very high throughput, massive datasets, unpredictable growth or other patterns. Facebook Messaging, Facebook Insights and Facebook Metrics System were the applications that were over growing the powers of RDBMS. The requirements for a new storage system and the reasons for choosing Hadoop and HBase were elasticity, high write throughput, efficient and low-latency strong consistency semantics within a data center, efficient random reads from disk, high availability and disaster recovery, fault isolation, atomic read-modify-write primitives, and range scans.

CHAPTER 5

Methodolody

Storage frameworks involve interactions between the distributed components. These interactions pertain to ensuring consistency, fault tolerance, load balancing, and preserving high throughput. These interactions entail message exchanges (network I/O), storage (disk I/O), updates to data structure (memory) and processing encompassing several outputs (CPU). How and when these interactions are performed impact efficiency of the processing that is performed on the underlying data. We profile the costs (disk I/O, network I/O, and CPU) associated with these interactions, and identify their impact on different applications that have a slightly different mix of CPU and I/O processing.

5.1. Profiling

The systems are profiled on costs involved for disk I/O on each node in the cluster, network I/O between all pairs of nodes and run-time for the application.

5.1.1. Disk I/O Monitoring

For a CPU to process the data, it requires the data to be present in memory. For this, data is fetched from disk and cached in memory. If the disk I/O operations are lagging, the processor sits idle till the data is cached in memory. Hence, reading and writing to disk has an impact on the processing of an application. Using Linux disk monitoring tools, we gather the number of bytes of data read from and written to disk in each node of the cluster, during both staging and processing of the data.

5.1.2. Network Monitoring

For a system comprising of multiple machines, it is important to ensure smooth flow of information between nodes as the functioning of one system is dependent on others as well. Also, a network bottleneck can also delay the execution of a process.

We used Wireshark tool to perform network analysis on the cluster. Wireshark is an open-source network packet analyzer that does network analysis at a microscopic level. It is similar to tcpdump in functionality. It captures, logs and analyzes data transferred between and over a network. One significant advantage of Wireshark over other network analyzers is that it can record traffic on a network and store it for analysis later on. Captured network data can be browsed via GUI, or via the TShark utility.

TShark enables us to capture the network traffic data directly into a file which can be processed later. Table 5.1 shows a sample of data captured using Wireshark utilities. Here, *A* and *B* denote the nodes between which the network traffic has been captured. *Address A* and *Address B* are the IP addresses of the nodes and *Port A* and *Port B* are the ports through which the data was transmitted or received. *Packets* and *Bytes* column give us the total number of packets and total number of bytes that went to and fro between nodes *A* and *B*. Wireshark also gives us the direction specific data: number of packets and bytes transmitted from node *A* to node *B* and vice versa (*Packets A→B, Bytes A→B, Packets A←B, Bytes A←B*). It also gives us the rate of transmission both ways (*bps A→B, bps A←B*), the relative start time of that particular capture (*Rel Start*) and the duration of the capture (*Duration*).

We are interested in finding the total amount of data transferred between all pairs of nodes in the cluster. For this, we run the TShark utility on all the nodes in the cluster, including the NameNode, Secondary NameNode, Data Nodes and Region Servers. For each pair of nodes A and B, we gather the data from the captured files at both A and B and we then take the average of these values. This is done for all pairs of nodes in the cluster.

Table 5.1. Sample Wireshark Captured Data

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A→B	Bytes A→B	Packets A←B	Bytes A←B	Rel Start	Duration	bps A→B	bps A←B
129.82.46.32	35037	129.82.46.31	issd	13059	95002040	6716	94564799	6343	437241	29.593103000	2.0770	364236105.92	1684125.18
129.82.46.33	38296	129.82.46.31	issd	17872	132853214	9669	132293731	8203	559483	868.552229000	1.2155	870717247.01	3682347.56
129.82.46.34	48994	129.82.46.31	issd	18532	136572331	9910	135984654	8622	587677	874.219346000	1.6478	660213022.84	2853204.37
129.82.46.37	35242	129.82.46.31	issd	17093	126548879	9072	125994587	8021	554292	121.693596000	2.0454	492791488.42	2167953.29

5.1.3. Application Run Time

Run time is the time taken for fully processing an application from start to end. It includes any setup time taken by Hadoop before starting the actual MapReduce processes. The Staging time is not included in the application run-time but is taken into consideration separately. This is the time taken to stage the dataset before running the applications. It includes the time to load the input data into the file system and distribute it among the cluster nodes.

5.2. Applications

In this section, we discuss the applications that have been analyzed, algorithms involved, mapReduce implementations for these applications and datasets used. In general, most applications are either compute bound or I/O bound. We take one I/O intensive application and one compute intensive application to perform our experiments and analyze the storage system's performances in both cases. For an I/O bound job, we perform Word Frequency Count and for compute bound job, we perform K-Means Clustering. Since HDFS, being a file system, lacks random read and write accesss, we did not utilize this feature of HBase. So as to ensure fairness in the comparison, the datasets were accessed linearly for both HDFS and HBase.

5.2.1. Word Frequency Counter

The Word Frequency Counter counts the frequency of all distinct words in a document. One of the applications of Word Frequency Counter is during sentiment analysis. Sentiment analysis refers to the use of language processing and text analysis tools to identify and extract subjective information in source materials. It determines the attitude of the writer on some topic or the overall contextual polarity of the document [wiki]. The applications for sentiment analysis are endless. It is used extensively in social media monitoring and customer-oriented business analytics.

In this algorithm, the mapper reads the input text and generates the key-value pairs of distinct words as keys and their corresponding frequency counts as values. In the Hadoop implementation of this algorithm, the mapper functions read the input text data and emit each encountered word as key with value as one. The mappers also do some preprocessing of the data. All the special characters are removed and all the characters are converted to lower case as we do not want the final output to be case sensitive. For example, for the input sample text ”Colorado State University#” the output from the mapper would be (colorado,1), (state,1) and (university,1). All the instances of the same key go to the same reducer irrespective of at which mapper it was generated at. At the reducer, the results for each word are summed up to get the final count of that word in the whole text document. The outputs from all the reducers are combined to get the final output of the program.

For this application, test data was built by combining different books available under Project Gutenburg, which is the online repository of free e-books. Ten different files were created with the size of each file being between 10-15GB. The total size of all the 10 files combined together was 126GB.

5.2.2. K-Means Clustering

K-means Clustering [36] is an unsupervised learning methodology for grouping data points into clusters. It partitions n observations into k clusters in which each observation belongs to the cluster with the nearest mean. The mean of all the data points in a cluster serves as a prototype of that cluster and all the points in that cluster should be closest to this mean than to the mean of any other cluster. The clusters formed using this algorithm are non-hierarchical and they do not overlap with each other.

To compare two observations, we need a notion of similarity or dissimilarity between the two. If the observations are numerical or can be equated to numerical values, we can take the distance between two observations as their dissimilarity index. There are many distance measures available such as Manhattan Distance, Euclidean Distance and Cosine Distance. For our experiments, we will be using Euclidean Distance measures. If the observations or items are comparable to points in an d-dimensional space, where d is the number of attributes in each observation, then we can take the Euclidean distance as the metric of dissimilarity. The Euclidean distance between two points is square root of the sum of the squares of the differences between the corresponding coordinates of the points.

The K-Means Clustering algorithm takes in two inputs: the d-dimensional data and k initial cluster centers. The k initial cluster centers are randomly picked from n observations of the data, without replacement. For all the experiments, same initial clusters are taken so as to maintain uniformity in the iterations. The aim of this algorithm is to group the n observations into k clusters such that all items in same cluster are as similar to each other as possible and items not in same cluster are as different as possible. Each cluster has a centroid which is the most representative point of the whole cluster. For numerical data, as is the case here, we can take the mean of all the points in the cluster to be the cluster center.

The first step of the algorithm is to, for each of the n points, find the nearest center among the k cluster centers and assign it to that cluster. Then the cluster centers are updated by taking the mean of the points in the cluster. In the next iteration, the same steps are repeated with the new set of k cluster centers. These iterations proceed in a loop till there is no change in any cluster between two consecutive iterations.

Though sometimes this algorithm may take exponential time, it always converges. The time complexity of the sequential K-means clustering algorithm is $O(ndki)$ where n is the number of observations in the data-set, d is the dimensionality of the data, k is the number of clusters and i is the number of iterations the algorithm executed. So this algorithm is very compute intensive.

In MapReduce, we take the same approach but distribute the work between multiple Mappers and Reducer. Each iteration of the algorithm is executed by a new MapReduce job. Each Map task receives a portion of the n data points as input and reads the k cluster center points from a common location of the file system. The map function calculates the similarity/ dissimilarity index for each of its observations against the k centers and outputs the closest cluster center as key and the data point as value. There is only one Reducer in this application. The Reducer receives all the cluster center and data point key-value pairs and for each center, calculates the new cluster center by taking the mean of all the data points with that particular center key. The MapReduce tracks the number of cluster centers that have converged as a parameter in its configuration. The Reducer checks the convergence of each cluster center with its new center and reports the final count of number of convergences to the Job. The Reducer then updates the values of the cluster centers stored in the file system. After this Job completes, the program starts a new instance of the same Job if the number of converged centers is less than k. This program ends when all the centers have converged with the last run centers.

For this application, test data was generated randomly with a mix of Integer, Float and Boolean values. The test dataset contained 10 million points with 10 attributes each. The number of classes to cluster the data into was taken as 10.

5.3. Hardware Specifications

For both HDFS and HBase, a ten node cluster was formed. The NameNode, Resource Manager and HMaster servers were run on different nodes. The same ten systems were used as the DataNodes for HDFS and as HRegionServers for HBase. On the whole, thirteen systems were employed for the experiments. All the thirteen systems were HP Generation 6 machines running on Linux (Fedora) Operating System.

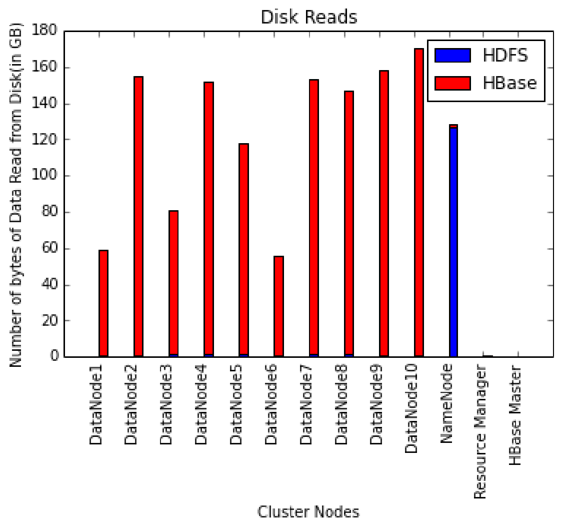
CHAPTER 6

Results

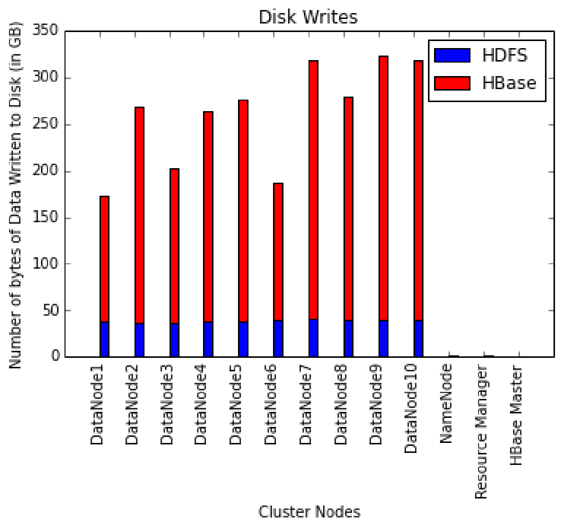
We show the performances of HDFS and HBase for two applications: Word Frequency Counter and K-Means Clustering. A subjective comparison is done based on disk I/O, network I/O, staging time and application run time.

6.1. Word Frequency Counter: HDFS vs. HBase

When staging the data-set in HDFS, the total number of bytes of data read from the local disk of all cluster nodes is 136GB. Out of this, 126GB of data is read from a single node as that node hosts the original input file. As for the disk writes, 383 GB of data is written across all the nodes and this is evenly distributed among all the DataNodes. The disk writes is three times the size of input data as the replication factor is, by default, three for HDFS. For staging a data-set in HBase, the data-set should first be loaded into HDFS as the data in HBase actually resides in HDFS and HBase just stores the locations of the data blocks. But HBase is a NoSQL distributed database whereas HDFS is a distributed file system. So, to enable random access to data, it requires the location of not just data blocks but of each record as well. Since HBase is a Column-Family storage system, it needs to write the data back to HDFS in a column-oriented structure. This process generates a lot of disk I/O. Among all the nodes in the cluster, the total number of disk reads is 1230GB and the total number of disk writes is 2217GB. Figure 6.1 shows the comparison between HDFS and HBase in disk reads (Figure 6.1a) and disk writes (Figure 6.1b) at each node in the system. From these plots, we can see that, as expected, the disk I/O is negligible for the HDFS NameNode, YARN Resource Manager Node and HBase Master. The number of disk reads on NameNode is an exception here because the the original input file was stored in local disk of the NameNode.



(A) Disk Reads



(B) Disk Writes

FIGURE 6.1. Disk I/O during staging of Word Frequency Counter application

Figure 6.2 shows the network I/O between pairs of nodes in descending order of the network traffic. We are showing the network traffic between the top 60 pairs of nodes as the rest of the pairs have very negligible network I/O between them. Network traffic in HBase is not as drastically higher than HDFS as was disk I/O. This shows that there is one on one link between the HDFS DataNodes and the HBase RegionServers and HBase maintains the same data locality as HDFS. Figure 6.2 also shows the staging times of the two systems which is around 43 minutes in HDFS and 133 minutes in HBase. On the whole, staging a data-set in HBase is much costlier than staging it in HDFS in terms of disk I/O and staging time.

For the Word Frequency Counter application, the total number of disk reads in HDFS is 128GB and that in HBase is 191GB. The distribution and comparison of disk reads between the two systems is shown in Figure 6.3a. The disk reads are quite evenly distributed among

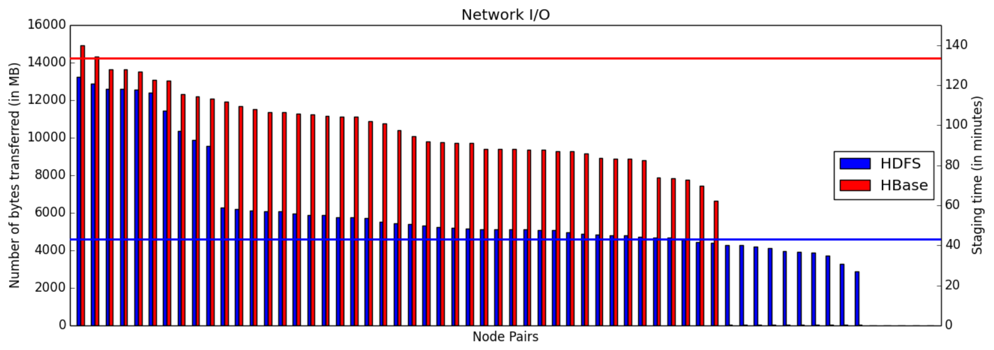
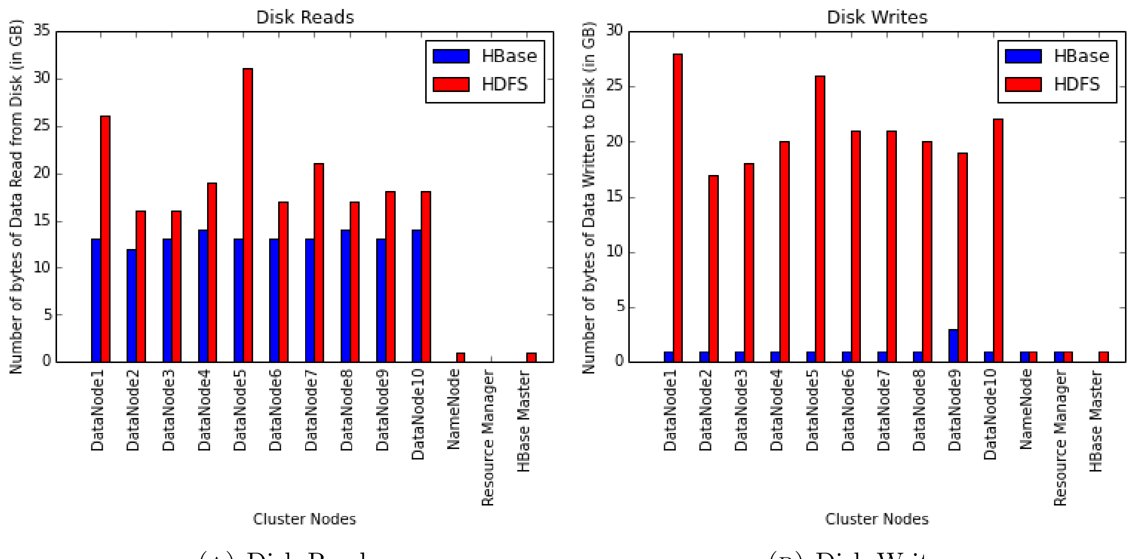


FIGURE 6.2. Network I/O during staging of Word Frequency Counter application

the data nodes which ascertains the load balancing capabilities of these storage systems. As for the disk writes, we can again see the drastic difference as was observed during staging. HDFS writes a total of 8GB to disk as output and HBase a total of 205GB. Though both HDFS and HBase follow 'write-once read-many' approach, writing to HBase is much more costlier in terms of disk usage and time.



(A) Disk Reads (B) Disk Writes

FIGURE 6.3. Disk I/O for Word Frequency Counter application

Figure 6.4 shows the network traffic distribution and the application run time. The network traffic for HDFS is very less as it enforces data locality while running applications.

In HBase, the highest amount of network traffic between two nodes is 24GB. Though this is more than that in HDFS, it is not inconsiderably high. The interesting results found in this application are the application run times. HDFS takes 39.22 minutes to complete this program whereas HBase takes only 36.33 minutes. This is because HDFS processes the input data blocks line by line and in HBase, the words are already separated into different columns and can be accessed column-wise.

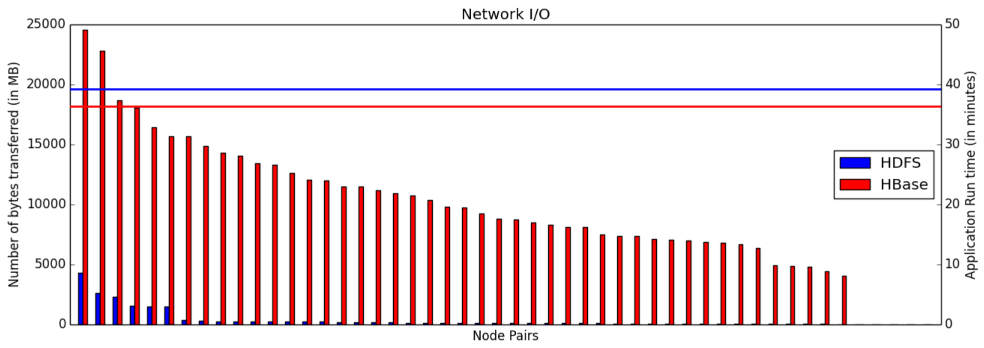
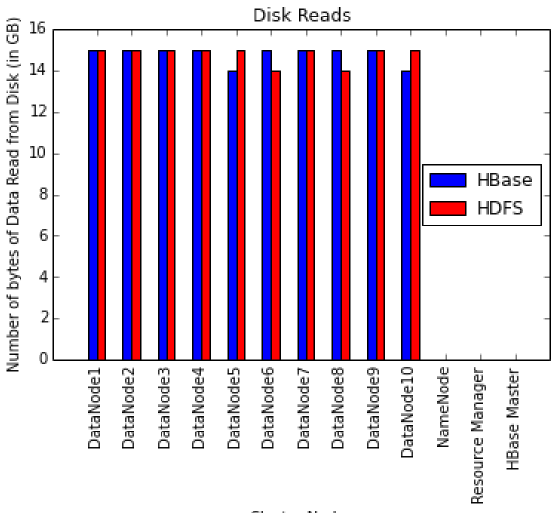


FIGURE 6.4. Network I/O for Word Frequency Counter application

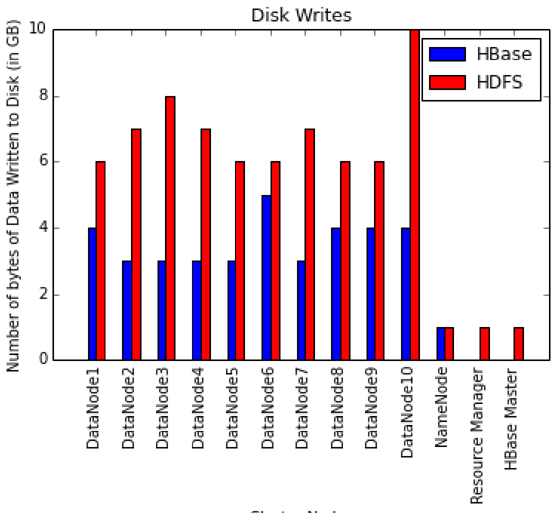
For an I/O bound application like the Word Frequency Counter, the overall performance of HDFS is better. Even though the application run time for HBase is faster, the staging time and amount of disk I/O and network I/O consumed by HBase makes HDFS a better choice.

6.2. K-Means Clustering: HDFS vs. HBase

There are two input files for K-Means Clustering program. The first contains the ten million input vectors and the second file has the ten initial centers. The input vectors file is 327MB in size and the file with initial centers is 326B. Because the input files are so small in size, the staging will not be of significance here. K-Means Clustering is a compute-bound application. For the sample data-set and the initial centers, the map-reduce job runs 80 times iteratively for convergence. We compute the disk I/O and network I/O for all the 80 runs together.



(A) Disk Reads



(B) Disk Writes

Figure 6.5. Disk I/O for K-Means Clustering application

Figure 6.5 shows the disk reads and writes on all the nodes in the cluster. From Figure 6.5a, we can see that the disk reads in HDFS and HBase are much the same. The disk writes for HBase are slightly more than that of HDFS (as shown in Figure 6.5b). Again, this is expected as HBase uses more resources for write operations than HDFS.

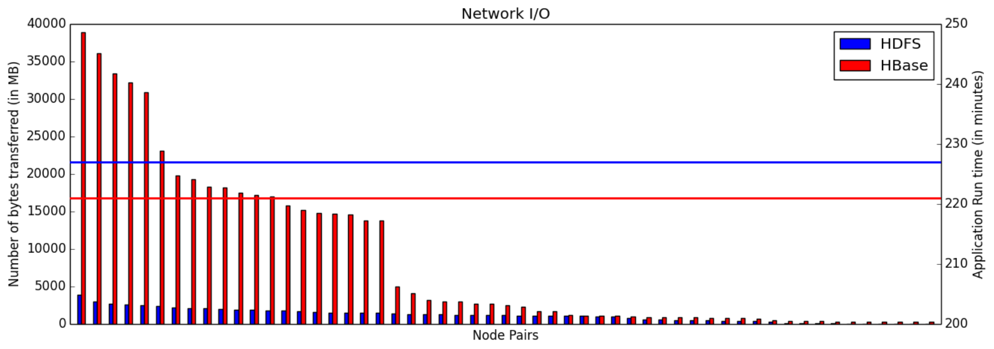


Figure 6.6. Network I/O for K-Means Clustering application

The network traffic generated by HBase is 472GB whereas HDFS generates only 65GB (as shown in Figure 6.6). But the application run time for HBase is faster than that of HDFS. For the 80 iterations of the algorithm, HDFS takes 227 minutes and HBase takes only 221 minutes. This means that for each iteration, HBase is 5.25 seconds faster than HDFS. Though this might seem meager for one run, for large computations it makes a significant difference. For larger input files and/ or more number of of classes to be clustered into, K-Means clustering can even take thousands of iterations for completion. Also, the time complexity of this algorithm is directly proportional to both the number of input vectors and the number of classes. So, an increase in either would also increase the run time of each individual iteration.

Based on the disk I/O, network traffic and application run time analysis for the K-Means clustering application on the sample data, we can say that for compute-bound applications with inconsequential I/O requirements, HBase is more efficient than HDFS.

CHAPTER 7

Conclusion

Distributed File Systems and NoSQL databases are fast becoming the standard data platform for storing large datasets. While there are many distributed storage systems in the market, there is not a single universal top storage system outperforming the rest for all applications. Depending on the requirements of the application, the I/O and network communications involved, the structure of the data-set and deployment conditions, it is almost always possible for one storage system to outperform another and lag behind the same when the rules of engagement change.

While it is recommended to assess a system's performance for specific use cases and environments, we can postulate based on the performance analysis of a similar system. For instance, the performance of Hierarchical Clustering, a connectivity based clustering algorithm, would be very similar to the performance of K-Means Clustering application against different storage systems.

From the Word Frequency Counter results, we can infer that, for an I/O bound application, HDFS would be a better choice over HBase. HBase consumes much more I/O resources than HDFS and hence for an application that is already I/O intensive, it would put extra burden on the system. Therefore, writing large outputs to HBase should be avoided. If the output is not required to be in a column-family structure, then writing the end result to some other file system such as HDFS would make a significant improvement in the performance of HBase in terms of the disk I/O and network I/O.

The K-Means Clustering application's performance with HBase is better than with HDFS. This is a significantly compute-bound application with comparatively negligible I/O involved. In most real time applications of K-Means, not every attribute of the input vector contributes towards the clustering. In such cases, where only a smaller set of attributes from a larger set will be utilized, performance of HBase would improve significantly, as the data in HBase is stored and accessed column wise.

The storage architecture and data access patterns play an important role in determining the performance of a distributed system.

CHAPTER 8

Future Work

In this work, we have analyzed the performance implications of using HDFS and HBase as the storage system for Word Frequency Counter and K-Means Clustering applications. This covers one Distributed File System and one Column-Family Storage System. The same analysis can be performed on other types of distributed databases: Key-Value Storage, Graph Storage and Document Storage.

The Word Frequency Counter is a purely I/O bound application and K-Means Clustering a significantly compute-bound application. Additionally, these applications require a specific access pattern. We can extend this work on applications with a mix of I/O and computation restrictions and on applications with different access requirements.

Bibliography

[1] A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY. "Hadoop: a framework for running applications on large clusters built of commodity hardware". *Wiki at http://lucene. apache. org/hadoop*, 2005.

[2] K. Shvachko, H. Kuang, S. Radia and R. Chansler. "The Hadoop Distributed File System". In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1-10. IEEE, 2010.

[3] The Apache Software Foundation. Apache ZooKeeper.

[4] The Apache Software Foundation. Apache HBase, 2012.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters" In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco CA, Dec. 2004.

[6] S. Ghemawat, H. Gobioff and S. Leung. "The Google file system" In *Proceedings of ACM Symposium on Operating Systems Principles*, Lake George, NY, Oct 2003, pp 2943.

[7] M.N. Vora. "Hadoop-HBase for Large-Scale Data". In *Proceedings of 2011 International Conference on Computer Science and Network Technology*. IEEE, 2011.

[8] R. Vijayakumari, R. Kirankumar and K.G. Rao. "Comparative analysis of Google File System and Hadoop Distributed File System". In *Proceedings of International Journal of Advanced Trends in Computer Science and Engineering*, volume 3, pages 553-558, 2014.

[9] MongoDB, Inc. MongoDB.

[10] The Apache Software Foundation. Planet Cassandra.

[11] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu and R. Murthy. "Hive A Petabyte Scale Data Warehouse Using Hadoop". In *Proceedings of International Conference on Data Engineering - ICDE*, pages 996-1005, 2010.

[12] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan and U. Srivastava. "Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience". In *Proceedings of the VLDB Endowment*, volume 2, issue 2, pages 1414-1425, 2009.

[13] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica. "Spark: Cluster Computing with Working Sets". In *Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing*, pages 10-10, 2010.

[14] International Data Group. International Data Corporation.

[15] Wikipedia Contributors. Storage area network.

[16] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In *Newsletter of ACM SIGACT News*, volume 33, issue 2, pages 51-59, 2002.

[17] Wikipedia Contributors. Key-value database.

[18] Wikipedia Contributors. Document-oriented database.

[19] Wikipedia Contributors. Column-oriented DBMS.

[20] Wikipedia Contributors. Graph database.

[21] The Apache Software Foundation. CouchDB.

[22] Redis Labs. Redis.

[23] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazons highly available key-value store". In *Proc SOSP*. Citeseer, 2007

[24] Microsoft Azure. DocumentDB

[25] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. "Bigtable: A distributed storage system for structured data". *ACM Transactions on Computer Systems (TOCS)*, volumne 26, issue 2, article 4, 2008.

[26] Twitter. FlockDB.

[27] Objectivity. InfiniteGraph.

[28] Orient Technologies. OrientDB.

[29] Andrew Project, Carnegie Mellon University. Andrew File System.

[30] R. Sandberg. "The Sun Network Filesystem: Design, Implementation and Experience". In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*. 1986.

[31] The Apache Software Foundation. Hadoop MapReduce.

[32] J. Shafer, S. Rixner, A.L. Cox. "The Hadoop Distributed Filesystem: Balancing Portability and Performance". In *Proceedings of 2010 International Symposium on Performance Analysis of Systems and Software*. IEEE, 2010.

[33] D. Dewitt, J. Gray. "Parallel Database Systems: The Future of High Performance Database Processing". In *Communications of the ACM*, volume 35, issue 6, pages 85-98, 1992.

[34] A. McClean, R. Conceicao, M. O'Halloran. "A Comparison of MapReduce and Parallel Database Management Systems". In *The Eighth International Conference on Systems*, pages 64-68, 2013.

[35] D. Borthakur, J.S. Sarma, J. Gray, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt and A. Aiyer. "Apache hadoop goes realtime at Facebook". In *Proceedings of SIGMOD International Conference on Management of data*, pages 1071-1080. ACM, 2011.

[36] Wikipedia Contributors. k-means clustering.

[37] The Wireshark Foundation. Wireshark.