



郑州轻工业学院
ZHENGZHOU UNIVERSITY OF LIGHT INDUSTRY

Analysis of Optional Architecture
for Big Data Processing

by

Lv Ran

A report is submitted in fulfillment
of the requirements of Bachelor of Engineering
in E-Commerce

Report was produced during the internship at the School of Information and
Communication Technology Griffith University, Gold Coast, Australia

International Educational College
Zhengzhou University of Light Industry, Zhengzhou, China
June, 2015

ACKNOWLEDGEMENTS

I can not express enough thanks to my Supervisor Associate Professor Bela Stantic from Griffith University, and supervisors Professor Min Huang, Professor Xin Chen, and Professor Hongli Song from ZZULI IEC for their instruction, support as well as encouragement. I offer my sincere appreciation and honour for having opportunity to be a Visiting Scholar at the School of Information and Communication Technology, Griffith University, Australia, which based on Students' satisfaction survey was ranked number one ICT School in Australia in 2015.

Sincere appreciation to the study opportunity overseas, which broadened my horizon as well as taught me how to live and study independently. I have made a great improvement in English and I gained a lot of knowledge in professional academic area, this all I could not imagine I would be able before. From the *Research Method* course and the Thesis, I learned a lot and also about very hot topic - Big Data. My study pattern has changed from learning all from teachers and textbooks to do the research and to learn by myself whenever I encounter unknown problems to think critically. It is considerably helpful and useful for my future study and research.

My completion of this thesis also could not have been accomplished without the support and cooperation from my fellows from Big Data Group Nigel Franciscus, Crystal He, and Emily Chen.

Last but not least, I should offer my sincere appreciation to my mother and father, who give me support and encouragement when the times got rough and are much appreciated and duly noted.

CONTENTS

1	Proposal	1
1.1	Problem Definition	1
1.2	Purpose, Requirements, and Expected Achievements	2
1.3	Time line	2
2	Background	4
2.1	Cloud Computing	4
2.1.1	Characteristics of Cloud Computing	5
2.1.2	Architectures of Cloud Computing	6
2.2	Big Data	7
2.2.1	Characteristics of Big Data	8
2.2.2	Introduction of Big Data	9
2.3	NoSQL Database	12
2.3.1	The Basic Principles of NoSQL	12
2.3.2	The Features of NoSQL	13
3	Literature Review	17
3.1	Hadoop	17
3.1.1	Characteristics of Hadoop	18
3.2	Hadoop Component	18
3.2.1	Hadoop Distributed File System (HDFS)	18

3.2.2	Mapreduce	29
3.2.3	YARN	32
3.2.4	Pig and Hive	35
3.3	Stream Processing	36
3.3.1	Spark	36
3.3.2	Storm	38
4	Contributing Chapter	40
4.1	Comparison of Hadoop Version1 and Version2 on YARN	40
4.1.1	Hadoop Version1	40
4.1.2	Hadoop Version2 on YARN	46
4.1.3	Benefits of Hadoop on YARN	47
4.2	Comparison of Processing Engine	48
4.2.1	Benefits of Stream Processing	50
4.2.2	Storm	50
4.2.3	Spark	52
4.2.4	Similarity and Difference between Storm and Spark Streaming . . .	53
5	Conclusion	56
5.1	Summary of Contributions	56
5.2	Future Work	57
	List of References	58

List of Figures

2.1	Cloud Computing	7
2.2	Big Data Value Chain	11
2.3	CAP Theorem	12
2.4	NoSQL and NewSQL comparision I [Grolinger et al., 2013]	16
2.5	NoSQL and NewSQL comparision II [Grolinger et al., 2013]	16
3.1	HDFS framework	22
3.2	HDFS Reading Process	23
3.3	HDFS Writting Process	24
3.4	Data Pipeline While Writing a Block	25
3.5	HDFS Block Replication	28
3.6	Mapreduce Framework	30
3.7	MapReduce Framework with HDFS	31
3.8	Hadoop YARN	33
3.9	YARN Framework	34
3.10	Spark	37
4.1	Poor-utilization Of Reduce	42
4.2	Map Reduce Usage Comparison	43
4.3	Busy Job Tracker On Large Cluster	45
4.4	Streaming Processing VS Batch Processing	50

4.5	Storm Components	51
4.6	Spark Streaming Process	52
4.7	Message Sending Mechanism In Spark	53
4.8	Storm Vs Spark Streaming Comparison	54
4.9	Benchmarking Results	55

1

Proposal

This report will cover the topic on Big Data Processing and requirements which need to be satisfied in order to achieve good performance in accessing information. In following section we will elaborate on topic area, problem definition which will be addressed as well as discuss the purpose of the research, requirements, technology which will be used and also expected achievements. Finally we will provide a time line for this work and conclude report with bibliography assessed.

1.1 Problem Definition

It is evident that the future competitions in business productivity and technologies will benefit from the Big Data explorations. However to be able to obtain valuable information from Big data there are many technological challenges as current concept, both in infrastructure and technologies, cannot cope with volume, velocity and variety of data that will be considered. First of all traditional architectures are not able to cope with demands. With regards to available technologies difficulties are evident in many aspects, specifically

in data capture, data storage, data analysis as well as data visualization. Therefore it is required to look for alternatives both in architectures and technologies which can address the need of Big data. In this work we will at first present state-of-the-art currently adopted to deal with the Big Data problems [Chen and Zhang, 2014].

We will consider Hadoop horizontal scaling with share nothing architecture. Hadoop is able to parallel process massive amount of data and therefore it can be utilized in real time analysis [White, 2009], [Lam, 2010].

1.2 Purpose, Requirements, and Expected Achievements

In this work we will look into different concepts introduced in literature for parallel processing of waste volume of data and perform comparison between them in order to identify which concept is best with regard to different volume, different type and different velocity of data [Frampton, 2015].

Requirement is to have a Hadoop server with sufficient storage and sufficient data in place [Borthakur et al., 2011], [Kim et al., 2014].

Technologies Hadoop [Shvachko et al., 2010], [White, 2009], HDFS,

Mapreduce [Dean and Ghemawat, 2008], dynamicMR [Zikopoulos et al., 2011], YARN [Vavilapalli et al., 2013], MongoDB [Chodorow, 2013],

We expect to identify what would be the best architecture for particular problems depending on data type, volume of data as well as required response time for real time analysis [Tang et al., 2014].

1.3 Time line

- March 02 - 06

Task : Defining the topic and initial Literature review

- March 09 - 13

Task : Introducing to problem area and writing Opening Proposal

- March 14 - 21

Task : Literature review

- March 22 - 31 : Empirically evaluate the performance of the cached directory based cost model.
- April 01 - 07 : Investigate new multidimensional splitting and merging, algorithms related to the packed binary partitioning node.
- April 08 - 21 : Simulate and compare the performance of different concepts and different data partitioning node.
- April 22 - 28 : Empirically test performance of techniques used.
- May 01 - 21: Write up of thesis.

2

Background

In this chapter, we present an overview of some background information including the Architectures of Cloud Computing and Characteristics, Architectures of Big Data, and the principle and features of NoSQL Databases.

2.1 Cloud Computing

Cloud computing is the development of Parallel Computing, Distributed Computing as well as Grid Computing. With the development of the Internet, cloud computing technology is popularly used in processing the growth of massive data in a short time. It can be a platform which has a high fault tolerance, high computing power, high scalability and high storage capacity by using virtual machine, workload balancing, network technology, and distributed file storage. There are three main types of industry clouds: cloud software, cloud platforms, cloud infrastructure.

- **Upper grade: Software as a Service (SaaS).** SaaS breaks the previous monopoly and offers a wide range of software services, which means users can upload ideas

freely. Consumers need not manage any cloud computing infrastructure, such as internet, server, operation system as well as storage. Participants: software developers all over the world;

- **Middle Level: Platform as a Service (PaaS).** PaaS builds application development and operating system platform, which allows developers to write programs and offer services through the Internet. Consumers can run the program through the platform and deploy the application, without managing any cloud computing infrastructure(internet, server, operation system as well as storage). Participants : Google, XenSystem, Apple, and Microsoft, Yahoo;
- **Lower grade: Infrastructure as a Service (IaaS).** The service is to use all of the infrastructure for consumers, including processing, storage, network and other computing resources. Users can deploy and run on any software, such as operating systems and applications, and control the choice and deployment of operating system, while can not manage or control any cloud computing infrastructure. Participants : IBM and Dell.

2.1.1 Characteristics of Cloud Computing

1. **Large-scale:** The scale of cloud is large, for example Google has owned more than one million servers. Even Amazon, Microsoft, IBM, Yahoo have more than hundreds of thousands servers. There are hundreds of servers in a middle-sized enterprise. Cloud computing enlarges the user's computing capacity.
2. **Virtualization:** Cloud computing makes user get service anywhere in any kind of terminal. The resources required come from cloud instead of visible entity, allowing users get all information they want through net service using a PC or a mobile phone. Users can attain or share resources safely through an easy way, anytime, anywhere and complete a task that cannot be completed in a single computer.
3. **High reliability:** Cloud uses data multi-transcript fault-tolerant, the computation node isomorphism exchangeable and so on to ensure the high reliability of the service. Using cloud computing is more reliable than local computer.
4. **High extendibility:** The scale of cloud can extend dynamically to meet the requirement. One cloud can support different applications at the same time.

5. **On-demand self-service:** Cloud is a large resource pool that consumers can buy according to their need and charged by the amount that users used, just like water, electric, and gas.
6. **Extremely inexpensive:** Because of the special character of cloud-fault tolerance, clusters can be built in very inexpensive nodes. The versatility can increase the utilization rate of the available resources compared with traditional system, so users can spend only a few hundred dollars and a few days to accomplish a task which must spend thousands of dollars and several months in the past. [Zhang et al., 2010].

2.1.2 Architectures of Cloud Computing

A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, dynamically-scalable, virtualized, managed computing capacity, platforms, storage, and services are delivered on the need to external customers over the Internet. There are some key points as following. First, Cloud Computing is a specialized distributed computing paradigm; it differs from traditional ones in that 1) massively scalable, 2) can be encapsulated as an abstract entity that offers different levels of services to customers outside the Cloud, 3) driven by economies of scale, 4) the services can be dynamically configured (through virtualization or other approaches) and delivered on need.

Governments, enterprise, research institutes, and industries are tend to adopt Cloud Computing to solve their increasing computing and storage problems. There are three main factors contributing to the soar and interests in Cloud Computing: 1) Rapid decrease in hardware cost and increase in computing capacity and storage, and the advent of multi-core architecture and modern supercomputers consisting of thousands of cores. 2) The exponentially growing data size in scientific simulation and Internet archiving. 3) The wide-spread adoption of Services Computing and the advent of Web 2.0 applications [Foster et al., 2008].

Clouds are developed to deal with the internet-scale computing problems and usually referred to as a large pool of computing and storage resources, which can be accessed through standard protocols using an abstract interface. Clouds can be built on the top of many protocols such as Web Services(WSDL), and some advanced Web 2.0 technologies such as RSS, REST, AJAX, etc. Actually, it is possible for Clouds to be implemented over

existing Grid technologies more than a decade of community efforts in standardization, security, resource management, and virtualization support.

There are many versions of definition for Cloud architecture, a four-layer architecture for Cloud Computing in comparison to the Grid architecture are composed of 1) fabric, 2) unified resource, 3) platform, 4) application Layers [Foster et al., 2008].

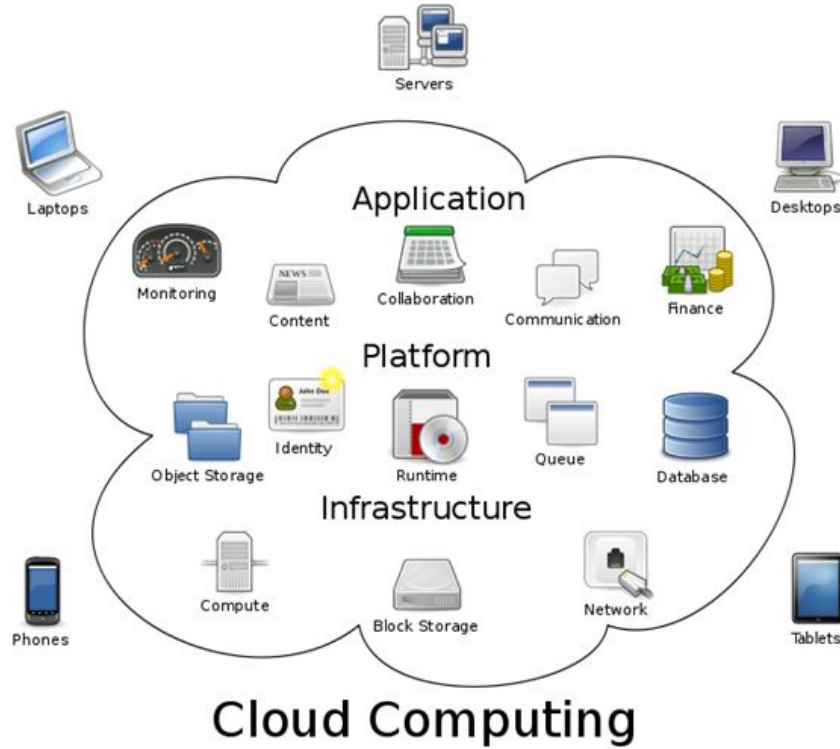


Figure 2.1: Cloud Computing

2.2 Big Data

Recent technological developments have led to a deluge of data from different domains (Internet and e-commerce companies, health care, scientific sensors, user log file data, as well as supply chain systems) over the past decade. The term big data become a very hot word. Apart from its sheer volume, big data also has some distinct characteristics compared with the traditional data. For example, big data is usually unstructured or semi-structured and requires more real-time analysis. This new data value chain needs a new system architectures for data acquisition, transmission, storage, and analytics [Hu et al., 2014].

2.2.1 Characteristics of Big Data

Due to the increasing data quantity and the change of data generation, for instance, pervasive sensing and computing across Internet, business, government and social network system are generating huge amount of heterogeneous data with unprecedented complexity, the massive datasets may have distinctive data characteristics in terms of temporal dimensional , scale, or variety of data types. For example, data generated from mobile phone and related to location, motion, proximity, communication, multimedia, application usage and audio environment were recorded. There are 4 attributes of big data compared with traditional data, which are listed below:

- **Volume:** Data scale in the range of PetaBytes, Exabytes and even more. It is estimated that the amount of available data is more than doubling every two years. The number of files, or containers that encapsulate the information, is growing even faster than the information itself as more and more systems store data and software systems are not in the stage that can process and obtain information from it. In the next five years, these files will grow by a factor of 8. It is important to highlight that the the amount of information individuals create themselves in form of writing documents, taking pictures, downloading music, etc, is far less than the amount of information being created about them.
- **Velocity:** relates both on how quickly data is being produced and how quickly the data must be processed to meet the demand for extracting useful information. There are many sources of high-velocity data such as log files of websites, devices, and number of other technologies that log events. Log mining enabled valuable information to be extracted from these log files which has resulted in an increase in logging. Also, IT devices such as routers, switches, firewalls, printers produce data with high velocity. One of the largest sources of high-velocity data are user devices such as smart phones where everything you do is/can be logged. Another form of massive amounts of real-time data is Social media (Twitter, Facebook, or any other social data streams), however, the value of such data degrades over time. There are many more other sources of high-velocity data and everyday we witness new ones. Also, useful information must be extracted in a timely manner otherwise it loses meaning, for example, promotions, which should be based on your current location, your purchase history, to identify what you like.

- **Variety:** data is in many format types - structured, unstructured, semi-structured, text (even in different languages), media, etc. A single application can be generating/collecting many types of data. This variety of unstructured data creates problems not only for storage but also for mining and analyzing data. To extract the knowledge all these types of data need to be linked together.
- **Veracity:** relates to managing the reliability and predictability of inherently imprecise data with noise and abnormality. It also refers to the fact that the data that is being stored, and mined must be meaningful to the problem being analyzed. Veracity in data analysis is the biggest challenge when compared with volume and velocity as it is harder to solve problems to keep your data clean [Stantic and Pokorny, 2014].

2.2.2 Introduction of Big Data

A big-data system is complicated, providing functions to process different types of data in every phases, ranging from its generation to the destruction. At the same time, the system always involves multiple phases for distinct applications. A systems-engineering approach is adopted to decompose a typical big data system into four integrated phases (data generation, data acquisition, data storage, as well as data analytics). In general, one shall visualize data to find some rough patterns first, and then employ specific data mining methods. The details for each phase are explained as follows.

Data generation concerns how data are generated. In this case, the term big data is designated to mean large, diverse, and complex datasets that are generated from various longitudinal and distributed data sources, including sensors, video, click streams, and other available digital sources. Normally, these datasets are associated with different levels of domain-specific values. In this paper, we focus on datasets from three prominent domains, business, Internet, and scientific research, for which values are relatively easy to understand. However, there are overwhelming technical challenges in collecting, processing, and analyzing these datasets that demand new solutions to embrace the latest advances in the information and communications technology (ICT) domain.

Data acquisition refers to the process of obtaining information and is subdivided into data collection, data transmission, and data pre-processing. First, because data may come from a diverse set of sources, websites that host formatted text, images and/or videos - data collection refers to dedicated data collection technology that acquires raw

data from a specific data production environment. Second, after collecting raw data, we need a high-speed transmission mechanism to transmit the data into the proper storage sustaining system for various types of analytical applications. Finally, collected datasets might contain many meaningless data, which unnecessarily increases the amount of storage space and affects the consequent data analysis. For instance, redundancy is common in most datasets collected from sensors deployed to monitor the environment, and we can use data compression technology to address this issue. Thus, we must perform data pre-processing operations for efficient storage and mining.

Data storage concerns persistently storing and managing large-scale datasets. A data storage system can be divided into two parts: hardware infrastructure and data management. Hardware infrastructure consists of a pool of shared ICT resources organized in an elastic way for various tasks in response to their instantaneous demand. The hardware infrastructure should be able to scale up and out and be able to be dynamically reconfigured to address different types of application environments. Data management software is deployed on top of the hardware infrastructure to maintain large-scale datasets. Additionally, to analyze or interact with the stored data, storage systems must provide several interface functions, fast querying and other programming models.

Data analysis leverages analytical methods or tools to inspect, transform, and model data to extract value. Many application fields leverage opportunities presented by abundant data and domain-specific analytical methods to derive the intended impact. Although various fields pose different application requirements and data characteristics, a few of these fields may leverage similar underlying technologies. Emerging analytics research can be classified into six critical technical areas: structured data analytics, text analytics, multimedia analytics, web analytics, network analytics, and mobile analytics. This classification is intended to highlight the key data characteristics of each area [Hu et al., 2014].

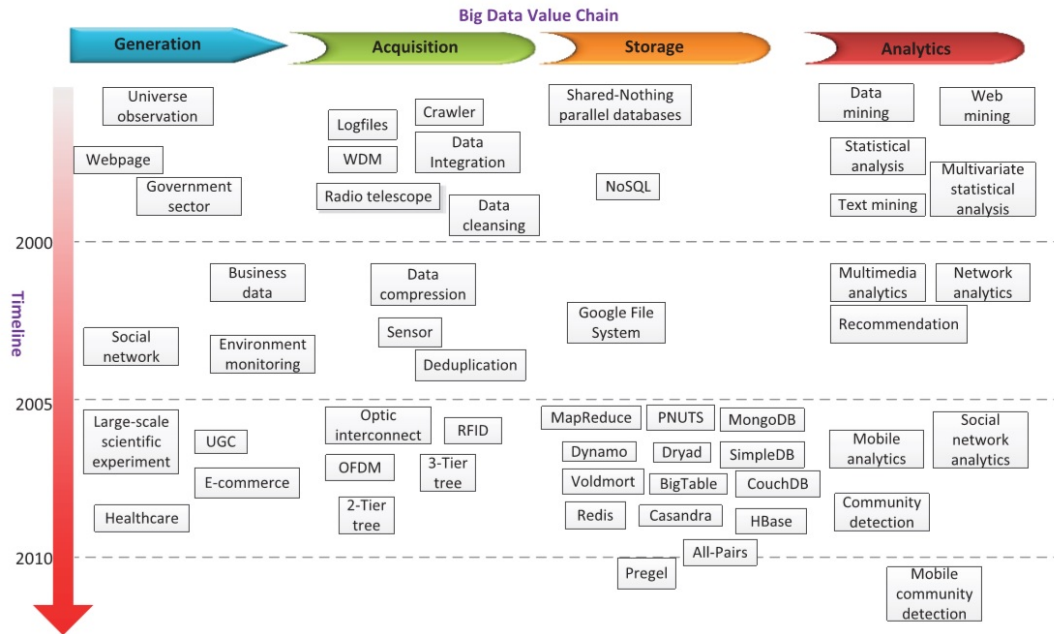


Figure 2.2: Big Data Value Chain

In figure 2.2, it has been shown Big data technology map. It pivots on two axes, data value chain and timeline. The data value chain divides the data lifecycle into four stages, including data generation, data acquisition, data storage, and data analytics. In each stage, main exemplary technologies over the past 10 years has been shown in the picture.

The Big Data Architecture Framework (BDAF) consists of the following 5 components that process different layers of the Big Data system which are considered to some extent orthogonal and complementary:

- *Data Models, Types, Structures*: Data formats, file systems, non/relational, etc.
- *Big Data Management*: Big Data transformation/staging, Big Data Lifecycle Management, Provenance, Curation, Archiving
- *Big Data Analytics and Tools*: Big Data Applications and Analytics Methods, Target use, visualisation, presentation
- *Big Data Infrastructure (BDI)*: General Compute and Storage infrastructure (cloud based), High Performance Computing (HPC), Network, Sensor network, target/actionable devices, Big Data services delivery and Operational support
- *Big Data Security and Privacy*: Data security and privacy in-rest, in-move, trusted processing environments.

2.3 NoSQL Database

NoSQL (not only SQL) is not a RDBMS (relational database management system). It is a new technology, so it is facing many challenges. Nowadays, the internet world has billions of users. With the increasing ACQUISITION of data from Sensor, Log File and Web Crawler, it becomes increasingly complex and unstructured, which means it is hard to manage massive data by traditional relational database management system. NoSQL technology becomes an answer to these problems, which breaks the limit of the relational database and its ACID (Atomicity, Consistency, Isolation, Durability) characteristics. NoSQL is designed for Unstructured Big Data and Cloud Computing, and it is exactly the type of database that could handle unstructured, semi-structured and unpredictable data.

2.3.1 The Basic Principles of NoSQL

The CAP theorem, BASE theorem as well as the Eventual Consistency theorem are the foundation stones. In 2000, Professor Eric Brewer put forward the famous CAP theorem. CAP are Consistency, Availability, tolerance of network Partition. The core idea of CAP theorem is the distributed system cannot meet the three needs at the same time, while can meet two of them. Figure 2.3 showing according to CAP theorem and different concerns of NoSQL databases, there are three orientations of designing system (CA, AP, CP). Availability is very important to the Web2.0. So when designing system, there are only CA orientation and AP orientation can be chosen. For the Web2.0 availability and tolerance of network Partition are more important than consistency. It is enough when the system meets the eventual consistency.

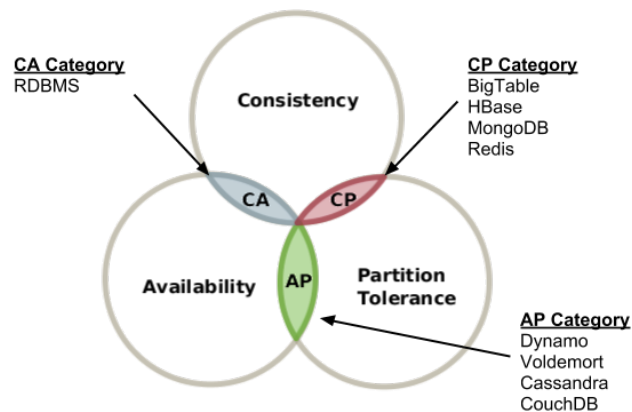


Figure 2.3: CAP Theorem

Design for consistency and availability (CA): Part of the database is not concerned about the partition of tolerance, and mainly use replication approach to make sure data consistency and availability. Systems concern the CA are: the traditional relational database, such as Vertica (Column-oriented), Greenplum (Relational), Aster Data (Relational) and etc.

Design for consistency and partition tolerance(CP): CP database system stores data in the distributed nodes, and ensure the consistency of these data, while support is not good enough for the availability. The main CP system are BigTable (Column-oriented), H-Base (Column-oriented), MongoDB (Document), Redis (Key-value), Hypertable (Column-oriented), Terrastore (Document), Scalaris (Key-value), Berkeley DB (Key-value), MemcacheDB (Key-value).

Design for availability and partition tolerance (AP): AP database system ensure availability and partition tolerance by achieving consistency, AP database mainly are CouchDB (Document-oriented), Voldemort (Key-value), SimpleDB (Document-oriented), Tokyo Cabinet (Key-value), KAI (Key-value), Riak (Document-oriented) [Han et al., 2011].

BASE theorem is a product of CAP theorem which integrates with practice, but it is completely different from ACID model. BASE is the abbreviation of Basically Available, Soft-state and Eventual consistency. Basically available is easily understood, partition failed could be supported. Soft state, it means in a period time the state of the system could be nonsynchronous. And at last, the data should be consistency.

Eventual consistency is one of the consistency models used in the domain of parallel programming. It means that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system and all the replicas will be consistent.

2.3.2 The Features of NoSQL

Main advantages of NoSQL are the following aspects:

1. Reading and writing data quickly;
2. Supporting mass storage;
3. Easy to expand;

4. Low cost.

Meanwhile, NoSQL have some shortages, such as does not support SQL, which is a industry standard, and lack transactions, reports and other additional functions. Characters of NoSQL database described above are common ones, in fact, each product comply with a different data models and CAP theorem. Therefore, I will introduce NoSQL database data model, and classify NoSQL according to CAP theorem [Han et al., 2011].

According to the storage models and the features, these databases could be divided into [Wang and Tang, 2012]:

- **Key-value** data model means that a value corresponds to a Key, although the structure is simpler, the query speed is higher than relational database, support mass storage and high concurrency, etc., query and modify operations for data through the primary key were supported well. **Redis** is a very new project, the following is its characteristics: 1) Redis is the a Key-value memory database: when Redis run, data were entire load into memory, so all the operations were run in memory, then periodically save the data asynchronously to the hard disk. The characteristics of pure memory operation makes it very good performance, it can handle more than 100,000 read or write operation per second; 2) Redis support List and Set and various related operations; 3) The maximum of value limit to 1GB; 4) the main drawback is that capacity of the database is limited by physical memory, so Redis cannot be used as big data storage, and scalability is poor. Therefore, Redis is suitable for providing high-performance computing to small amount of data.
- **Column-oriented** database using Table as the data model, but does not support table association. Column-oriented database has the following characteristics: 1) data is stored by column, that is data stored separately for each column; 2) each column of data is the index of database; 3) only access the columns involving the queries result to reduce the I/O of system; 4) concurrent process queries, that is, each column treat by one process; 5) there have the same type of data, similar characteristics and good compression ratio. Overall, the advantage of this data model is more suitable application on aggregation and data warehouse. **Cassandra** is an open source database of facebook. Its characteristics are: 1) the schema is very flexible and does not require to design database schema at first, and add or delete field is very convenient; 2) support range queries, that is it can range queries for

Key; 3) high scalability: a single point of failure does not affect the whole cluster, and it support linear expansion. Cassandra system is a distributed database system which was composed of lots of database nodes, a write operation will be replicated to other nodes, and read request will be routed to a certain node. For a Cassandra cluster, only to add node can achieve the goal of scalability. In addition, Cassandra also supports rich data structure and powerful query language.

- **Document database** is very similar to key-value structure, but the Value of document database is semantic, and is stored in JSON or XML format. In addition, the document databases can generally a Secondary Index to value to facilitate the upper application, but Key-value database cannot support this. **MongoDB** is a database between relational databases and non-relational database, its features are: 1) it is non-relational database, which features the richest and most like the relational database; 2) support complex data types: MongoDB support bson data structures to store complex data types; 3) powerful query language: it allows most of fuctions like query in single-table of relational databases, and also support index. 4) High-speed access to mass data: when the data exceeds 50GB, MongoDB access speed is 10 times than MySQL. Because of these characteristics of MongoDB, many projects with increasing data are considering using MongoDB instead of relational database. **Apache CouchDB** is a flexible, fault-tolerant database, which supports data formats such as JSON and AtomPub, it provides REST-style API. To ensure data consistency, CouchDB comply with ACID properties. In addition, CouchDB provides a P2P-based distributed database solution that supports bidirectional replication. However, it also has some limitations, such as only providing an interface based on HTTP REST, concurrent read and write performance is not ideal and so on [Han et al., 2011].

Table 2 Partitioning, replication, consistency, and concurrency control capabilities

NoSQL data stores	Partitioning	Replication	Consistency	Concurrency control	
Key-value stores	Redis	Not available (planned for Redis Cluster release). It can be implemented by a client or a proxy.	Master-slave, asynchronous replication.	Eventual consistency. Strong consistency if slave replicas are solely for failover.	Application can implement optimistic (using the WATCH command) or pessimistic concurrency control.
	Memcached	Clients' responsibility. Most clients support consistent hashing.	No replication Replicated can be added to memcached for replication.	Strong consistency (single instance).	Application can implement optimistic (using CAS with version stamps) or pessimistic concurrency control.
	BerkeleyDB	Key-range partitioning and custom partitioning functions. Not supported by the C# and Java APIs at this time.	Master-slave	Configurable	Readers-writer locks
	Voltdemort	Consistent hashing.	Masterless, asynchronous replication. Replicas are located on the first R nodes moving over the partitioning ring in a clockwise direction.	Configurable, based on quorum read and write requests.	MVCC with vector clock
	Riak	Consistent hashing.	Masterless, asynchronous replication. The built-in functions determine how replicas distribute the data evenly.	Configurable, based on quorum read and write requests.	MVCC with vector clock.
Column family stores	Cassandra	Consistent hashing and range partitioning (known as order preserving partitioning in Cassandra terminology) is not recommended due to the possibility of hot spots and load balancing issues.	Masterless, asynchronous replication. Two strategies for placing replicas: replicas are placed on the next R nodes along the ring; or, replica 2 is placed on the first node along the ring that belongs to another data centre, with the remaining replicas on the nodes along the ring in the same rack as the first.	Configurable, based on quorum read and write requests.	Client-provided timestamps are used to determine the most recent update to a column. The latest timestamp always wins and eventually persists.
	HBase	Range partitioning.	Master-slave or multi-master, asynchronous replication. Does not support read load balancing (a row is served by exactly one server). Replicas are used only for failover.	Strong consistency	MVCC
	DynamoDB	Consistent hashing.	Three-way replication across multiple zones in a region. Synchronous replication	Configurable	Application can implement optimistic (using incrementing version numbers) or pessimistic concurrency control.
	Amazon SimpleDB	Partitioning is achieved in the DB design stage by manually adding additional domains (tables). Cannot query across domains.	Replicas within a chosen region.	Configurable	Application can implement optimistic concurrency control by maintaining a version number (or a timestamp) attribute and by performing a conditional put/delete based on the attribute value.
Document stores	MongoDB	Range partitioning based on a shard key (one or more fields that exist in every document in the collection). In addition, hashed shard keys can be used to partition data.	Master-slave, asynchronous replication.	Configurable Two methods to achieve strong consistency: set connection to read only from primary; or, set write concern parameter to "Replica Acknowledged".	Readers-writer locks

Figure 2.4: NoSQL and NewSQL comparison I [Grolinger et al., 2013]

Table 2 Partitioning, replication, consistency, and concurrency control capabilities (Continued)

	CouchDB	Consistent hashing.	Multi-master, asynchronous replication. Designed for offline operation. Multiple replicas can maintain their own copies of the same data and synchronize them at a later time.	Eventual consistency.	MVCC. In case of conflicts, the winning revision is chosen, but the losing revision is saved as a previous version.
	Couchbase server	A hashing function determines to which bucket a document belongs. Next, a table is consulted to look up the server that hosts that bucket.	Multi-master.	Within a cluster: strong consistency. Across clusters: eventual consistency.	Application can implement optimistic (using CAS) or pessimistic concurrency control.
	Neo4J	No partitioning (cache sharding only).	Master-slave, but can handle write requests on all server nodes. Write requests to slaves must synchronously propagate to master.	Eventual consistency.	Write locks are acquired on nodes and relationships until committed.
Graph databases	Hyper GraphDB	Graph parts can reside in different P2P nodes. Builds on autonomous agent technologies.	Multi-master, asynchronous replication. Agent style communication based on Extensible Messaging and Presence Protocol (XMPP).	Eventual consistency.	MVCC.
	Allegro graph	No partitioning (federation concept which aims to integrate graph databases is abstract at the moment).	Master-slave.	Eventual consistency.	Unclear how locking is implemented "100% Read Concurrency, Near Full Write Concurrency".
NewSQL	VoltDB	Consistent hashing. Users define whether stored procedures should run on a single server or on all servers.	Updates executed on all replicas at the same time.	Strong consistency.	Single threaded model (no concurrency control).
	Spanner	Data partitioned into tablets. Complex policies determine in which tablet the data should reside.	Global ordering in all replicas (Paxos state machine algorithm).	Strong consistency.	Pessimistic locking in read-write transactions. Read-only transactions are lock-free (versioned reads).
	Clustrix	Consistent hashing. Also partitions the table indices using the same approach.	Updates executed on all replicas at the same time.	Strong consistency.	MVCC.
	NuoDB	No partition. The underlying key-value store can partition the data, but it is not visible by the user.	Multi-master (distributed object replication). Asynchronous.	Eventual consistency.	MVCC.

Figure 2.5: NoSQL and NewSQL comparison II [Grolinger et al., 2013]

3

Literature Review

In this chapter, we present a literature review of the architecture of Hadoop, including HDFS and HBase (how to replicate and store data), MapReduce (how it computes batches data), YARN, Pig and Hive, Storm (how to process real time streaming data), Spark (a fast and general-purpose cluster computing system), and Shark (a port of Apache Hive onto Spark).

3.1 Hadoop

Apache Hadoop is an open-source software framework that supports massive data storage and processing. Instead of relying on expensive , proprietary hardware to store and process data, Hadoop enables distributed processing of large amounts of data on large clusters of commodity servers. Because of the great success of Googles distributed file system and the MapReduce computation model in handling massive data processing, its clone, Hadoop , has attracted substantial attention from both industry and scholars alike. In fact, Hadoop has long been the mainstay of the big data movement.

3.1.1 Characteristics of Hadoop

Hadoop has many advantages, and the following features make Hadoop particularly suitable for big data management and analysis:

- **Scalability:** Hadoop allows hardware infrastructure to be scaled up and down with no need to change data formats. The system will automatically redistribute data and computation jobs to accommodate hardware changes.
- **Cost Efficiency:** Hadoop brings massively parallel computation to commodity servers, leading to a sizeable decrease in cost per terabyte of storage, which makes massively parallel computation affordable for the ever-growing volume of big data.
- **Flexibility:** Hadoop is free of schema and able to absorb any type of data from any number of sources. Moreover, different types of data from multiple sources can be aggregated in Hadoop for further analysis. Thus, many challenges of big data can be addressed and solved.
- **Fault tolerance:** Missing data and computation failures are common in big data analytics. Hadoop can recover the data and computation failures caused by node breakdown or network congestion [Hu et al., 2014].

3.2 Hadoop Component

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are commonplace and thus should be automatically handled in software by the framework.

3.2.1 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high

throughput access to application data and is suitable for applications that have large data sets. The reasons of the development and popularity of HDFS are as following :

- **Hardware Failure.** Hardware Failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.
- **Streaming Data Access.** Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.
- **Large Data Sets.** Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.
- **Simple Coherency Model.** HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.
- **Moving Computation is Cheaper than Moving Data.** A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides

interfaces for applications to move themselves closer to where the data is located.

- **Portability Across Heterogeneous Hardware and Software Platforms.** HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications [Borthakur, 2008].

NameNode: The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file), and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of blocks to DataNodes. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

DataNodes: Each block replica on a DataNode is represented by two files in the local native filesystem. The first file contains the data itself and the second file records the block's metadata including checksums for the data and the generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional filesystems. Thus, if a block is half full it needs only half of the space of the full block on the local drive. During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode, the DataNode automatically shuts down. The namespace ID is assigned to the filesystem instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus protecting the integrity of the filesystem. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID. After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that. A DataNode

identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block ID, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster. During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats: Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's block allocation and load balancing decisions. The NameNode does not directly send requests to DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, and shut down the node. These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

Image and Journal: The inodes and the list of blocks that define the metadata of the name system are called the image. NameNode keeps the entire namespace image in RAM. The persistent record of the image stored in the NameNode's local native filesystem is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native filesystem. The location of block replicas are not part of the persistent checkpoint.

Each client-initiated transaction is recorded in the journal, and the journal file is flushed and synced before the acknowledgment is sent to the client. The checkpoint file is never changed by the NameNode; a new file is written when a checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal. A new checkpoint and an empty

journal are written back to the storage directories before the NameNode starts serving clients.

For improved durability, redundant copies of the checkpoint and journal are typically stored on multiple independent local volumes and at remote NFS servers. The first choice prevents loss from a single volume failure, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize this process, the NameNode batches multiple transactions. When one of the NameNode's threads initiates a flush-and-sync operation, all the transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

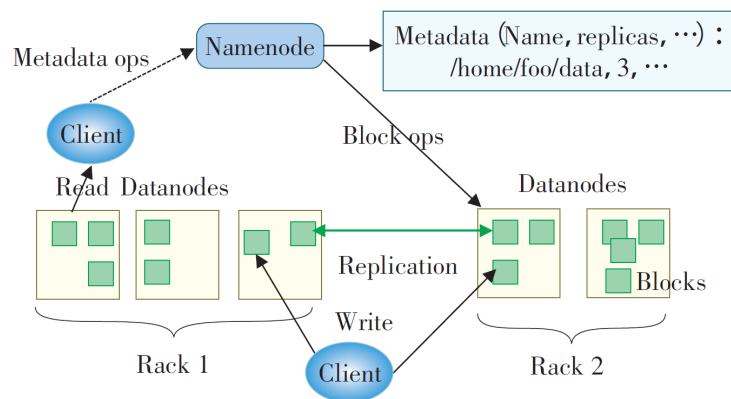


Figure 3.1: HDFS framework

User applications access the filesystem using the HDFS client, a library that exports the HDFS filesystem interface. Like most conventional filesystems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application does not need to know that filesystem metadata and storage are on different servers, or that blocks

have multiple replicas. When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. The list is sorted by the network topology distance from the client. The client contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Choice of DataNodes for each block is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 3.2: HDFS architecture. Unlike conventional filesystems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves tolerance against faults and increases read bandwidth.

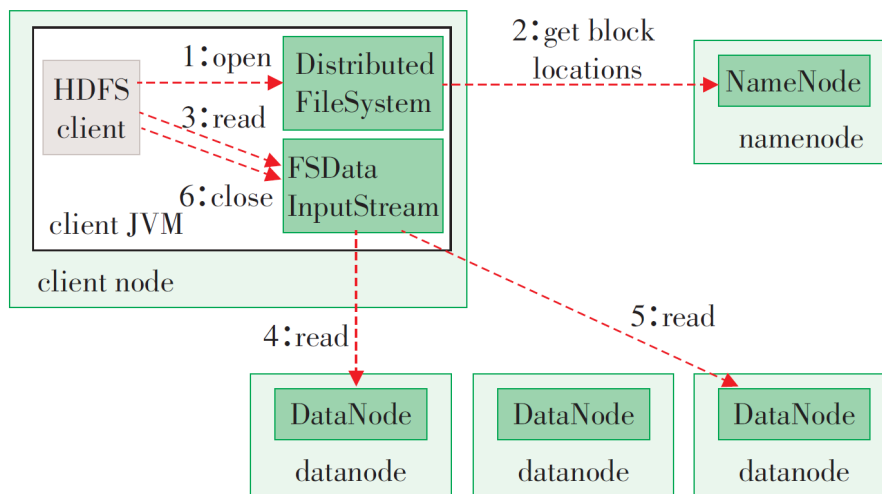


Figure 3.2: HDFS Reading Process

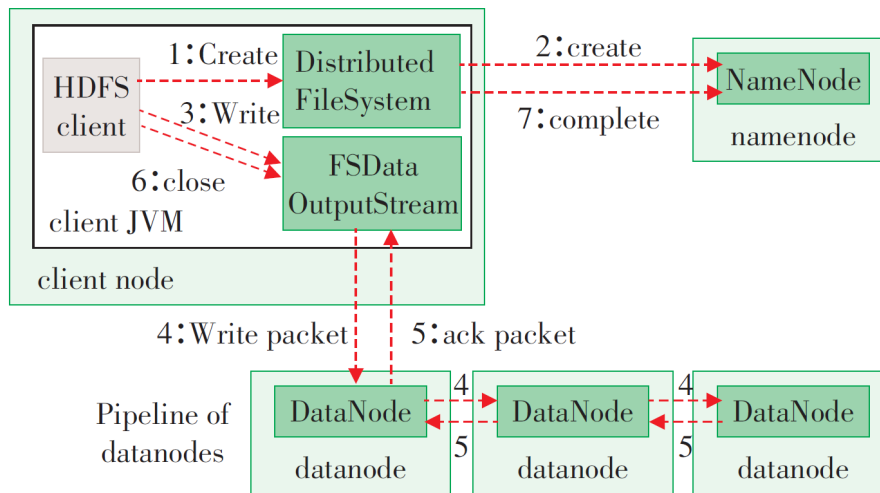


Figure 3.3: HDFS Writing Process

In Figure 3.2 and 3.3 show the process of data reading and writing in HDFS files.

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline.

The next packet can be pushed to the pipeline before receiving the acknowledgment for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.

After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the `hflush` operation. Then the current packet is immediately pushed to the pipeline, and the `hflush` operation will wait until all DataNodes in the pipeline acknowledge the successful transmission of the packet. All data written before the `hflush` operation are then certain to be visible to readers.

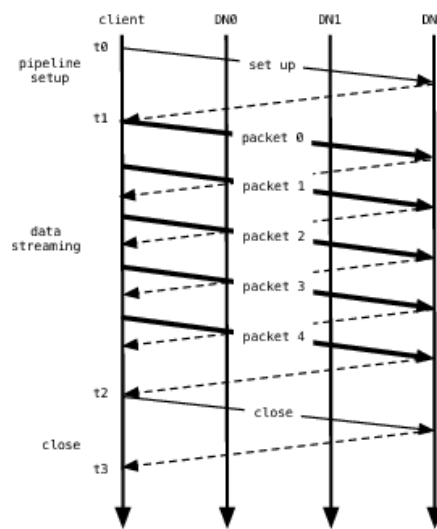


Figure 3.4: Data Pipeline While Writing a Block

If no error occurs, block construction goes through three stages as shown in figure 3.4 illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t_0 to t_1 is the pipeline setup stage. The interval t_1 to t_2 is the data streaming stage, where t_1 is the time when the first data packet gets sent and t_2 is the time that the acknowledgment to the last packet gets received. Here an `hflush` operation transmits packet 2. The `hflush` indication travels with the packet data and is not a separate operation. The final interval t_2 to t_3 is the pipeline close stage for this block.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because

of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content. The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.

Data Replication: HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time. The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

- **Replica Placement: The First Steps.** The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current implementation for the replica placement policy is a first effort in this direction. The short-term goals of implementing this policy are to validate it on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated policies. Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks. The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks. For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance. The current, default replica placement policy described here is a work in progress.
- **Replica Selection.** To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader.

If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

- **Safemode.** On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes [Borthakur, 2008].

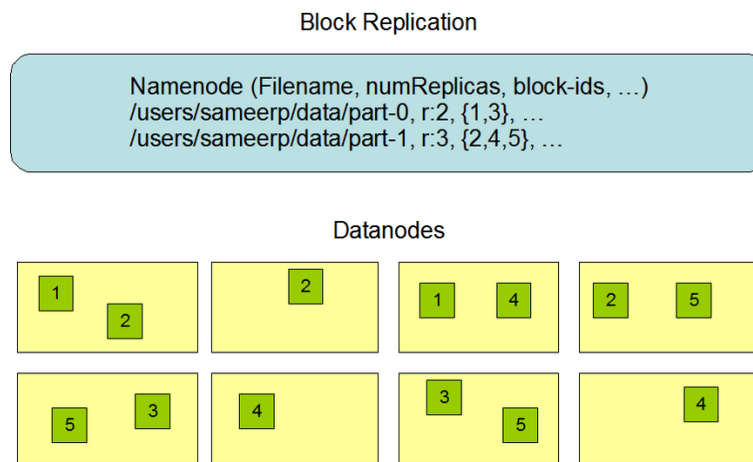


Figure 3.5: HDFS Block Replication

Balancer: HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new more likely to be referenced data at a small subset of the DataNodes with a lot of free storage. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster. The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node differs

from the utilization of the whole cluster⁴ by no more than the threshold value. The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.

The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A. A configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

3.2.2 Mapreduce

MapReduce is a framework originally designed by Google to exploit large clusters to perform parallel computations. It is based on an implicit parallel programming model that provides a convenient way to express certain kinds of distributed computations, particularly those that process large data sets. The framework is composed of an execution runtime and a distributed file system. In this subsection we present a prototype implemented on top of Hadoop, an open-source runtime provided by the Apache Software Foundation.

The Hadoop runtime consists of two types of processes, called JobTracker and TaskTracker. The JobTracker partitions the input data into splits using a splitting method defined by the programmer, populates a local task-queue based on the number of splits, and distributes work to the TaskTrackers that in turn process the splits. If a TaskTracker becomes idle, the JobTracker picks a new task from its queue for that TaskTracker to execute. Thus, the granularity of the splits has considerable influence on the balancing capability of the scheduler. Another consideration is the location of the data blocks: among all the tasks of the selected job to be executed, it prioritizes those tasks with data local to the TaskTracker. Notice that this data locality consideration never affects the decision about the scheduled job.

Each TaskTracker controls the execution of tasks on a node. It receives a split descriptor from the JobTracker, and spawns a new worker process which runs a so-called map task

to process the split received from the JobTracker. The TaskTracker also runs the so-called reduce tasks as soon as they can be initiated. Notice here that a map task will eventually result in the execution of a `map()` function, and that a reduce task similarly results in the execution of a `reduce()` function. The programmer can also decide how many simultaneous `map()` and `reduce()` functions can be run concurrently on a node [Polo et al., 2013].

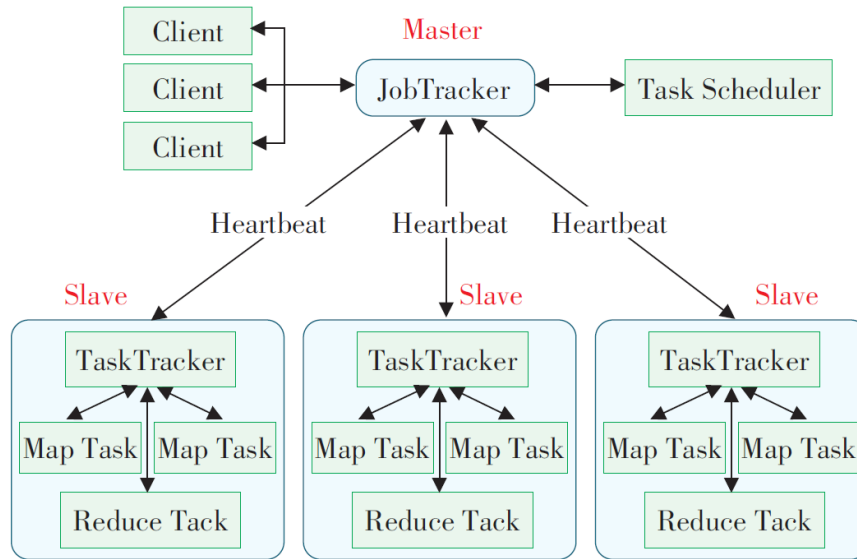


Figure 3.6: Mapreduce Framework

MapReduce programs have its base in functional programming paradigm. The main objective of the programmer is to define and create map function and reduce function. One of the highlight of functional programming paradigm is that it accepts other functions as function arguments. The primitive data types in mapreduce programs are key-value pairs. MapReduce allows programmers to concentrate on business needs and keeps him/her free from the distributed file systems complexities. As shown in Fig 2.2 the Map task produces a set of key value pairs from the data chunks which is the input to the reduce task. MapReduce uses a shared nothing model which means that each node in the architecture is self sufficient and independent. This no shared model helps to achieve scalability without much complexity. In a nutshell we can conclude the advantages of mapreduce as high fault tolerance, comfortable programming structure, automatic distribution of workload and high elasticity.

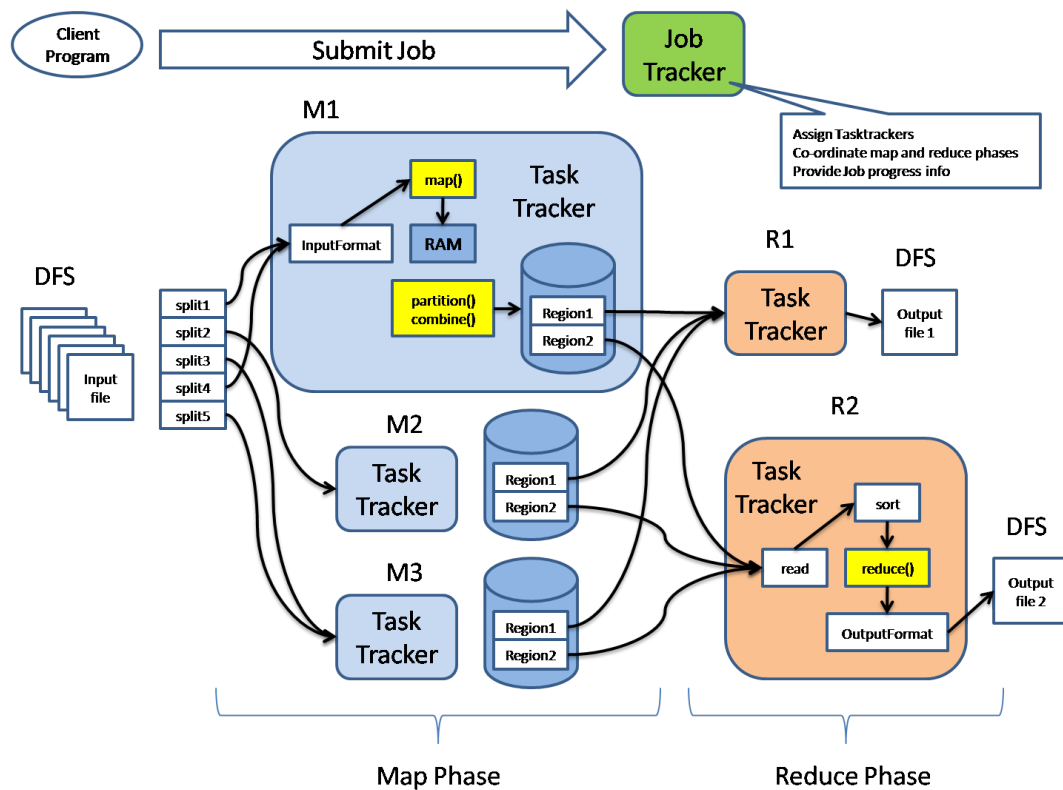


Figure 3.7: MapReduce Framework with HDFS

Hadoop is an open source project implementation of Google's MapReduce architecture to GFS. The MapReduce environment as shown In figure 3.7 describe of Mapreduce framework with the input and the output to DFS(GFS). Hadoop will be running on top of all there three components. The files will be very of large sizes in the range of Terabytes or Petabytes. It is built on top of low cost hardware.

Data access in HDFS is by bucketing or streaming method. HDFS basically follows Google file system. The advantage of HDFS is high scalability and very high availability. It is both hardware and software tolerant. It has moderately good fault tolerance as it does automatic re-replication of data on failed nodes. Hadoop addresses the two big challenges put forward by BigData firstly how to store large volumes of data and secondly how to work with copious data Hadoop Distributed File System comprise of a client server architecture consisting of a name node (also called as master node) and a set of data nodes. The name node has a job tracker which keeps monitoring the task assigned to the data node. The data node contains a task tracker which will be constantly supervising the task given by the name node. Hadoop is been used by big giants such as yahoo, Facebook and Amazon.

The replica process is handled by name node. HDFS also has a secondary node which can be configured in some other system other than the one in which name node resides. But in case of failure of a name node the secondary node cannot replace the name node. The duty of secondary name node is to perform frequent checks on the name node and to store images of the name node. In case of a failure of name node, the most recent image will be uploaded into the name node by the secondary node. HDFS is introduced with the motive that it is easier to move the computation rather than moving the data. Hadoop is another implementation of MapReduce. Even though Hadoop is widely accepted and has many applications it still has some pitfalls. The failure of name node will result in manual intervention and it is a single point failure. Another glitch is HDFS small files problem which will restrict scalability of the cluster. Hadoop can be compared to that of a database and it requires more optimization for increased processing performance [Deni Raj et al., 2014].

3.2.3 YARN

In 2002, Mike Cafarella and Doug Cutting started working on Apache Nutch, a new Web search engine. In 2004, they implemented an open source MapReduce version of Nutch; the same project, in 2006 became Hadoop. In 2008 Yahoo! adopted Hadoop on a 10,000-core cluster. Thanks to this choice, one year later, Yahoo! broke the world record for sorting 1 TB of data in 62 seconds and Hadoop was consecrated as mainstream in industry.

Hadoop architecture spans across several layers. The bottom level (Hardware) contains libraries and utilities, stores data, and supports jobs execution. The file-system layer (HDFS) is a distributed file-system. In order to preserve information in case of failures, which are common at large scale, HDFS uses by default a scheme of three way replication. A resource-management platform (MapReduce execution time) is responsible for managing computing resources in clusters and for scheduling end users' applications. In Hadoop v2.0 this layer is split in two parts: the first level is YARN (Yet Another Resource Negotiator), the second level includes several programming models that can co-exist in the cluster and MapReduce is only one of them. Finally, the Application layer is composed of several frameworks, among which Pig and Hive are the most used. This abstraction layer enables users to express applications with a high level declarative language.

Hadoop 1.X is characterized by a Master node (called Job Tracker) that is in charge of

managing both the resources (called Task Trackers) and the programming model. Scheduling and allocation decisions are made on tasks and node slots level for both the map and reduce phases, based on a FIFO queue policy. Its simplicity implies a series of limitations in terms of performance, availability, and allocation efficiency. Additionally, the Job Tracker is the bottleneck and also the single point of failure of the entire system. These limitations led to scalability problems.

Among the different solutions proposed to tackle these problems (Mesos, Omega, Corona), YARN, developed by Yahoo!, is the one that is most widely adopted nowadays and it is described in the following section [Rumi et al., 2014].

The fundamental idea behind Hadoop 2.X is to split the functionalities of the Job Tracker (resource management and job scheduling/monitoring) into two separate entities: a global Resource Manager, and a per-application Application Master.

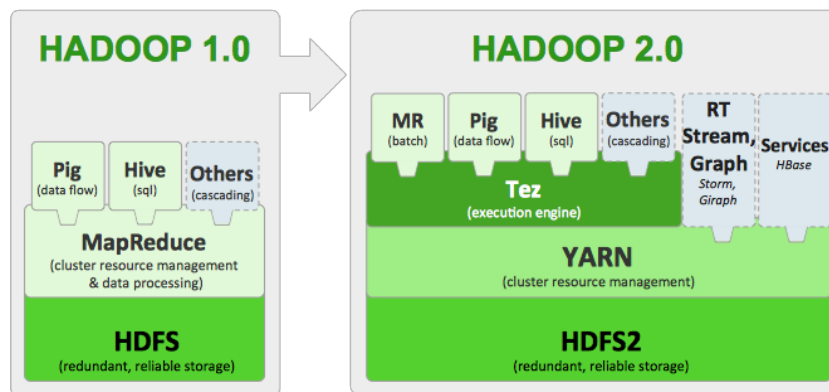


Figure 3.8: Hadoop YARN

The Resource Manager is responsible for allocating resources to the various running applications but it does not perform any monitoring or tracking of their status. It performs its scheduling function based on the resource requirements of the applications; this is done based on the abstract notion of a Resource Container, which incorporates elements such as memory, CPU, disk, and network. The Application Masters, instead, are those processes that coordinate the application execution in the cluster. Periodically, Application Masters contact the Resource Manager in order to notify their status and to ask/release their Resource Container requirements.

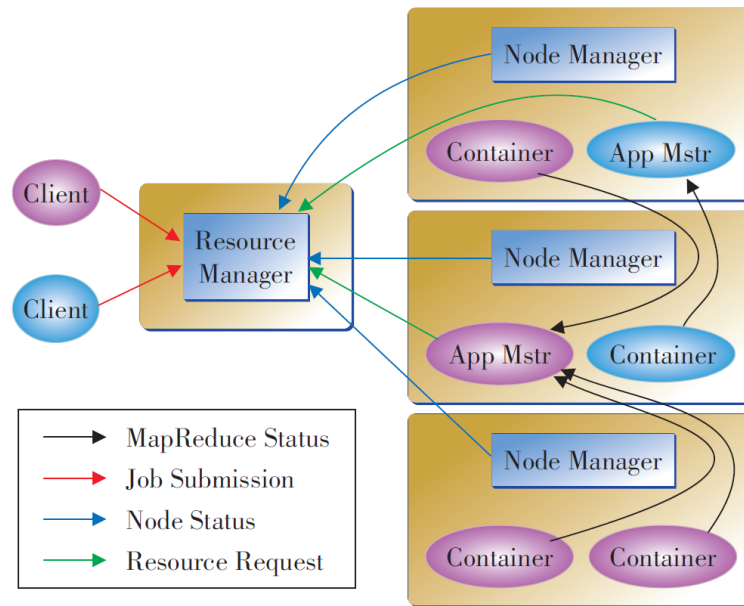


Figure 3.9: YARN Framework

YARN has a different architecture with respect to the first Hadoop implementation, as shown in Figure 3.9. The innovation is that MapReduce became an application running on top of YARN. This separation leads to reduce the number of activities in charge of the Resource Manager, to an increased system scalability, and favours also the introduction of different programming models. Fault tolerance also benefits from this new architecture. YARN also eliminates the limitations of Hadoop 1.X, introducing other default schedulers, and gives the possibility to share MapReduce cluster among users. The Scheduler, which had a secondary role in Hadoop 1.X, becomes fundamental in YARN. The additional schedulers implement alternative fair share capacity algorithms where separate queues are maintained for separate pools of users, and each one has some service guarantees over time. These additional schedulers are the FAIR, and Capacity Scheduler.

The Fair Scheduler introduces two innovations: isolation and statistical multiplexing. Isolation is achieved allowing users to assign jobs to pools, in this way they have the illusion of running a private cluster; while statistical multiplexing allows to reallocate idle slots in other pools to increase system utilization. Moreover, the FAIR scheduler supports work-conserving pre-emption (i.e., pre-empted task can be resumed and their partial results are not lost).

The Capacity Scheduler, by Yahoo!, guarantees user capacity and fairness. The available resources are shared among users and any underutilized node is shared between the

existing jobs. The Capacity Scheduler is based on queues, which are set-up by administrators to reflect the economics of the shared cluster and the submission time of the jobs; it also supports work-conserving pre-emption. Scheduling has an important impact in applications performance and is also application specific. For these reasons, YARN allows to develop pluggable schedulers. These are the common set of problems addressed by a scheduling policy: Data locality occurs when data needed by a job are not stored in the local node; in this case data must be transmitted over the network. Sticky slots problem relates the re-allocation of different tasks of the same job on the same node when the node does not have local data; in this case for each task the data will be continuously sent over the network. Poor system utilization happens when system resources are not well distributed among users. Skewness and Map-Reduce interdependence refers to the fact that unbalanced jobs can bring to an over/sub estimation of their execution time (e.g., an unexpected slow map can delay the whole job execution, since reduce can not start before every maps have finished). This can cause the overload of a node or waste of resources. Starvation occurs when a task does not complete due to continuous interruptions by high priority jobs; this problem is present when a non-fair scheduling policy is adopted. Finally, Fairness occurs when there are no mechanisms to guarantee users' priority [Rumi et al., 2014].

3.2.4 Pig and Hive

Various criticisms of Hadoop have revolved around its long development cycle: setting the environment, writing the mappers and reducers, compiling and packaging the code, job submissions, retrieving the results are time consuming activities. Pig and Hive free users from all these technicalities and reduce the programming effort by a factor of at least 7.5 (in term of lines of code written) allowing end users to focus on data analysis.

Pig is a platform for analysing large datasets that consists of a high-level language (Pig Latin) for expressing data analysis programs, coupled with an infrastructure for evaluating them. It abstracts the procedural style of MapReduce in the direction of the declarative style of SQL. A Pig program generally goes through three steps: load, transform, and store. At first the data on which the program has to work are loaded (in Hadoop the objects are stored in HDFS); then a set of transformations are applied to the loaded data and the mappers and reducers are handled transparently to the user; finally, if needed, the results are stored in a local file or in HDFS.

Pig main features are: i) ease of programming, ii) optimization opportunities, iii) customization, and iv) extensibility. Those features make it largely widespread: for instance at Ya-hoo! 40 percent of all Hadoop jobs are Pig programs. However, its limitations are: i) Pig Latin is a totally new language; ii) Pig shows high-latency when performing operations on a small portion of input, since it is designed for processing batch data analyses. The effort in learning Pig Latin could discourage data analysts willing to leverage the Hadoop platform on a higher level. Hence, Hive has been proposed as an alternative.

Apache Hive facilitates large datasets query and management in distributed storage. It offers SQL-like queries called Hive-QL (Hive Query Language) for writing statements very similar to standard SQL. Hive automatically manages the compilation, optimization, and execution of a Hive-QL statement. A compiler translates statements into a DAGs (Direct Acyclic Graphs) of MapReduce jobs transparently to the user. Hive is best suited for batch jobs over large sets of append-only data, providing i) scalability; ii) extensibility; iii) fault tolerance; and iv) loose-coupling with its input formats. Its limitations are: i) it has a limited set of commands; ii) queries show high latency (order of minutes) and is intended for long sequential scans; iii) it is mainly read-only, hence it is not appropriate for OLTP (On-line Transaction Processing), which requires continuous writes on operational data. As an example, Hive warehouse instance in Facebook contains over 700 Terabytes of data and supports over 5,000 daily queries. Hive and Pig tasks are generally expressed as high-level declarative abstractions to be then compiled into DAGs of MapReduce jobs. Each DAG, splits in sequential and parallel jobs, starts with the input datasets division into small chunks and their distribution over the nodes. The additional Pig/Hive layer introduces the need of a social scheduling: the jobs to computational nodes mapping has to take into account applications time constraints over the shared cluster, performing optimization both at the application and at the task level. These issues are discussed in the following sections [Rumi et al., 2014].

3.3 Stream Processing

3.3.1 Spark

Apache Spark is a fast and general-purpose cluster computing system which Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. It provides

high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only added to, such as counters and sums. This guide shows each of these features in each of Spark's supported languages. It is easiest to follow along with if you launch Spark's interactive shell either `bin/spark-shell` for the Scala shell or `bin/pyspark` for the Python one¹.

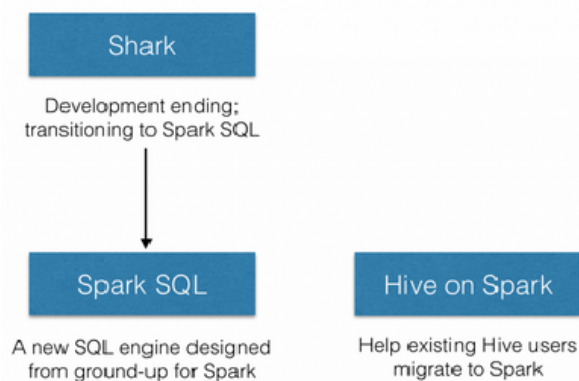


Figure 3.10: Spark

Shark, a port of Apache Hive onto Spark which is compatible with existing Hive

¹[HTTPS://spark.apache.org/](https://spark.apache.org/)

warehouses and queries, can answer HiveSQL queries up to 100 times faster than Hive without modification to the data and queries, and is also open source as part of BDAS.

3.3.2 Storm

Storm² is a open source distributed realtime computation system, which makes it easy to reliably process unbounded streams of data, and doing for realtime processing what Hadoop did for batch processing. Storm is simple, and can be used with plenty of programming language. Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees the data will be processed quickly, and is easy to set up and operate.

According to Nathan Marz, the lead engineer of Storm: "Storm makes it easy to write and scale complex realtime computations on a cluster of computers, doing for realtime processing what Hadoop did for batch processing. Storm guarantees that every message will be processed, it is also fast so you can process millions of messages per second with a small cluster. Best of all, you can write Storm topologies using any programming language." The important properties of Storm are:

- Simple programming model. Similar to how MapReduce lowers the complexity of doing parallel batch processing, Storm lowers the complexity for doing real-time processing.
- Runs any programming language. You can use any programming language on top of Storm. Clojure, Java, Ruby, Python are supported by default. Support for other languages can be added by implementing a simple Storm communication protocol.
- Fault-tolerant. Storm manages worker processes and node failures.
- Horizontally scalable. Computations are done in parallel using multiple threads, processes and servers.
- Guaranteed message processing. Storm guarantees that each message will be fully processed at least once. It takes care of replaying messages from the source when a task fails.

²<http://storm.apache.org/>

- Fast. The system is designed so that messages are processed quickly and uses MQ as the underlying message queue.
- Local mode. Storm has a "local mode" where it simulates a Storm cluster completely in-process. This lets you develop and unit test topologies quickly³.

³<http://www.infoq.com/news/2011/09/twitter-storm-real-time-hadoop/>

4

Contributing Chapter

“If we knew what we were doing, it would not be called research.”

Albert Einstein

In this Chapter, a contribution which concern about the comparison of Hadoop veision1 and version2 on YARN, and the comparison of processing engine (spark and storm) has been presented.

4.1 Comparison of Hadoop Version1 and Version2 on YARN

4.1.1 Hadoop Version1

Hadoop version1.0 mainly includes three components: Hadoop Common, HDFS and MapReduce. Hadoop Common provides some General functions (such as RPC, serialization, etc.), which is used by other components in Hadoop. For users, Hadoop Common

does not provide actually useful function; while HDFS is a (hadoop) distributed file system, which offers distributed storage of massive datasets; and MapReduce is a computing framework which adopt to a simple programming model to achieve distributed processing massive data on large-scale computer clusters. Hadoop version1.0 has been found slow in the past two years. In literature review, the architecture and characters of hadoop ecosystem and each component (HDFS, Mapreduce, pig and hive) have been discussed. So the shortages are easy to be seen, from both HDFS and Mapreduce. Two of the most serious problems of HDFS are the single-node failure of NameNode and the limitation of capacity and performance in cluster. Data replicated and stored on different datanodes in HDFS, so there is no single-node failure when one of the datanode does not work, and data can be recovered. But there is only one Namenode, once Namenode fails when it is required to participate in all file operations, the entire HDFS cannot work. Similarly, due to the participation of all file operations, Namenode can easily become the bottleneck of the performance of the entire cluster, which has a large amount of nodes. In addition, the cluster performance is limited by the number of Namenodes and its capacity is also limited by the memory size of Namenode. A method which can be roughly estimated is that: suppose a file related information occupies 1 kB, storing 100 million files need 100 GB of memory, then store 1 billion files you need 1 TB of memory. For such a large memory requirements, a single server cannot meet.

a) Similar to HDFS architecture there is an existing problem in Mapreduce as JobTracker has a possibility to meet a single-node failure due to the Master-Slave relationship.

b) TaskTracker is divided into Slots. Due to the MapReduce FIFO scheduler, jobs are executed according to the order they arrived in the queue (as long as they have the same priority). The next job will wait and not start until the current job is finished, even if the current job is not completely consuming the capacity so that the waiting jobs cannot be run with the remaining capacity. MapReduce FIFO scheduler leads to irrational distribution of resources and low utilization of cluster. For example, some tasks which consume large memory are allocated to some nodes respectively, while the memory capacity of the other idle nodes are not fully utilized. That is to say the bottleneck comes from the imbalance of map and reduce tasks. Map and reduce tasks are separately and in Hadoop version one all the mapper slots are occupied while only several reduce slots are utilized.

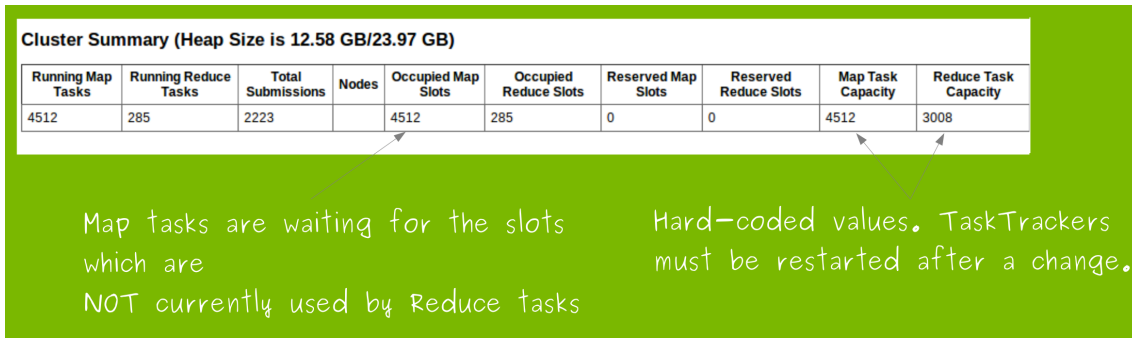


Figure 4.1: Poor-utilization Of Reduce

In figure 4.1 it has been shown when all the mapper slots are occupied (Running Map Tasks = Occupied Map Tasks = Map Tasks Capacity = 4512), only several reduce slots are utilized (Running Reduce Tasks = Occupied Reduce Tasks = 285 << Reduce Tasks Capacity = 3008). Reduce Tasks Capacity to some degree is matching the Map Tasks Capacity, but the usage percentage is only 1/10.

c) JobTracker is overused and undertakes too many functions, such as the resource management, job allocation and tracking. The overload of JobTracker in large-scale cluster cannot only trigger bottleneck of performance, but also increase the probability of failure.

d) While Hadoop is used more and more widely, but MapReduce framework does not fit all cases. When MapReduce deal with some complicated tasks such as directed acyclic graph (DAG), it should be split into multiple jobs, which not only increases the cost of initializing operations, but increases programming complexity. Due to this, some more excellent general-purpose computing framework or other special-purpose computing frameworks (such as stream computing, diagram calculation) appeared. Considering the Hadoop has been widely deployed, many companies require Hadoop can support more computing framework except for Mapreduce.



Figure 4.2: Map Reduce Usage Comparison

So, the limitations of classical Mapreduce mainly are: 1) Limited scalability; 2) Poor resource utilization; 3) Lack of support for the alternative framework; 4) Lack of wire-compatible protocols¹.

The direct reason is that in MapReduce version1, resources have to be divided into map and reduce slots and hard-coded in the TaskTracker's configuration. After setting, TaskTracker cannot run more map tasks than "mapred.tasktracker.map.tasks.maximum" at any given moment, even though no reduce task is running. To change the settings, TaskTracker must be restarted (which causes failure of tasks running on this node).

Here is the code: "mapred.tasktracker.map.tasks.maximum and mapred.tasktracker.reduce.tasks.maximum properties".

In addition, here are some radical reasons:

First, the computing resources, such as CPU cores (the basic computing units), are statically configured by administrator in advance and abstracted into map and reduce slots. A MapReduce job execution has two unique characters: (I) The slots allocation constraint assumption that map slots can only be allocated to map tasks while reduce slots can only be allocated to reduce tasks; (II) The general execution constraint that map tasks are only executed before reduce tasks. Due to these characters, two observations are as follows: 1) there are considerably different performances and system utilization for MapReduce workload under different slot configurations; 2) even under the optimal MapReduce slot configuration, there can be many idle reduce slots, which adversely affects the system utilization and performance.

¹<http://hakunamapdata.com/be-map-slot-or-not-to-be-that-is-the-question/>

Second, due to unavoidable runtime contention for memory, processor, network bandwidth and other resources, there can be straggled map and reduce tasks, causing dramatic delay of the whole job.

Third, data locality maximization is significant for the efficiency of slot utilization and performance improvement of MapReduce workloads. However, there are often conflicts between fairness and data locality in a shared Hadoop cluster among multiple users. [Tang et al., 2014]

The large Hadoop clusters revealed a limitation involving a scalability bottleneck caused by having a single JobTracker. According to Yahoo!, the practical limits of such a design are reached with a cluster of 5,000 nodes and 40,000 tasks running concurrently. Due to this limitation, smaller and less-powerful clusters had to be created and maintained. Moreover, both smaller and larger Hadoop clusters had never used their computational resources with optimum efficiency. In Hadoop MapReduce, the computational resources on each slave node are divided by a cluster administrator into a fixed number of map and reduce slots, which are not fungible. With the number of map and reduce slots set, a node cannot run more map tasks than map slots at any given moment, even if no reduce tasks are running. It harms the cluster utilization because when all map slots are taken (and we still want more), we cannot use any reduce slots, even if they are available, or vice versa. Last, but not least, Hadoop was designed to run MapReduce jobs only. With the advent of alternative programming models (such as graph processing provided by Apache Giraph), there was an increasing need to support programming paradigms besides MapReduce that could run on the same cluster and share resources in an efficient and fair manner. In 2010, engineers at Yahoo! began working on a completely new architecture of Hadoop that addresses all the limitations above and multiple additional features.

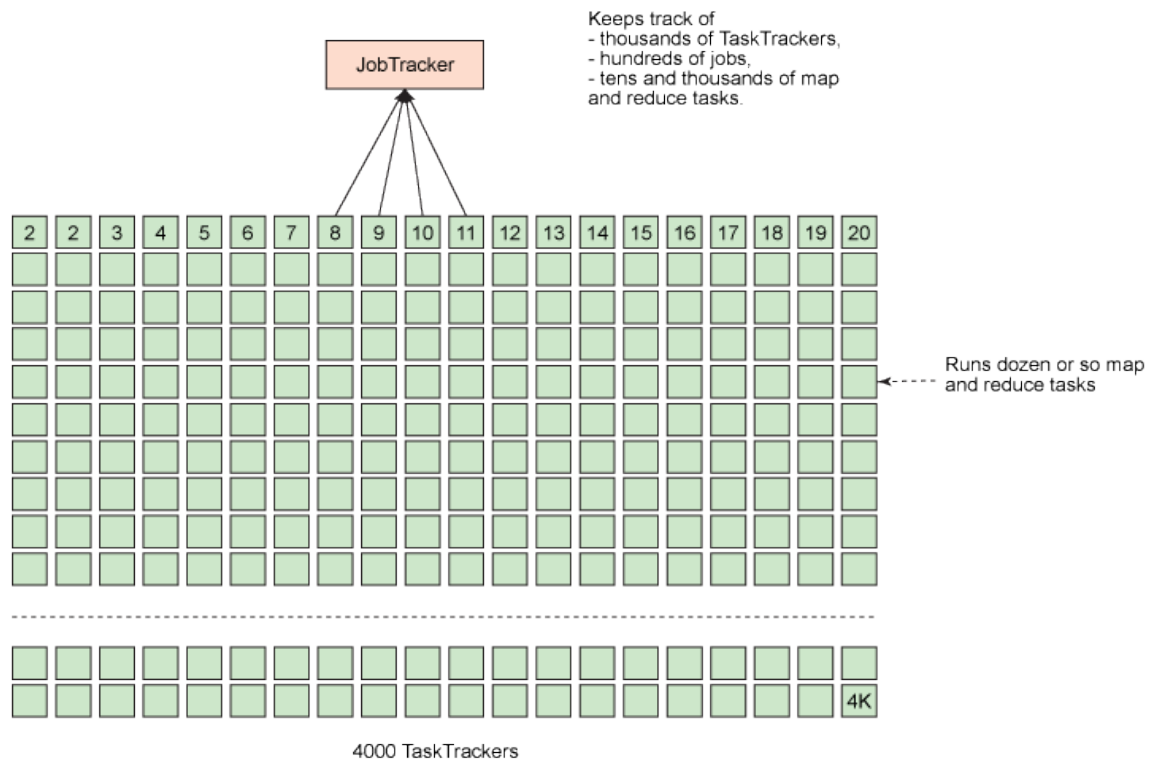


Figure 4.3: Busy Job Tracker On Large Cluster

In Hadoop MapReduce, the JobTracker is charged with two distinct responsibilities:

- Management of computational resources in the cluster, which involves maintaining the list of live nodes, the list of available and occupied map and reduce slots, and allocating the available slots to appropriate jobs and tasks according to selected scheduling policy.
- Coordination of all tasks running on a cluster, which involves instructing TaskTrackers to start map and reduce tasks, monitoring the execution of the tasks, restarting failed tasks, speculatively running slow tasks, calculating total values of job counters, and more. The large number of responsibilities given to a single process caused significant scalability issues, especially on a larger cluster where the JobTracker had to constantly keep track of thousands of TaskTrackers, hundreds of jobs, and tens of thousands of map and reduce tasks. The image below illustrates the issue. On the contrary, the TaskTrackers usually run only a dozen or so tasks, which were assigned to them by the hard-working JobTracker. To address the scalability issue, a simple but brilliant idea was proposed: Let's somehow reduce the responsibilities of the single JobTracker and delegate some of them to the TaskTrackers since there are many of them in a cluster. This concept was reflected in a new design by separating

dual responsibilities of the JobTracker (cluster resource management and task coordination) into two distinct types of processes. Instead of having a single JobTracker, a new approach introduces a cluster manager with the sole responsibility of tracking live nodes and available resources in the cluster and assigning them to the tasks. For each job submitted to a cluster, a dedicated and short-living JobTracker is started to control the execution of tasks within that job only. Interestingly, the short-living JobTrackers are started by the TaskTrackers running on slave nodes. Thus, the Coordination of a job's life cycle is spread across all of the available machines in the cluster. Thanks to this behavior, more jobs can run in parallel and scalability is dramatically increased.

4.1.2 Hadoop Version2 on YARN

In MapReduce version1, resources have to be divided into map and reduce slots and hard-coded in the TaskTrackers configuration("mapred.tasktracker.map.tasks.maximum and mapred.tasktracker.reduce.tasks.maximum properties"). After setting, TaskTracker cannot run more map tasks than "mapred.tasktracker.map.tasks.maximum" at any given moment, even though no reduce task is running. To change the settings, TaskTracker must be restarted (which causes failure of tasks running on this node).

Obviously, the situation could be alleviated, if we had simply task slots and a smart scheduler (e.g. mapred.tasktracker.tasks.maximum) dynamically making a decision about how many map and reduce tasks should be run on a given node. This idea is actually implemented on yarn, which provides much better resource utilization in cluster than mrv1 (compared with other advantages). In general, it is difficult for finding ideal hard-coded values for the maximum number of map and reduce tasks running on the cluster, as the workload may change very frequently. In some moment bottleneck is caused by map tasks, while in some other moments bottleneck is caused by reduce tasks. Apache Hadoop 2.0 includes yarn, which separates the resource management and other processing components. The yarn-based architecture is not constrained to mapreduce. Yarn and its advantages are described over the previous distributed processing layer in hadoop, and enhance the clusters with its scalability, efficiency, and flexibility.

The following name changes give a bit of insight into the design of YARN²:

²[HTTP://http://www.ibm.com/developerworks/library/bd-yarn-intro/](http://http://www.ibm.com/developerworks/library/bd-yarn-intro/)

- ResourceManager instead of a cluster manager
- ApplicationMaster instead of a dedicated and short-lived JobTracker
- NodeManager instead of TaskTracker
- A distributed application instead of a MapReduce job YARN is the next generation of Hadoop's compute platform, as shown below.

4.1.3 Benefits of Hadoop on YARN

First, because the architecture changed, the Job Tracker (resource management and job scheduling/monitoring in Mapreduce) changes into two separate entities: a global Resource Manager, and a per-application Application Master (YARN, the performance make a great improvement. The Resource Manager, the Node Manager, and a container are not concerned about the type of application or task. All application framework-specific code is simply moved to its ApplicationMaster so that any distributed framework can be supported by YARN as long as someone implements an appropriate ApplicationMaster for it. Thanks to this generic approach, the dream of a Hadoop YARN cluster running many various workloads comes true. Imagine: a single Hadoop cluster in your data center that can run MapReduce, Giraph, Storm, Spark, Tez/Impala, MPI, and more. The single-cluster approach obviously provides a number of advantages, including:

- One cluster can run lots of alternative distributed application;
- Higher cluster utilization, whereby resources not used by one framework could be consumed by another;
- Lower operational costs, because only one "do-it-all" cluster needs to be managed and tuned;
- Reduced data motion, as there's no need to move data between Hadoop YARN and systems running on different clusters of machines.

Managing a single cluster also results in a better solution to data processing. Less data center space is used, less power used, less silicon wasted, and less carbon emitted simply as we run the same calculation on a smaller but more efficient Hadoop cluster.

Storm-YARN: It enables Storm applications to utilize the computational resources in a Hadoop cluster along with accessing Hadoop storage resources such as HBase and HDFS. Collocating Storm with YARN offers advantages over segregated clusters as following.

- **Resilient computing resources:** Collocating Storm with YARN, it can share resources with other applications in a cluster (such as MapReduce batch processing). In this case, more resources can be allocated to Storm dynamically when the workload soar, while on the contrary, available resources can be released when the workload decrease, so that these resources will be temporarily allocated to the overload batch processing.
- **Sharing the storage:** Storm shares HDFS with other frameworks running on YARN, which lower the costs of maintaining multiple clusters, as well as avoid time delay caused by data replicating across different clusters of network.
- **Supporting multiple versions:** Different versions of Storm can run on YARN simultaneously, which avoid maintenance costs caused by one storm in one cluster.

Spark-YARN: Spark on YARN also has benefit, due to Spark itself only provides job management capabilities, so it relies on third-party resource scheduling systems, such as YARN or Mesos. The reason why YARN is more popular currently mainly because it has gradually become a standard resource management system as its strong community supporting function³.

4.2 Comparison of Processing Engine

Data generated mainly from following aspects:

- **E-commerce:** With the rapid development of technology and the popularity of internet, it is convenient necessary for people to get access to information. At the same time, the efficiency of information is highly required. For example, when a seller posted a product information, he expects that the information could be immediately shown, clicked, and the product could be purchased more; on the contrary, if the product is found out after a long time, it may lost its' popularity, which may a

³<http://spark.apache.org/docs/0.6.0/running-on-yarn.html>

great loss for the seller. Another example, if a user who bought a pair of socks on Taobao yesterday, wants to buy a pair of glasses to go swimming today, but the system always recommend him socks, shoes instead of swimming glasses(Due to the background system process the whole data once a day, usually at the end of a day, it is reasonable that the system make reflection the next day.), However, at that time, buyer will feel the information useless. Alibaba adopt real-time analysing system, rewriting a JStorm, which use Storm's core class name, has good fault-tolerance and stability. According to data generated by users' query logs and browsing histories, system make recommendations, which bring more flow and income to companies. The best advertisement is to recommend the users product which may be interested in, or they need but did not realize.

- **News:** The efficiency of news is highly required, which means news will become useless and worthless after a period. A proper processing engine can get useful information and recommend to users immediately after an event happened, and it will bring considerable page view.
- **Social Network System:** It generated mass data every moment. Real-time processing makes it possible that information could be feedback to drivers in time so that they can know the road condition on time. For example, avoid the crowded road or traffic accident.
- **Transportation System:** Transportation system generate big multimedia data every moment, and real-time processing make it possible that information could be feedback to users in time so that they can avoid traffic jam in certain extremely crowded road.
- **Data Mining and Machine Learning:** Data Mining and Machine Learning, which are the core platforms in system used by Internet companies, mainly provide data support for online services. So the efficiency is most important in the system, which means the ideal condition is the vast amounts of data generated daily could be processed timely, and the accumulated data (includes all the data generated in the past) could be updated effectively.

According to the data generation type from above four types, high speed and efficiency is highly required. So the processing time is becoming a more and more important index, compared Streaming processing and batch processing, here is a table shows the characters.

	streaming processing	batch processing
Input	stream of new data or updates	data chunks
Data size	infinite or unknown in advance	known & finite
Storage	not store or store non-trivial portion in memory	store
Hardware	typical single limited amount of memory	multiple CPUs, memories
Processing	a single or few pass(es) over data	processed in multiple rounds
Time	a few seconds or even milliseconds	much longer
Applications	web mining, sensor networks, traffic monitoring	widely adopted in almost every domain

Figure 4.4: Streaming Processing VS Batch Processing

4.2.1 Benefits of Stream Processing

Nowadays, more and more companies generate a large number of TB-level data daily, so how to analysis these huge amount data and use what kind of processing engine becoming more and more important. Streaming processing system play a vital characters among Internet companies , especially in the online massive data processing, which has a higher requirements of low latency and high reliability. Online system is the core of the Internet companies, especially the big company (Facebook, Twitter, Tencent qq, etc), and its' quality directly affects the flow, which is the most significant index of company. Speed and accuracy are the most significant indexes from data processing layer, such as logs and user behavior, especially for highly effective data, such as hot news, electronic business promotions, and micro-blog hot words which need to reflect in a very short period of time to satisfy users' demand or the companies' profits.

4.2.2 Storm

- **Design of Storm** The computing logic in Storm is called Topology. Spout is the data resources of Topology, and the data resources may be a log file, a message queue, or a table in the database, and so on. The first element which is reading the stream of data in Figure 4.5 is spout. A reader which is listening to the queue and emitting messages for processing down the line and can also perform other important functions, such as transforming incoming messages into storm's tuples(a serializable data structure). In addition, Spout could keep track of messages, and can resend failing messages. Spout then sends messages to the next element, which is called bolt and responsible for data maintaining. Bolt can do further transformation, such

as some calculations, saving to database, etc. The resources of Bolts come from a Spout or the joint from two Bolts.

In mapreduce work, we can have spout emitting messages and multiple bolts doing map-reduce job. Relationship between bolts can represent a tree. For example, imagine spout read from constant stream of text data, and converts raw text data into lines of text and emits each line for further processing to the next bolt. This bolt is responsible for breaking down line into words. Then it emits each word to the next bolt which in turns keeps track of how many times each word appeared in the text. It may pass the data to the next bolt which may just write data to some permanent data storage.

The core of Streaming in Storm is the abstract connection between Spout and Bolt, Bolt and Bolt dataflow, which is constituted by tuples. And Tuple is consisted of several Fields, which is setting when the Bolt is defined.

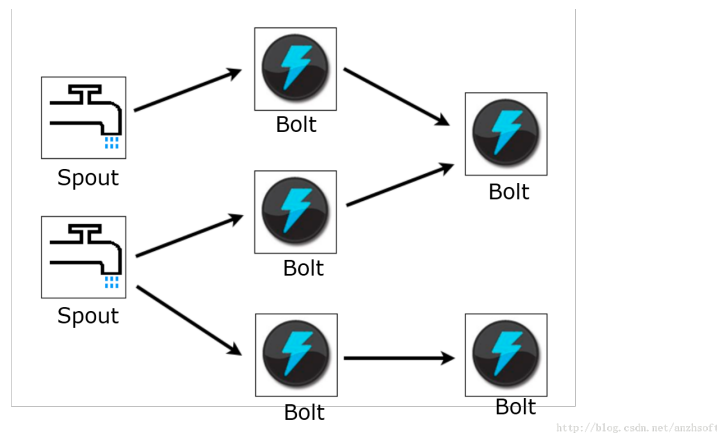


Figure 4.5: Storm Components

- **Message Sending Mechanism in Storm** In Storm, the message Sending mechanism is defined when the Topology is defined. There are main patterns of grouping:
 1. **Shuffle Grouping:** Tuples are randomly distributed across the tasks of bolts in a way that each bolt is guaranteed to get an equal number of tuples.
 2. **Fields Grouping:** The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, while tuples with different "user-id"'s may go to different tasks.
 3. **All Grouping:** The stream is replicated across all the tasks of bolts. Use this grouping with caution, or it will cause great waste of resources.

4. Global Grouping: The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.

5. None Grouping: None groupings are equivalent to shuffle groupings. Eventually though, Storm will push bolts down with none groupings to execute in the same thread as the bolt or spout they subscribe from.

6. Direct Grouping: Direct grouping. It is a more specific grouping, which means that the sender of the message packet process the message specified by the task which the message container. This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams.

4.2.3 Spark

- **Design of Spark Streaming** Spark Streaming analyze tasks in a series of small batch jobs, that is the input data are divided into a mini-batch of the data(use Spark) according to batch size (eg. 1 second). Each mini-batch of data is converted to RDD (in Spark), and then the transformation of D-Stream in Spark Streaming convert to transformation of RDD in Spark. The RDD After the operation, RDD became the intermediate results stored in memory. In the entire stream computing intermediate results can be superimposed or store in an external storage device according to the commercial requirement. Following figure shows the entire process of Spark Streaming.

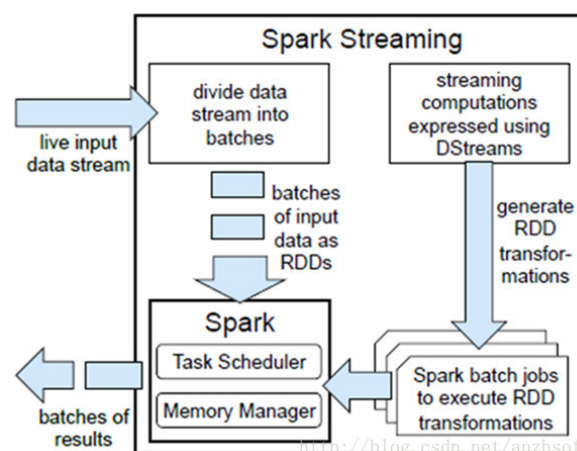


Figure 4.6: Spark Streaming Process

• Message Sending Mechanism in Spark

In Spark, the dataflow is first converted into the DAG through a series of user-defined RDD, and then DAG Scheduler convert the DAG into a TaskSet, which can apply for computing resources for the cluster. The cluster arranges this TaskSet to compute through Workers.

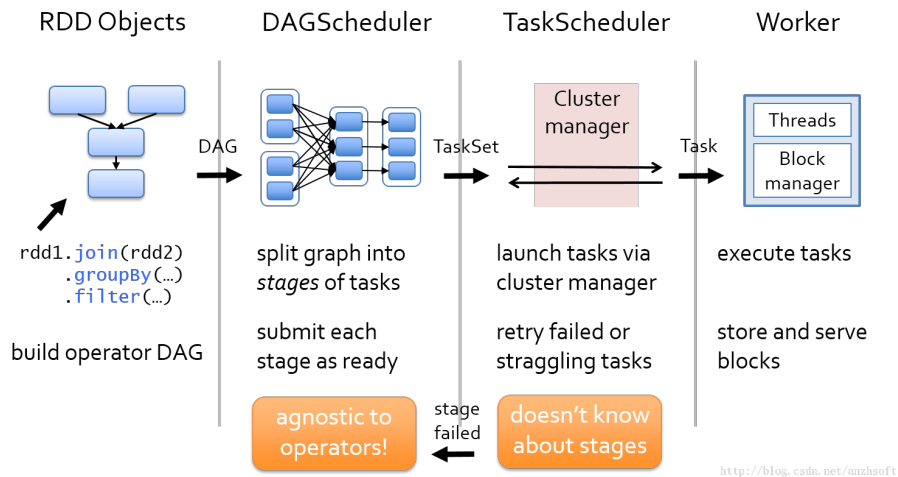


Figure 4.7: Message Sending Mechanism In Spark

4.2.4 Similarity and Difference between Storm and Spark Streaming

In terms of the similarity of Storm and Spark Streaming, both of them are open-source frameworks for distributed streaming processing, and can operate in a Hadoop cluster as well as have access to Hadoop storage. Storm-YARN is Yahoos open source implementation of Storm and Hadoop convergence. Spark is providing native integration, which is achieved through YARN with Hadoop. Integrating real-time analytics with Hadoop based systems allows for better utilization of cluster resources through computational resilience and flexibility and being in the same cluster means that network transfers can be minimal. Both frameworks provide fault tolerance and scalability, while there are also some differences between them.

Storm vs. Spark Streaming



Criteria	 STORM	
Processing Model	Record at a time	Mini batches
Latency	Sub second	Few seconds
Fault tolerance—every record processed	At least one (may be duplicates)	Exactly one
Batch Framework integration	Not available	Core Spark API
Supported languages	Any programming language	Scala, Java, Python

Figure 4.8: Storm Vs Spark Streaming Comparison

- **Processing Model:** Storm processes one record at a time into memory, and performs the transformation of the record and writes the target to disk. while Spark Streaming split the tasks into mini batches.
- **Latency:** Storm processes incoming events one-at-a-time, so the latency is in sub second. While Spark batches events up which arrive in a short time window before processing them so that it will take few seconds for each tuple. Therefore, Storm can achieve sub-second latency in processing an event, whereas Spark Streaming needs several seconds.
- **Fault-Tolerance:** Both Storm and Spark provide scalability and fault-tolerance, but they differ in their processing model fundamentally. On the one hand, Spark Streaming provides a better support for fault tolerant in computation. While in Storm, each single record has to be tracked as it moves through the system, so Storm only guarantees that each record will be processed at least once, but allows duplicates to appear during recovery from a fault, which means mutable state may be incorrectly updated twice.

On the other hand, Spark Streaming only need track processing at the batch level, so it could efficiently guarantee that each mini-batch will be processed exactly once, even if a fault such as a node failure occurs. Whereas, Storm's Trident library also provides exactly once processing, but it relies on transactions to update state, which make processing slower and has to be implemented frequently by the user.

Storm is a better choice for sub-second latency and no data loss. While Spark Streaming is better for those who need stateful computation, with the guarantee

that each event is processed exactly once. In addition, programming logic of Spark Streaming is also simpler because it is similar to batch programming (Hadoop), especially in small batches.

- **Implementation and Programming API**

Storm is primarily implemented in Clojure, and comes with a Java API as well as support for other languages. While Spark Streaming uses Scala as well as Java. Storm was developed at BackType and Twitter; while Spark Streaming was developed in UC Berkeley. Another good point of Spark Streaming is that it runs on Spark. Thus, the same (or very similar) code can be used in writing batch processing and/or interactive queries in Spark, on Spark Streaming. So there is no need to write separate code to process streaming data and historical data.

- **Cluster Manager Integration** Both systems can run on their own clusters. Specifically, Storm also runs on any programming language, while Spark Streaming runs on both YARN and Mesos.

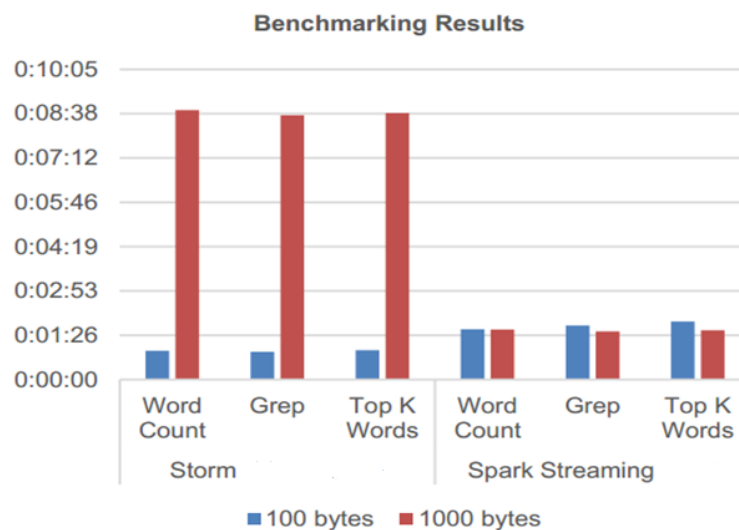


Figure 4.9: Benchmarking Results

As the experiment shows Storm can achieve sub-second when the data size is small (around 100 bytes), which cost half time compared with Spark Streaming. While when the data is 10 times larger around 1000 bytes, according to the latency, Storm uses much more time than Spark Streaming (uses nearly same time in dealing with small data).

5

Conclusion

In this chapter, we conclude the thesis with a summary of the contribution, some discussions, and future research directions.

5.1 Summary of Contributions

Recent technological advancements lead to a deluge of data from different domains, such as health care and scientific sensors, Internet and commercial companies, user-generated data, as well as supply chain systems over the past decades. In this Thesis, at first we introduce the background knowledge of cloud computing and the increasing data generation, acquisition, the requirement of accuracy in storage, and the speed in analytics. According to these requirements, the disadvantages outweigh the advantages. We introduce a new Hadoop architecture YARN, which is compatible to many API in contribution. Finally, we make comparison of batch processing with streaming processing, as well as comparing several different streaming processing engines.

The contributions of each chapter are summarised as follows:

In Chapter 2, we presented an overview of some background information including the architectures and characteristics of Cloud Computing and big data as well as the design theory and 4 types of NoSQL Databases.

In Chapter 3, we reviewed the literature of the architecture of Hadoop and the main component in Hadoop, including HDFS and HBase (how to replicate and store data), MapReduce (how it computes batches data), YARN (Next generation of Mapreduce), Pig and Hive, Storm (how to process real time streaming data), Spark (a fast and general-purpose cluster computing system), and Shark (a port of Apache Hive onto Spark). HDFS and HBase belong to the basic storage layer while Mapreduce is the computing layer above the storage layer. Also in this Chapter, we introduced YARN, which can be seen as an improved version of the previous architecture. Using resource manager rather than 'Job tracker' and task tracker, YARN avoids some failures and provides support for more processing engines, such as streaming processing engine Storm and Spark.

In Chapter 4, we analysed and compared Hadoop veision1 with version2 on YARN, as well as compared processing engine, especially Storm with Spark Streaming side by side.

More specifically, this Thesis makes the following contributions:

- We investigated the data resources provenance and using different processing engine dealing with different type of data to achieve different requirements.
- We identified the benefits of processing data on YARN and the benefits of Streaming processing.
- We highlighted the similarity and compared different processing engines and in experimental evaluation evaluated their speed.

5.2 Future Work

It is feasible that choosing a proper API (Spark or Storm on YARN) processing data would better satisfy the needs rather than optimizing Mapreduce. Though some researchers are still doing research on optimizing Mapreduce architecture by recoding (such as avoiding hard coding, preserving more interface, etc.) and using extremely complex algorithm, it would be interesting to investigate would inserting API into existing framework will be a

better and easier solution, and would it be also more efficient. Therefore, in the future work, we will work on the combination of different application program interfaces with YARN or NoSQL database directly. It would be interesting to test more parameters and conduct more detail experiments to proof this hypothesis.

List of References

- [Borthakur, 2008] Borthakur, D. (2008). Hdfs architecture guide. *Hadoop Apache Project*, page 53.
- [Borthakur et al., 2011] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., et al. (2011). Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM.
- [Chen and Zhang, 2014] Chen, C. P. and Zhang, C.-Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347.
- [Chodorow, 2013] Chodorow, K. (2013). *MongoDB: the definitive guide.* ” O’Reilly Media, Inc.”.
- [Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.
- [Deni Raj et al., 2014] Deni Raj, E., Nivash, J., Nirmala, M., and Dhinesh Babu, L. (2014). A scalable cloud computing deployment framework for efficient mapreduce operations using apache yarn. In *Information Communication and Embedded Systems (ICICES), 2014 International Conference on*, pages 1–6. IEEE.
- [Foster et al., 2008] Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE’08*, pages 1–10. Ieee.
- [Frampton, 2015] Frampton, M. (2015). Cluster management. In *Big Data Made Easy*, pages 225–256. Springer.

- [Grolinger et al., 2013] Grolinger, K., Higashino, W. A., Tiwari, A., and Capretz, M. A. (2013). Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22.
- [Han et al., 2011] Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE.
- [Hu et al., 2014] Hu, H., Wen, Y., Chua, T., and Li, X. (2014). Towards scalable systems for big data analytics: A technology tutorial.
- [Kim et al., 2014] Kim, S. D., Lee, S. M., Lee, S. M., Jang, J. H., Son, J.-G., Kim, Y. H., and Lee, S. E. (2014). Compression accelerator for hadoop appliance. In *Internet of Vehicles Technologies and Services*, pages 416–423. Springer.
- [Lam, 2010] Lam, C. (2010). *Hadoop in action*. Manning Publications Co.
- [Polo et al., 2013] Polo, J., Becerra, Y., Carrera, D., Steinder, M., Whalley, I., Torres, J., and Ayguadé, E. (2013). Deadline-based mapreduce workload management. *Network and Service Management, IEEE Transactions on*, 10(2):231–244.
- [Rumi et al., 2014] Rumi, G., Colella, C., and Ardagna, D. (2014). Optimization techniques within the hadoop eco-system: A survey. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 437–444. IEEE.
- [Shvachko et al., 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE.
- [Stantic and Pokorny, 2014] Stantic, B. and Pokorny, J. (2014). Opportunities in big data management and processing. In *Databases and Information Systems VIII: Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014*, volume 270, page 15. IOS Press.
- [Tang et al., 2014] Tang, S., Lee, B., and He, B. (2014). Dynamicmr: A dynamic slot allocation optimization framework for mapreduce clusters.
- [Vavilapalli et al., 2013] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). A-

- pache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.
- [Wang and Tang, 2012] Wang, G. and Tang, J. (2012). The nosql principles and basic application of cassandra model. In *Computer Science & Service System (CSSS), 2012 International Conference on*, pages 1332–1335. IEEE.
- [White, 2009] White, T. (2009). *Hadoop: the definitive guide: the definitive guide.* ” O’Reilly Media, Inc.”.
- [Zhang et al., 2010] Zhang, S., Zhang, S., Chen, X., and Huo, X. (2010). Cloud computing research and development trend. In *Future Networks, 2010. ICFN’10. Second International Conference on*, pages 93–97. IEEE.
- [Zikopoulos et al., 2011] Zikopoulos, P., Eaton, C., et al. (2011). *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media.