Distributed Computing Technologies in Big Data Analytics

Kaushik Dutta

1 Introduction

The database technology has evolved over time. As the application of database has extended from simple mainframe to desktop application to web application to mobile application, the size of data to store and manage through database has also increased. Figure 1 depicts this growth of the data. The first generation of data growth came from ERP software and following that with the introduction of CRM. Next, the introduction of web moved the data volume to terabyte range. However, with the mobile, sensor and social media based applications, the data volume is growing in the range of petabytes.

Relational database (RDBMS) has been one of the most successful database technology since the 1980s. However, even with its solid technological growth, the relational database has failed to scale with the growth of data. Despite the advances in computing, faster processors and high-speed networks, the scalability of the relational database has been restricted. The applications built using RDBMS technology either has failed to perform with increased data or the cost of the infrastructure to keep the application performing has grown exponentially.

Secondly, the relational database was designed for tabular data with a consistent structure and fixed schema. Relational database works best when the structure of the data is known beforehand. However, in the new world as the volume and velocity of the data are increasing, so is the variety and complexity of data. Applications need to be built into the database without the full understanding of the data to be stored and the structure of the data. Or, the structure of the data is being changed after the application has been built. For example, consider a retail application that is selling electronic goods. It can develop applications that can search and manage the known

K. Dutta (✉)
University of South Florida, Tampa, FL, USA
e-mail: duttak@usf.edu

S. Mazumder et al. (eds.), *Distributed Computing in Big Data Analytics*,
Scalable Computing and Communications, DOI 10.1007/978-3-319-59834-5_4

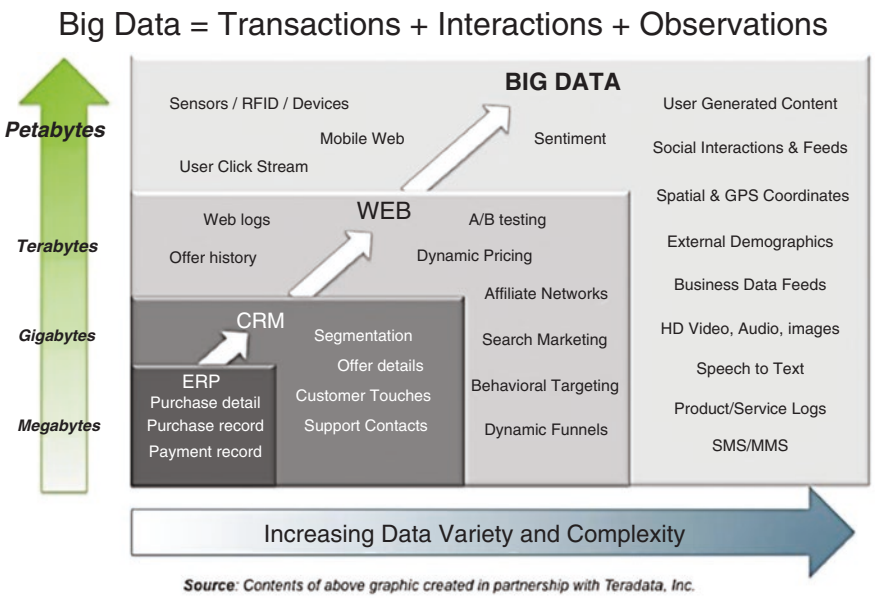


Fig. 1 Data growth

set of electronic goods. However, if in the future a new device comes up (such a brain-reader) with a new set of features and specifications, the applications will not know how to store and manage that device through a relational database. The relational structure does not allow to handle such unstructuredness with the data.

In the next section, we describe the fundamentals of distributed database and the basics of the no-SQL database. In describing the technologies, we rely on few software platforms supporting such tools. Though there are many different software platforms supporting the similar technologies, mostly we have chosen the software that is popular and preferably supported by open source platforms. The database discussion is followed by the distributed file system and distributed computing platform such as map-reduce and spark. Next, we follow the discussion on how these distributed technologies are being used to develop a newer generation of machine learning platforms. The textual document is one of the important sources of today's information. We describe the basics of textual search platform and associated software such as Lucene and ElasticSearch. Distributed caching enhances the performance of real-time access to data. We describe the distributed caching systems such as REDIS. As a number of components and systems grow exponentially in big data infrastructure, the communication across these components need to be managed more efficiently. In this context, we describe the message passing software such as RabbitMQ and Kafka. Lastly, the traditional tools are unable to represent big data visually. The Newer generation of visualization tools is being developed to present the data. We describe these big data visualization tools in the later part of this chapter.

2 Distributed Database

The scalability issues in the relational database come with the ACID property. The ACID (Atomicity, Concurrency, Isolation, and Durability) property ensures the consistency of data and helps to execute transactions in databases. Database vendors long ago recognized the need for partitioning databases and introduced a technique known as 2PC (two-phase commit) for providing ACID across multiple database instances [35].

It is relatively easy to maintain the ACID property in a single server database system or even with a two node master-slave database server. However, as the data volume grows, it becomes necessary to distribute the data across multiple nodes. With multiple nodes, the cost of communication to maintain ACID property increases. Also, the availability of any system is the product of the availability of the components required for operation. A transaction involving two databases will have the availability of the product of the availability of each database. For example, if we assume each database has 99.9% availability, then the availability of the transaction becomes 99.8%, or an additional downtime of 43 min per month [35].

This leads us to an important barrier of distributed system – Brewer's theorem [12] on the correlation between consistency, availability, and partition-tolerance. Brewer postulates three distinct properties for distributed systems with an inherent correlation [18].

Consistency The consistency property describes a consistent view of data on all nodes of the distributed system. That is, the system assures that operations have an atomic characteristic and changes are disseminated simultaneously to all nodes, yielding the same results.

Availability This property demands the system to eventually answer every request, even in the case of failures. This must be true for both read and write operations.

Partition Tolerance This property describes the fact that the system is resilient to message losses between nodes. And according to the availability property, every node of any potential partition must be able to respond to a request.

The core statement of Brewer's theorem is: "*You can have at most two of these properties for any shared-data system.*" (Fig. 2).

Though all the above properties in a distributed database system are desirable, any two of these three properties can be achieved [20].

In a distributed database the data is distributed across multiple geographical sites (as depicted in Fig. 3). In the new era of globalization, distributed database has become a common scenario due to several reasons – (1) Support the distributed Nature of Organizational Units (2) Support the need for Sharing of Data across multiple units and (3) Support for Multiple Application Software. Most of the COTS (commercial off-the-shelf) database server has a distributed version. For example, Oracle has a distributed database since Oracle 7. MySQL Cluster is the distributed version of MySQL Database. IBM's DB2 has several versions of distributed base as part of the DB2 package. However, in the spirit of maintaining

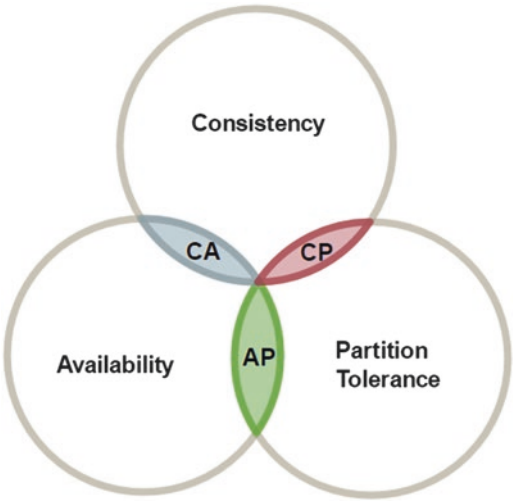


Fig. 2 Different properties that a distributed system can guarantee at the same time (Courtesy: Benjamin Erb [18])

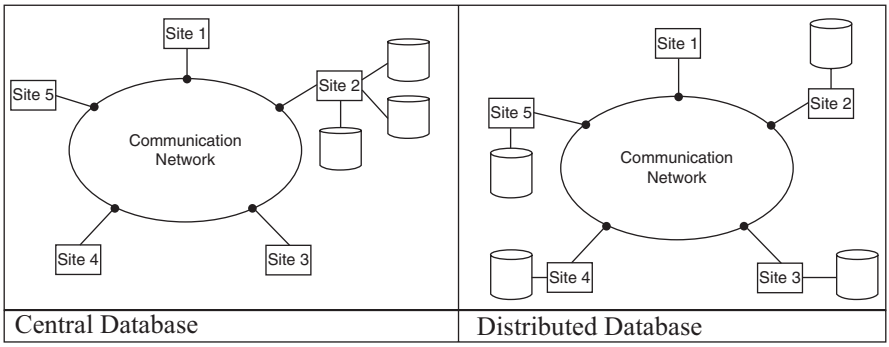


Fig. 3 Distributed database [34]

the ACID property, all these relational models distributed database system has given up either the partition tolerance or the availability for consistency. When network separation happens across multiple sites in the distributed relational database, the database fails to serve for that portion of data.

2.1 NoSQL Database

In recent years, a new generation of the database has come up to handle the issues as discussed above with distributed relational database. It is NoSQL database. As the name suggests, it does not follow the relational structure – that allows to store and manage unstructured/semi-structured and unknown data structures. NoSQL systems are distributed, non-relational databases designed for large-scale data

storage and massively-parallel data processing across a large number of commodity servers. The No-SQL database break through conventional RDBMS performance limits by employing NoSQL-style features such as relaxed ACID property, and schema-free database design. Unlike relational databases, NoSQL database has loosened up the consistency requirements to achieve better availability and partitioning [35].

There are three types of No-SQL databases [29].

Key-Value Stores As the name implies, a key-value store is a system that stores value indexed for retrieval by keys. These systems can hold structured or unstructured data. Typically, these database store items as alphanumeric identifiers (keys) and associated values in simple, standalone tables (referred to as ——hash tables). The values may be simple text strings or more complex lists and sets. Data searches can usually only be performed against keys, not values, and are limited to exact matches [29].

The simplicity of Key-Value Stores makes them ideally suited to lightning-fast, highly scalable retrieval of the values needed for application tasks like managing user profiles or sessions or retrieving product names. This is why Amazon makes extensive use of its Key-Value system, Dynamo, in its shopping cart. Dynamo is a highly available key-value storage system that some of Amazon's core services use to provide highly available and scalable distributed data store [16].

The examples in this category include Amazon's Dynamo [3, 16], Aerospike [2], BerkleyDB (now Oracle No-SQL database) [33] and Riak [11] (Table 1).

Document-Based Stores These databases store and organize data as collections of documents, rather than as structured tables with uniform-sized fields for each record. With these databases, users can add any number of fields of any length to a document. It is designed to manage and store documents. These documents are encoded in a standard data exchange format such as XML, JSON (Javascript Option Notation) or BSON (Binary JSON). Unlike the simple key-value stores described above, the value column in document databases contains semi-structured data – specifically attribute name/value pairs. A single column can house hundreds of such attributes, and the number and type of attributes recorded can vary from row to row. Also, unlike simple key-value stores, both keys and values are fully searchable in document databases [29] (Table 2).

Document databases are good for storing and managing Big Data-size collections of literal documents, like text documents, email messages, and XML documents, as well as conceptual documents like denormalized (aggregate) representations of a

Table 1 Example of key-value store

Product ID (KEY)	Value (Product)
123112	Apple iPhone, 8GB, Gold
146177	Android, Samsung, Galaxy S7, 32GB, US Warranty, Lock Free
123112	Android, Samsung, Galaxy J7, Gold, Dual Sim

Table 2 Example of document database

{ “ProductID” : “123112”, “Manufacturer”: “Apple”, “Model” : “iPhone”, “Memory” : “8GB”, “Color” : Gold}
{ “ProductID” : “146177”, “Manufacturer”: “Samsung”, “OS” : “Android”, “Model” : “Galaxy S7”, “Memory” : “32GB”, “Warrantee” : “US Warrantee”, “Lock” : “Lock Free”}
{ “ProductID” : “123112”, “Manufacturer”: “Samsung”, “OS” : “Android”, “Model” : “Galaxy J7”, “Color” : “Gold”, “SIM” : “Dual Sim”}

database entity such as a product or customer. They are also good for storing sparse data in general, that is to say, irregular (semi-structured) data that would require an extensive use of nulls in an RDBMS (nulls being placeholders for missing or nonexistent values). The examples of document database are – CouchDB (JSON) [5] and MongoDB (BSON) [32].

Column-Oriented Database These types of database store sets of information in a heavily structured table of columns and rows with uniform-sized fields for each record, as is the case with relational databases, column-oriented databases contain one extendable column of closely related data. It employs a distributed, column-oriented data structure that accommodates multiple attributes per key. While some Wide Column (WC) /Column-Family (CF) stores have a Key-Value DNA (e.g., the Dynamo-inspired Cassandra), most are patterned after Google‘s Bigtable [13]. Google Bigtable is the petabyte-scale internal distributed data storage system Google developed for its search index and other collections like Google Earth and Google Finance. The tables with column-oriented databases are called column family [29] (Fig. 4).

This type of DMS is great for (1) Distributed data storage, especially versioned data because of WC/CF time-stamping functions. (2) Large-scale, batch-oriented data processing: sorting, parsing, conversion (e.g., conversions between hexadecimal, binary and decimal code values), algorithmic crunching, etc. (3) Exploratory and predictive analytics performed by expert statisticians and programmers. Examples of the Column-oriented database includes Cassandra and SimpleDB.

The Key-Value store databases are completely unstructured. The only query possible in key-value databases is given a key retrieving the value. The document

Product Table (Column Family)

Row Key: 123112

ProductID	*Manufacturer*	*Model*	*Memory*	*Color*
123112	Apple	iPhone	8GB	Gold

Row Key: 146177

ProductID	*Manufacturer*	*Model*	*Memory*	*OS*	*Warrantee*	*Lock*
146177	Samsung	Galaxy S7	32GB	Android	US	Lock Free

Row Key 123112

ProductID	*Manufacturer*	*Model*	*Color*	*OS*	*SIM*
123112	Samsung	Galaxy J7	Gold	Android	Dual SIM

Fig. 4 Example of column-oriented database

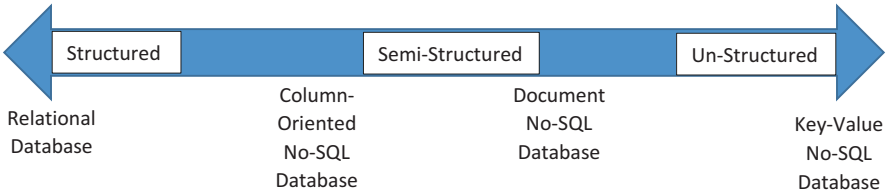
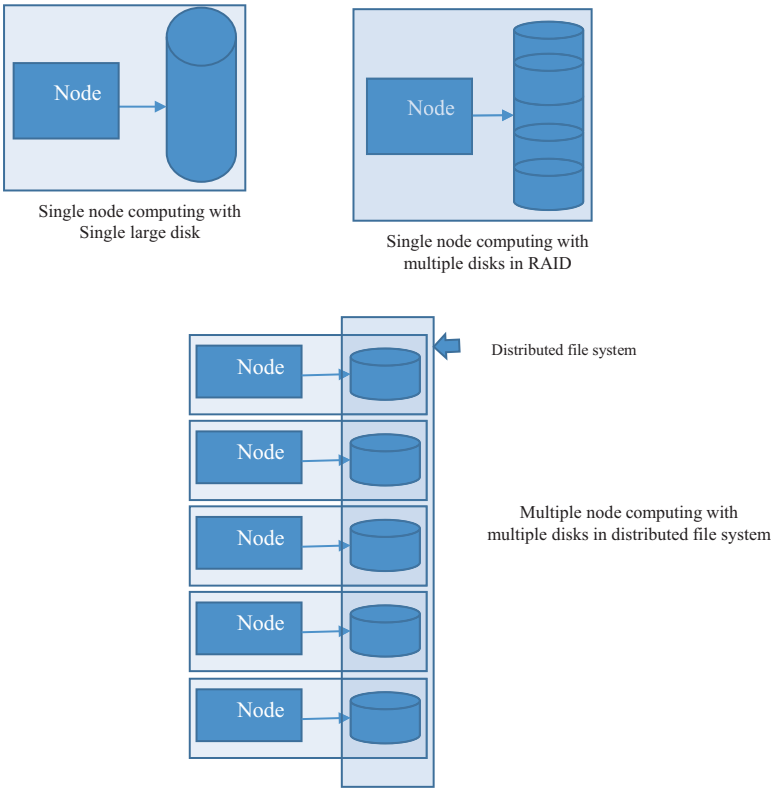


Fig. 5 No-SQL database types

database provides some structure in the value by providing a constraint that the value has to be in JSON or BSON (or any other standard format) format. Other than retrieving the value based on the key, in the document database, it is possible to query based on the content of the value. For example, in MongoDB, the JavaScript based query is used to run a complex query on the value. The column-oriented database has a table structure very similar to a relational database, however, unlike relational tables, the tables with the column-oriented database may have different rows in different columns in the same table. This makes the column-oriented database to handle semi-structured data, where data can be parsed and put into a structured format – but the structure may change from one data item to the next data item. Similar to relational databases, the column-oriented database has high-level query language very similar to SQL. For example, in Cassandra, we have CQL (Cassandra Query Language) [15]. Recently growing number of column-oriented No-SQL databases are implementing SQL-like query capability. Figure 5 depicts the sliding scale of structures in the data and where the different types of No-SQL database fall on this scale. The difference between these different types will be blurred as a growing number of products in one category will incorporate features from other categories.

3 Distributed Storage

Though No-SQL database grew up as a requirement to support the growth of data in volume and variety, not every application requires a database to store and manage the data. Documents, images, videos can be stored in the file system and processed with domain specific tools such as text parser and image processing software. As the size of data captured in these forms (i.e. Documents, images, and videos) are increasing, it became difficult to store and manage it in a single node computing system.



Traditionally a single node computing system has processed the data stored in local file system. RAID-based storage has come up to accommodate large volumes of data in the file system. The RAID also provides failover mechanism. However, still, only single node can process the data stored in RAID. Processing large volume of data in single node system has been almost impossible, just reading Terabytes of data from a hard disk by a single node computing machine will take several days – any processing on that data will increase that time considerably. To solve this issue the distributed file system has been developed, where the data is distributed across multiple local hard disks each associated with a separate computing node. In such a system, if the computation on the data in the distributed file

can be divided in such a way that each node does the processing on the data stored on its local hard disk and the processing in each of these nodes can be done in parallel – then we can complete the processing of terabytes of data in few minutes with the help several hundreds of such nodes. This motivated the development of distributed file system (such as HDFS (Hadoop distributed file system), GFS (Google File System), Amazon S3) and corresponding programming framework map-reduce and Spark. In this section, first we will discuss the distributed file system HDFS, then we will discuss the map-reduce programming framework, and lastly, we will discuss the Spark.

3.1 Hadoop Distributed File System (HDFS)

The HDFS is a distributed file system that spans across multiple nodes. Each of these nodes will have a local regular operating system (such as Linux), on top of which the HDFS file system is deployed. The interface to HDFS is patterned after the UNIX file system (Fig. 6).

HDFS store file system metadata and application data (i.e. the actual files) separately. It stores metadata on a dedicated server called NameNode. Application data are stored on other servers called DataNodes [39]. The DataNode in HDFS does not have any individual failover mechanism such as RAID. Rather the file content is replicated on multiple DataNode for reliability. This has the advantage of data being local to the node, where the computation will be carried out. This reduces the overhead associated with data transfers between the nodes for computational requirements. The GFS [19] has the similar structure (Fig. 7).

The HDFS namespace is a hierarchy of files and directories. File and directories are represented on the NameNode by inodes, which record attributes like permissions, modifications and access times, namespace and disk space quotas. The file content is split into blocks. Conceptually, this is very similar to regular file system blocks, but are much larger in size – typically 128 MB, but may be larger as selected by the user. These are HDFS blocks. Each HDFS block is replicated at multiple DataNodes. The NameNode maintains the namespace tree and the mapping of the HDFS blocks to DataNodes (the physical location of the HDFS block) [39] (Table 3, Fig. 8).

An HDFS client first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the

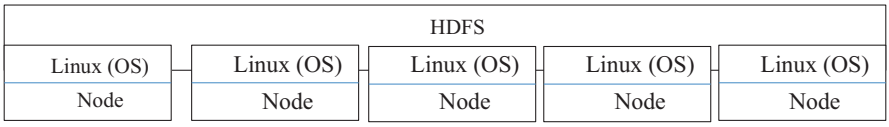


Fig. 6 HDFS on top of Linux

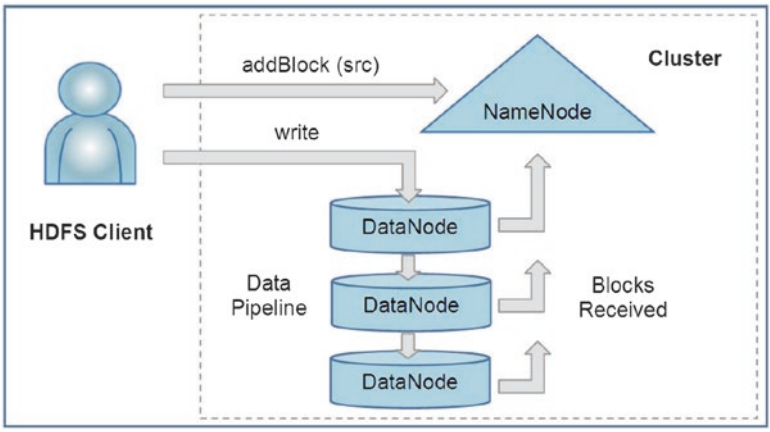


Fig. 7 HDFS architecture [39]

Table 3 NameNode metadata example

Filename	Number of replicas	Block-IDs
/usr/hue/test.dat	3	1, 3, 4, 6
/usr/hue/test2.dat	4	2, 5, 8, 9, 10

Block-ID	Location (DataNode) (Total number of DataNode = 10)
1	1, 3, 5
3	2, 4, 6
4	3, 5, 7
6	4, 6, 8
2	1, 3, 5, 7
5	2, 4, 6, 8,
8	3, 5, 7, 9,
9	4, 6, 8, 10

client. When writing the data, the client requests the NameNode to nominate a suite of DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. HDFS keeps the entire namespace in RAM.

Unlike conventional file system, HDFS provides an API that exposes the locations of a file block. This allows distributed programming like Map-Reduce framework to process data in a node locally where the data is located [39].

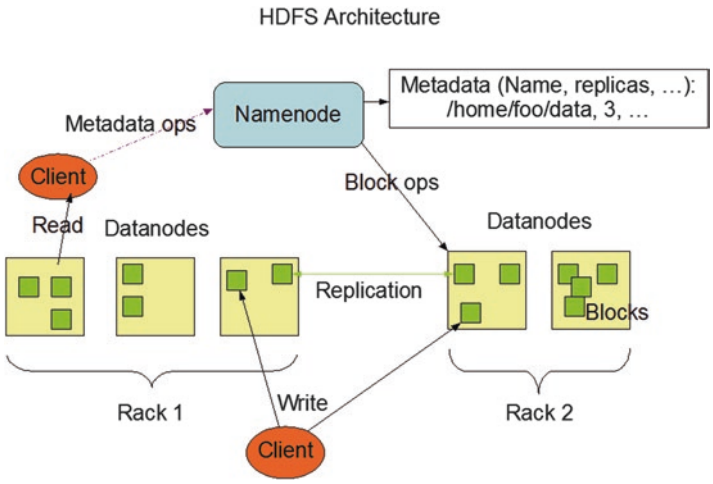


Fig. 8 HDFS architecture (Courtesy: Hortonworks Inc. [23])

4 Distributed Computation

Traditional parallel and distributed computation relied on synchronization and locking. However, the overhead of synchronization across multiple processes and locking the data has considerable overhead. Additionally, the traditional parallel and distributed computation have looked at the computation separately from the data. The assumptions that were made is the data resides in a database or any storage system that is equally accessible by multiple computing nodes. The parallel processing in these nodes will lock the data and process it. In addition to the overhead of locking such an approach adds tremendous overhead in transporting data from the data node (where the data is) to the processing node (the node that is processing the data). The Map-Reduce framework is a new parallel programming framework that addresses these issues with parallel and distributed computing. The Map-Reduce framework is based on two principles.

1. If the computation can be divided based on data segmentation, such that each computational node is processing a different part of the data, the requirement of lock and synchronization can be avoided. This will improve the performance of the parallel computation.
2. If the computation node processes data that is local to its node, then the overhead of data transmission from data node to computation node can be avoided.

Though the map-reduce framework was first developed as part of the Hadoop ecosystem along with the HDFS, the framework in generic and is applicable in a wider variety of data storage including No-SQL databases such as Cassandra and MongoDB.

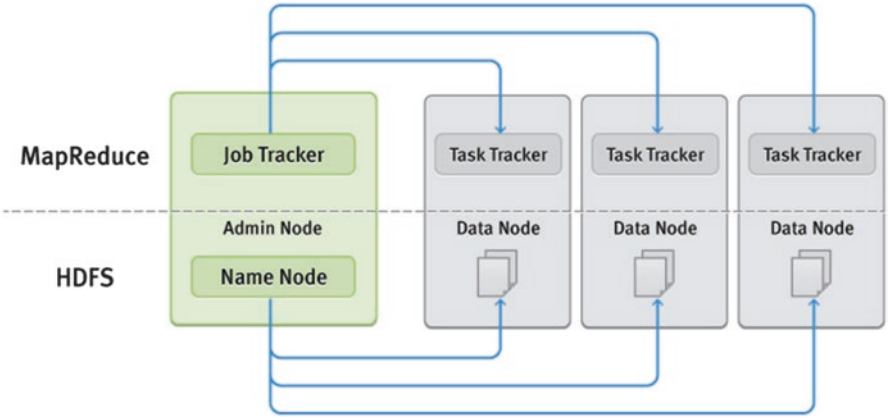


Fig. 9 Map-Reduce on HDFS (Courtesy: NDM Technologies)

4.1 Map-Reduce in Hadoop

Figure 9 depicts the map-reduce architecture on HDFS. The Job Tracker in Map-Reduce is responsible for breaking the job into multiple tasks and assigning to various nodes. The Task Trackers are responsible for completing a task. The Job Tracker and The Name Node of HDFS can coexist in the same node. The Task Tracker and the HDFS Data Node coexist in the HDFS Hadoop framework. Such coexistence allows the Task Tracker to process local data without transmitting the data from one node to another node. The Job Tracker distributes the jobs in such a way that task tracker processes only the local data as far as possible. In Hadoop 2.0 replaced the Job Tracker with Yarn, a separate software component to manage the tasks.

The programming framework of Map-Reduce is based on considering data not as a single unit, but as a collection of multiple units. The example of such collection is – a file is a collection of lines, a directory is a collection of files, a database table is a collection of multiple rows and so on. In Map-Reduce term this collection is considered as a map. Thus the input to map-reduce programming is a map. For example, if a map has N units and there are m task trackers, the Job Tracker can ideally provide N/m units to each task tracker to complete. Obviously, the division of tasks across task-tracker will seldom be so uniform due to non-uniform distribution of data across DataNodes (Fig. 10).

A map-reduce programming framework works in three steps.

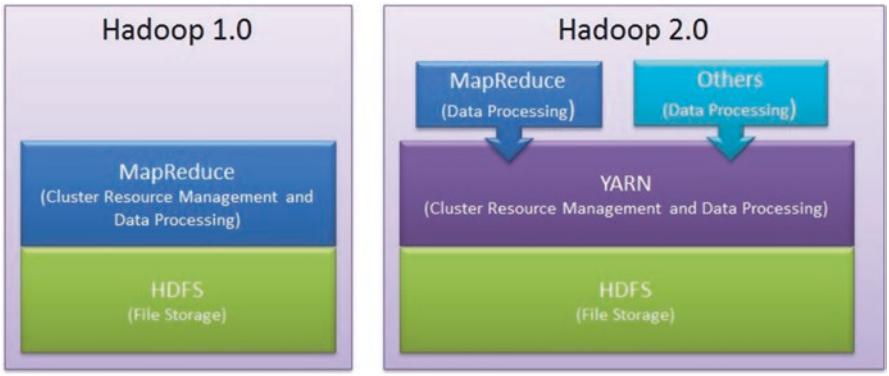


Fig. 10 Hadoop 1.0 vs. Hadoop 2.0 (Courtesy: Saphanatutorial [38])

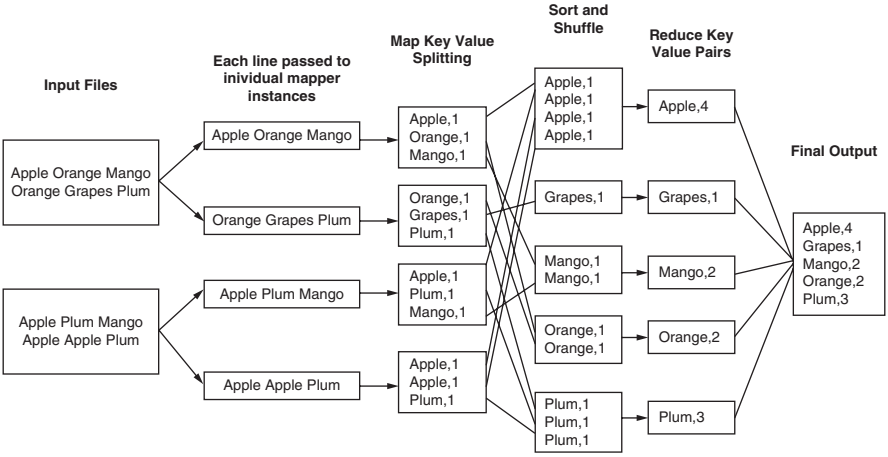


Fig. 11 Map-Reduce example (Source: kickstarthadoop [28])

The input to Map step is set of ($Key_{map\text{-}input}$, Value). The output of map step is another set of ($key_{map\text{-}output}$, value). The shuffle and sort step sorts the output of map based on keys, group them together and send it to the reducer. So, the reducer input is another map of the form ($key_{map\text{-}output}$, {Value, Value, … , Value}), where each Key of the reducer is associated with a set of values coming out of the map step against that key. Note that the output keys of Map step are the same as the input keys of the reduce step.

Figure 11 provides an example of a map-reduce way of doing distributed computing to compute the word-count distribution in a group of files. The input to the map step is a set of files. Each file is split as a collection of lines. The collection of all the lines is a map, where the key is the location of the line and the value is the line. This map is the input of the Map step in this program. In most cases the map-reduce program does not

use input keys, it uses only the values in the input map. The map step involves initiating multiple programs (in different threads in the same node and multiple nodes), each of these programs is called mapper. The input to each mapper is one entry of the input collection, i.e. the map of lines. Thus the input of each mapper in Fig. 11 is a line. The mapper program splits the line into words and creates a map of (Word, 1), where 1 is the count of the word in that line. The mapper program sorts and shuffles this map. With the help of sorts and shuffle, all the mapper programs send the entries (i.e. the count) associated with the same key (i.e. same word) to the same reducer. The reducer upon receiving all the values associated with a key (word, {count, count,…, count}) sums up all the counts for that word and writes into HDFS. Each reducer writes its output independently as a separate file in the HDFS. This creates multiple output file of a map-reduce program running on top of HDFS.

4.2 Spark

In reality, for big data, a single map-reduce program cannot complete the computation required out of the data for analytic purposes. In most cases, a realistic data analytic computation requires a series of map-reduce programs. For example, computing the mean in a map-reduce form can be done in a single map-reduce program. However, the computation of the standard deviation in a map-reduce form will require two sequential map-reduce programs. The first map-reduce computation will compute the mean. Using the result of the first map-reduce, the second map-reduce computation will compute the standard deviation. The input to each map-reduce program is taken from HDFS or some other distributed persistent storage (such as No-SQL database). The output of each map-reduce program is also written into HDFS. This is depicted in Fig. 12.

However, the above workflow will be slow and time-consuming due to multiple reads and write from the HDFS system. Additionally, though map-reduce was developed to run large distributed parallel computing process on a number of regular consumer hardware, in present days computing machines with higher memory and processing capacity is very common in enterprise architecture. The spark has been developed to use the larger memory capacity of today's computing hardware.

Spark exploits the memory capacity to avoid the repeated reading and writing on the map-reduce workflow. Spark has a concept called "Resilient Distributed Data" (RDD). In the most simplistic concept, the RDD considers the memory across

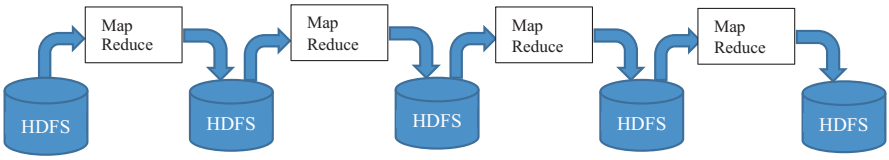


Fig. 12 Map-Reduce workflow

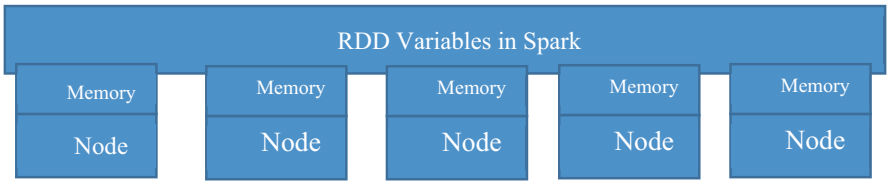
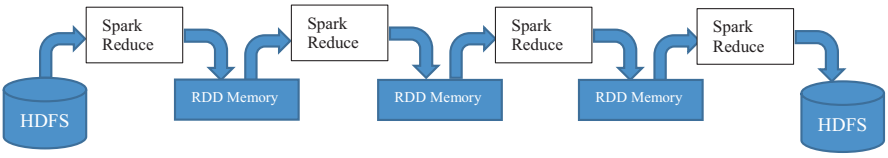


Fig. 13 RDD in spark

multiple computers as a single contiguous memory. Typical RDD variables are collections (such as map, array, list) that stay in memory but spans across computer boundary. This results in two advantages. First, as the collection is distributed across multiple machines, any processing of the collection can be done in parallel on all these machines, where each machine does the computation on its local memory (very similar to map-reduce computation). Second, the distributed map-reduce processing in the case of the spark is done on RDD (memory resident collections), so the processing is much faster than the HDFS based map-reduce (Fig. 13).

In the case of RDD in Spark, the map is a transformation that passes each item in the RDD through a function and returns a new RDD representing the result. For example, there is an RDD x a collection of 10 K integers. We want to increase each item in the RDD by 1. This can be carried out as an RDD map. The reduction on an RDD is an action that aggregates all the elements of the RDD use some function and returns the final result. The example of reducing on x will be sum all the values. Typically RDDs are kept in the memory and cease to exist once the spark program execution has finished. However, it is also possible to persist an RDD in memory, in which case the Spark will keep the elements around on the cluster for much faster access the next time we carry. There is also support for persisting RDDs on disk or replicated across multiple nodes.



5 Machine Learning Platforms

With the popularity of Spark, running machine learning algorithms on big data has become much easier. The landscape of machine learning on big data in changing dramatically with Spark. All latest machine learning platforms are using Spark in some way or other. Using these platforms companies can build models on large data sets without sampling and achieve accurate predictions. These tools use few optimizations to achieve so. First, they use more memory and processing power for

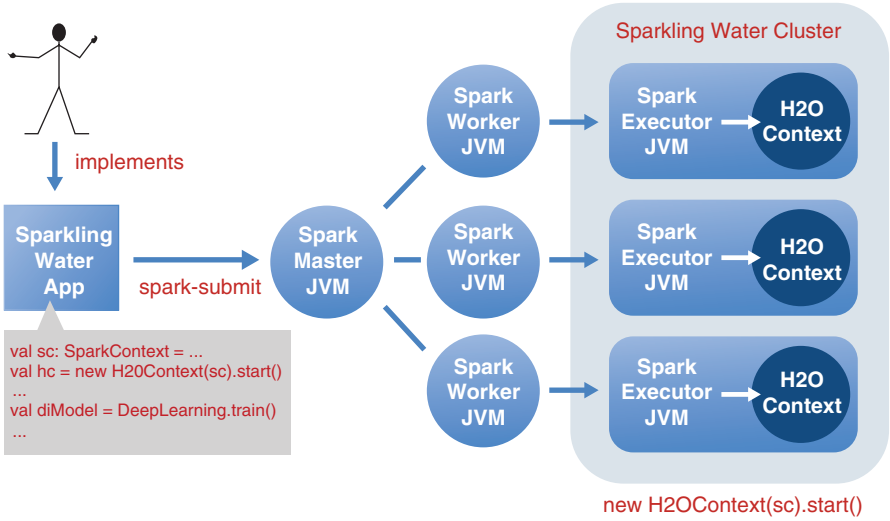


Fig. 14 Sparking water architecture (Courtesy: Cloudera [14])

making faster computations. Second, they use in-memory compression to handle large datasets. And third, they implement parallel distributed network training. The deep learning approaches used by these tools build hierarchies of hidden features that is composed to approximate complex functions with much less effort.

The Mahout [6] has been a very popular machine learning platform on HDFS platform. However, as Spark became popular many of the Mahout machine learning libraries migrated to Spark environment.

H2O [22] is another machine learning platform that can work on both Hadoop and Spark. SparkFlows [40] is a Big data application development platform for building and executing end-to-end data analytic products on Spark. It comes pre-packaged with an exhausting set of machine learning and ETL components making the workflow definition of big data use cases faster and easier.

The Sparkling Water project combines H2O machine-learning algorithms with the execution power of Apache Spark. Figure 14 illustrates the concept of technical realization. The application developer implements a Spark application using the Spark API and Sparkling Water library.

6 Search System

With the growth of data, the requirement of real-time delivery of information has grown also. This has particularly become true for textual data. A vast amount of big data is unstructured textual data, such as the posts derived from Twitter, Facebook, and blogs, or textual description of products, or archival data of legal documents.

ID	Text
1	Baseball is played during summer months.
2	Summer is the time for picnics here.
3	Months later we found out why.
4	Why is summer so hot here

↑ Sample document data

Dictionary and posting lists →

Term	Freq	Document ids
baseball	1	[1]
during	1	[1]
found	1	[3]
here	2	[2], [4]
hot	1	[4]
is	3	[1], [2], [4]
months	2	[1], [3]
summer	3	[1], [2], [4]
the	1	[2]
why	2	[3], [4]

Fig. 15 Inverted index example (Source: Hotcodeshare [25])

The storage consisting of such textual data can easily reach in the range of few hundreds of terabytes to a petabyte. The real-time search of this data is impossible to achieve with the traditional database indexing scheme.

To make the textual data searchable, an inverted index is created out of textual data. In forward index, a document is stored in the database, and with a document ID, we can retrieve the document. An inverted index an index is created by words in documents. Then each word in the index points to the set of the documents that contain that word. Figure 15 shows one example of the inverted index.

The first table in Fig. 15 is the data of documents along with the forward index; the second right-hand side table is the inverted index. With the second table, one can easily answer queries such as "Find all the documents containing the word 'summer.'" Without the inverted index, such query would have taken a long time by searching for the word `summer' in each and every document on the table. As the number of words in a language is limited, even with a very large number of documents the number of entries in the inverted index will be limited, and thus the makes it possible to hold the index in memory of a single node or a cluster of nodes. The basic algorithm of the inverted index was implemented as part of Lucene library [9].

6.1 Search Software

Solr [4] was developed on top of Lucene to have a server version that can support HTTP and XML based query. With big data, the requirement evolved to hold the index larger than single machine memory and have replication of the index to accommodate failover of a node hosting the index. This resulted in the development of ElasticSearch [17].

Elasticsearch is a distributed, RESTful search engine. It supports HTTP and JSON based query capability. Though the basics of ElasticSearch evolved to host the inverted index of textual data, the ElasticSearch can host index of any data. Say,

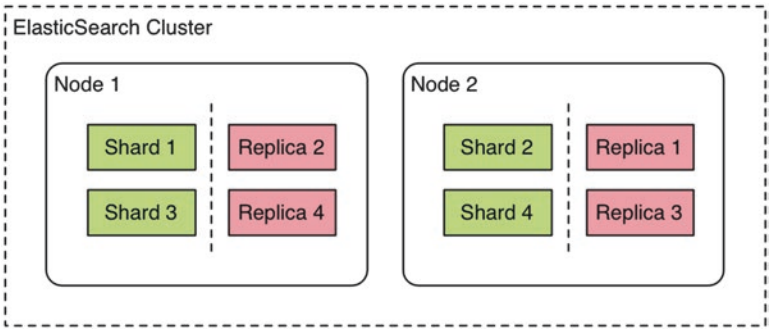


Fig. 16 Failover and clustering in ElasticSearch (Source: Liip [30])

for example; it can hold the index of product attributes such as manufacturer, model, price, year, rating, keywords in the product title and keywords in product descriptions. Typically ElasticSearch is not used to store the actual data; it is used to store memory-resident index structure that can search and queried in real-time. In big data architecture, it is very common practice to query the ElasticSearch to retrieve the document ID (such as product ID) and then to query the HDFS or No-SQL database to retrieve the actual document (or product details).

As shown in Fig. 16, the ElasticSearch has the inbuilt replica and sharding structure. The sharding allows the single index to be broken down into multiple partitions in different nodes. Each shard can be queried in parallel to retrieve data against a single query. This improves the query performance in ElasticSearch. Secondly, the sharding also allows the index larger than a single node memory to be stored and managed by ElasticSearch. The replica in ElasticSearch improves the reliability and failover mechanism in ElasticSearch making it a search platform of choice for online real-time applications.

Message Passing and Queuing System

In a big data system, nothing is a single node system – every component is a cluster of a large number of nodes that handle the distributed data and computation. In such a scenario, creating and managing one to one communication becomes a challenge. Consider a scenario where the data is coming from multiple sensors. A process is receiving the data from sensors and processing it to identify the structured data, and writing the semi-structure data into various data storage depending on the type of data and information (Fig. 17).

One of the critical problems with the above architecture is the flow of data out of sensor is non-uniform making it difficult to estimate the infrastructure requirement for processing the data. There will be a mismatch in the rate at which the data is coming out of sensors and the rate at which the data can be processed to write into the storage. This will result in having a large in-memory buffer in the data parsing and extraction program. Additionally, in case the data parsing and extraction program fail during processing, there will be a loss of data.

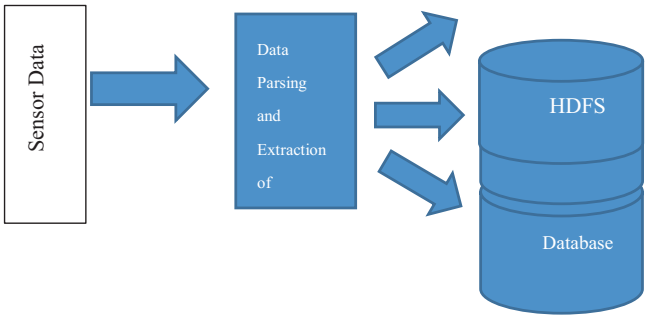


Fig. 17 Data processing & storing workflow

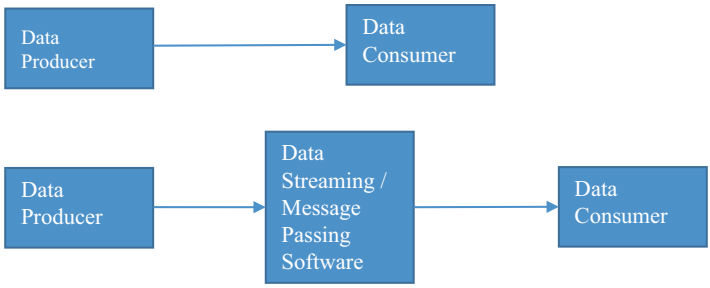


Fig. 18 Role of data streaming/message passing software

7 Big Data Messaging Software

To handle the above issues, a class of software has evolved – called message passing or stream processing software such as RabbitMQ, Kafka, Kinesis, Flink. These software components allow handling a large volume of messages. These software has the capability to hold the messages temporarily with failover and replication capability and can process the messages before passing it to the data consumer (Fig. 18).

RabbitMQ [36] is one of the leading message passing software that has been popular in IT infrastructure to manage streaming data since pre-big-data days. Traditionally RabbitMQ is a single server system, thought with the growing popularity of big data it has incorporated clustering in its architecture. The RabbitMQ has the capability to incorporate complex routing logic based on message content. The most popular message passing system in RabbitMQ is the pub-sub system. In the pub-sub system, a group of message producers publishes messages with subjects and a group of consumers consume these messages based on the subjects (Fig. 19).

Apache Kafka [8] is a clustered stream data processing software. Unlike RabbitMQ which can typically process messages in a range of 20–30 K per seconds, with the inbuilt clustering technology a Kafka cluster can process a much

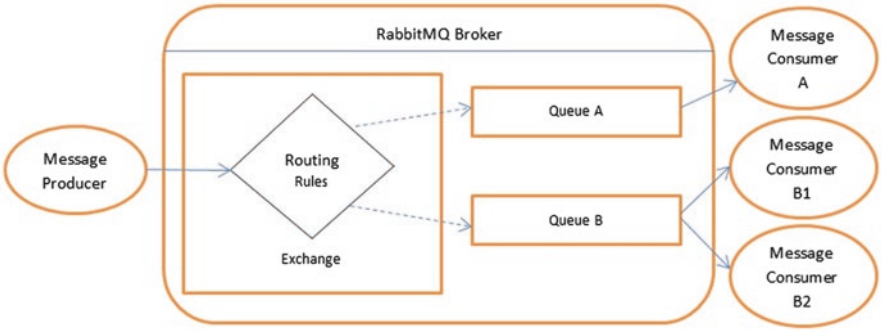


Fig. 19 Messaging with RabbitMQ (Source: https://keyholesoftware.com/2013/05/13/messaging-with-rabbitmq/)

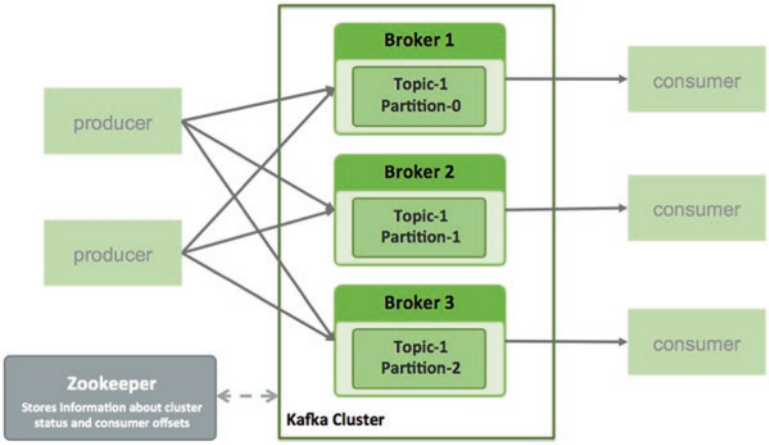


Fig. 20 Kafka cluster (Source: Hortonworks [24])

higher number of messages (100 K to few million messages per seconds). A Kafka cluster consists of multiple partitions and multiple servers. Each partition has one server which acts as the "leader" and zero or more servers which act as a "followers." The leaders handle all read and write requests for that partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others, so the load is balanced across multiple servers. Unlike RabbitMQ whose strength is in routing, the strength of Kafka can consume the massive volume of stream data (Fig. 20).

Apache Flink [7] is a streaming data processing system. It can handle large-scale system running thousands of nodes. It provides accurate computational results on streaming data. A very common use case for Apache Flink is analytics on stream data. Quite often Flink and Kafka are used together, where data streams for Flink are ingested from Kafka. Typically applications of Flink and Kafka start with event

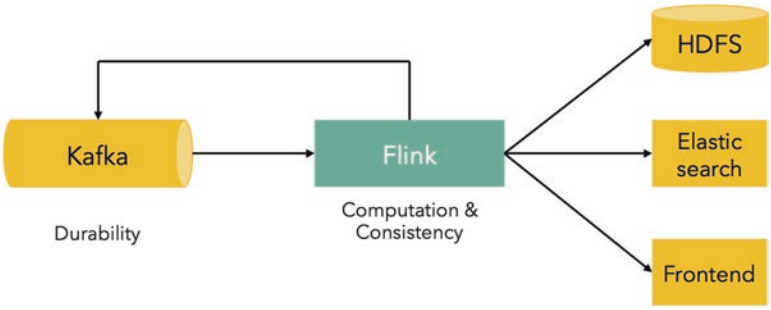


Fig. 21 Kafka and Flink together (Source: [42])

streams being pushed to Kafka, which are then consumed by Flink jobs. These jobs range from simple transformations of data import/export to more complex applications that aggregate data in windows. The results of these Flink jobs may be fed back to Kafka for consumption by other services or written out to other systems like HDFS, Elasticsearch, No-SQL database or web front end. In such a system, Kafka provides data durability, and Flink provides consistent data movement and computation (Fig. 21).

8 Cache

Caching is an important component of any big data-based systems that expect to provide a real-time response to requests. The generic idea of caching is most frequently accesses data items are brought near to the application so that frequent requests of these data items can be served in near real time. In the most application, there is exists a skew in access pattern to data. For example, following power law [1], 80% of the users will access 20% of the data item. These 20% data can be brought into a memory based caching system, from where the requests for the data can be server much faster than persistent storage such as database or file system.

8.1 Distributed Caching Systems

Memcached [31] is a very popular high-performance and distributed memory caching system. In essence, it is an in-memory key-value store for small chunks of data (strings, objects) from results of database calls, file-read or remote service call. Memcached is inbuilt in latest versions of MySQL to cache database calls. Traditionally Memcached has been a single server software component, but with the big data systems, Memcached has also grown to accommodate multi-node

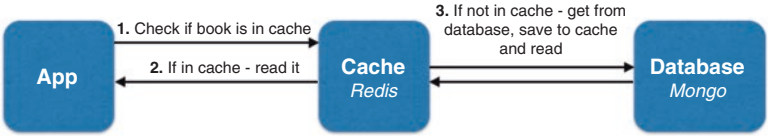


Fig. 22 REDIS cache in front of MongoDB No-SQL database (Source: Gino [21])

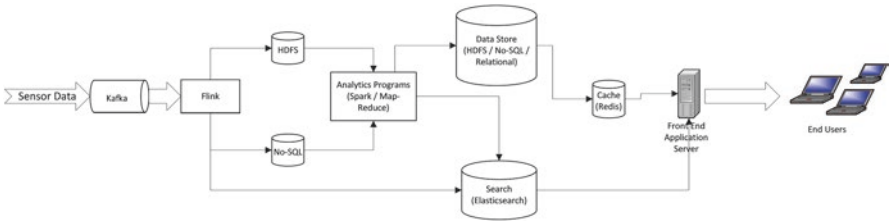


Fig. 23 Example architecture for big data analytics

cluster. Redis [37] is another software product that provides cache service. It is an in-memory data structure store and can store many complex objects such as arrays, sets, and lists. Unlike Memcached (which provides get and set operations only) Redis allows atomic operations on these objects such as appending to a string, pushing an element to a list, computing set union, intersection and difference or finding the item in a sorted set. Redis has built-in replication, high availability and data partition feature. Though Redis works with an in-memory dataset, it can persist the data by periodically dumping the data to disk. Redis can be used as both in-memory no-SQL Database and cache (Fig. 22).

Case Study: Big Data Analytics Example Architecture

Figure 23 presents an example architecture of a big data analytics in an organization. At the left-hand side, a massive stream of sensor data and social media data is coming as input. This data is passed to Kafka for temporary holding. The Flink consumes these messages from Kafka, processes and parses it, and writes to appropriate storage (HDFS or No-SQL database). The analytic programs run on top of the data that is stored in HDFS or No-SQL database. These analytics programs can be in the form of map-reduce, spark and use advanced machine learning libraries such as SparkML, SparkFlow, and TensorFlow. The output of analytics program is again saved into the database. Depending on the use case and the data volume, the analytic output can be stored in HDFS, No-SQL database or even in a relational database. The Elasticsearch or equivalent search system is populated with the output of Flink and analytics program for indexing purpose. End users use a front end application to search and retrieve information. The front application is running in the application server first contacts the Elasticsearch to search for the information (such as a list of product ID) based on various attribute values. Then it contacts the database to retrieve the actual information (such as product details) associated with the search results. The cache (such as Redis) may be deployed

in between the front end application (in application server) and the data storage to improve the speed of access to popular items (such as daily hot products).

9 Data Visualization

With the increase of data, the visualization of this volume and variety of data has become a challenge. Some tools have emerged in recent years to present the data in innovative ways. Tableau [41] has become a popular technology to do data visualization. There are some tools on visualization that works in the cloud and others that work as a desktop application with cloud-based access to reports. The traditional technologies like Tableau replied on the later. Tableau is primarily used to develop dashboard. Tableau is an end user-friendly tool. As soon as the data is connected with the Tableau, the Tableau GUI can be used to develop various GUI based reports and dashboards.

Recently a new technology "Notebook" has come up as a way to develop and maintain rich visualization of data. One of such software is Jupyter [27]. The notebook in Jupyter contains both computer code and rich text elements (paragraphs, equations, figures, links, etc.). These documents contain analysis description and the results along with the executable code which can be run to perform data analysis. This allows automatic generation of rich text document containing data analysis text and visual representation of the data. The executable code associated with a notebook can be shared and can be modified to develop new reports. In the past, the report generators and reports were two different component in the enterprises that have been maintained separately. This used to lead to a lot of mismatch in report generation code and the actual reports. The Notebook technology allows these two to be merged and considered as a single unit. IBM's Data Science Experience [26] is another such technology by IBM. Apache Zeppelin [10] is an Apache software that supports the Notebook functionality.

10 Conclusion

In this chapter we have discussed various technologies related to big data technology – NoSQL database, distributed file system, map-reduce and spark based distributed computation, distributed communication platform, distributed caching, search platform and visualization technologies. Our intention here was to give an overview of all these technologies so that appropriate technical discussion can be led in future by the readers. Later in the chapter, we delved down into some associated technologies such as search system, message processing and caching that makes the big data analytics application more robust and performance efficient. Lastly, we present an example architecture of a big data analytics based application

Table 4 Summary of big data technology

Technology type	Purpose	Product example	Use case
Database	Distribute data with replication and failover	No-SQL databases such as Cassandra, MongoDB, BerkleyDB, CouchDB, SimpleDB, DynamoDB	Store and manage large volume of structured and unstructured data
File system	File system distributed across multiple nodes	HDFS, GFS	Store and manage large files or a large number of small files
Programming	Distributed programming that can process data in parallel	Hadoop Map-Reduce, Spark	Computation and analytics job on top of data in distributed file system or no-SQL database
Machine learning platform	Complex analytical work using machine learning techniques	Mahout, H2O, SparkML, Sparking Water, SparkFlows	Deep learning and machine learning on big data
Search system	To search unstructured and semistructured data	ElasticSearch, Solr	Store and manage index on big data for search purposes.
Messaging system	To introduce an intermediate buffer between data collection and data storage	RabbitMQ, Kafka, Kinesis, Flink	Read high throughput incoming data (such as twitter data, sensor data) and preprocess it before writing into data storage
Caching	Distributed application level caching	REDIS, Memcached	Store frequently accessed data from in-memory distributed cache to reduce the access time to this data compared to accessing from persistent storage of database or filesystem.
Data visualization	Provides Notebook functionality where report generation code and the actual report co-exists	Tableau, Jupyter, Zeppelin, Data Science Experience	Generate complex report including visual representation and textual description of data

using these technologies. In Table 4 we summarize the technologies and products described in this chapter.

Though there is abundance of technology and software platforms to process, manage and use big data, the appropriate choice is very critical for the success of these platforms. We expect more technology to evolve in next few years to support distributed computation on big data. Once such technology is cryptocurrency (such as bitcoin) and blockchain. The blockchain technology is gradually getting traction to store the data in a peer to peer fashion without the control of any single entity. The technology is now being applied in wide variety of domains including financial,

healthcare and contract management. IOT (internet of things) is another distributed technology that is coming up. The application of IOT in every part of our life is becoming the norms, where the data gathering, communicating and processing are interconnected and distributed to the point where data is being generated.

References

1. Adamic, L. A., & Huberman, B. A. (2000). Power-Law Distribution of the World Wide Web. *Science*, *287*(5461).
2. Aerospike. (2017). Aerospike | High Performance NoSQL Database. Retrieved March 23, 2017, from http://www.aerospike.com/
3. Amazon. (2017). AWS | Amazon SimpleDB – Simple Database Service. Retrieved March 23, 2017, from https://aws.amazon.com/simpledb/
4. Apache. (2015). Solr. Retrieved from http://lucene.apache.org/solr/
5. Apache. (2017a). Apache CouchDB. Retrieved March 23, 2017, from http://couchdb.apache.org/
6. Apache. (2017b). Apache Mahout: Scalable machine learning and data mining. Retrieved March 23, 2017, from http://mahout.apache.org/
7. Apache Flink. (2017). Apache Flink: Introduction to Apache Flink®. Retrieved March 23, 2017, from https://flink.apache.org/introduction.html
8. Apache Kafka. (2017). Apache Kafka. Retrieved March 23, 2017, from https://kafka.apache.org/intro
9. Apache Lucene. (2017). Apache Lucene – Welcome to Apache Lucene. Retrieved March 23, 2017, from https://lucene.apache.org/
10. Apache Zeppelin. (2017). Zeppelin. Retrieved March 23, 2017, from https://zeppelin.apache.org/
11. Basho. (2017). Riak – Distributed Databases. Retrieved March 23, 2017, from http://basho.com/products/
12. Brewer, E., & Eric. (2010). A certain freedom. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing – PODC '10* (pp. 335–335). New York, New York, USA: ACM Press. http://doi.org/10.1145/1835698.1835701
13. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., … Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6–8, Seattle, WA, USA* (pp. 205–218). USENIX Association. Retrieved from http://research.google.com/archive/bigtable-osdi06.pdf
14. Cloudera. (2017). How-to: Build a Machine-Learning App Using Sparkling Water and Apache Spark – Cloudera Engineering Blog. Retrieved March 23, 2017, from http://blog.cloudera.com/blog/2015/10/how-to-build-a-machine-learning-app-using-sparkling-water-and-apache-spark/
15. Datastax. (2017). Introduction to Cassandra Query Language. Retrieved March 23, 2017, from https://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html
16. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., … Vogels, W. (2007). Dynamo. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles – SOSP '07* (p. 205). New York, New York, USA: ACM Press. http://doi.org/10.1145/1294261.1294281
17. Elastic. (2017). Open Source Search Analytics · Elasticsearch. Retrieved March 23, 2017, from https://www.elastic.co/
18. Erb, B. (2016). The Challenge of Distributed Database Systems. Retrieved March 23, 2017, from http://berb.github.io/diploma-thesis/community/061_challenge.html
19. Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. *ACM SIGOPS Operating Systems Review*.

20. Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, *33*(2), 51. http://doi.org/10.1145/564585.564601
21. Gino, I. (2017). Caching a MongoDB Database with Redis — SitePoint. Retrieved March 23, 2017, from https://www.sitepoint.com/caching-a-mongodb-database-with-redis/
22. H2O. (2017). H2O.ai. Retrieved March 23, 2017, from https://www.h2o.ai/h2o/
23. Hortonworks. (2017a). Apache Hadoop HDFS – Hortonworks. Retrieved March 23, 2017, from https://hortonworks.com/apache/hdfs/#section_2
24. Hortonworks. (2017b). Introduction to Kafka – Hortonworks Data Platform. Retrieved March 23, 2017, from https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.3.2/bk_kafka-user-guide/content/ch_using_kafka.html
25. Hotcodeshare. (2017). How Elasticsearch index document? | Hot code share. Retrieved March 23, 2017, from http://www.hotcodeshare.com/content/how-elasticsearch-index-document
26. IBM. (2017). IBM Data Science Experience. Retrieved March 23, 2017, from https://www.ibm.com/us-en/marketplace/data-science-experience/resources
27. Jupyter. (2017). Project Jupyter. Retrieved March 23, 2017, from http://jupyter.org/
28. kickstarthadoop. (2017). Kick Start Hadoop: Word Count – Hadoop Map Reduce Example. Retrieved March 23, 2017, from http://kickstarthadoop.blogspot.com/2011/04/word-count-hadoop-map-reduce-example.html
29. Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *Computer*, *43*(2), 12–14. http://doi.org/10.1109/MC.2010.58
30. Liip. (2017). On ElasticSearch performance – Liip Blog. Retrieved March 23, 2017, from https://blog.liip.ch/archive/2013/07/19/on-elasticsearch-performance.html
31. Memcached. (2017). memcached – a distributed memory object caching system. Retrieved March 23, 2017, from https://memcached.org/
32. MongoDB Inc. (2015). mongoDB. Retrieved from https://www.mongodb.org/
33. Oracle. (2017). Berkeley DB Products. Retrieved March 23, 2017, from https://www.oracle.com/database/berkeley-db/index.html
34. Ozsu, M. T., & Valduriez, P. (2011). *Principles of Distributed Database Systems – M. Tamer Özsu, Patrick Valduriez – Google Books*. Retrieved from https://books.google.com/books?hl=en&lr=&id=TOBaLQMuNV4C&oi=fnd&pg=PR3&dq=Distributed+Database&ots=LqFjgM_P-7&sig=mcmEnxerBLtixHY-0CrzS2hFojc#v=onepage&q=Distributed Database&f=false
35. Pritchett, D., & Dan. (2008). BASE: AN ACID ALTERNATIVE. *Queue*, *6*(3), 48–55. http://doi.org/10.1145/1394127.1394128
36. RabbitMQ. (2017). RabbitMQ – Messaging that just works. Retrieved March 23, 2017, from https://www.rabbitmq.com/
37. Redis. (2017). Redis. Retrieved March 23, 2017, from https://redis.io/
38. Saphanatutorial. (2017). How YARN Overcomes MapReduce Limitations in Hadoop 2.0. Retrieved March 23, 2017, from http://saphanatutorial.com/how-yarn-overcomes-mapreduce-limitations-in-hadoop-2-0/
39. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (pp. 1–10). IEEE. http://doi.org/10.1109/MSST.2010.5496972
40. Sparkflows. (2017). SparkFlows.io | Big Data Application Development Made Easy. Retrieved March 23, 2017, from https://www.sparkflows.io/overview
41. Tableau. (2017). Business Intelligence and Analytics – Tableau Software. Retrieved March 23, 2017, from https://www.tableau.com/
42. Tzoumas, K., & Metzger, R. (2015). Kafka + Flink: A practical, how-to guide – data Artisans. Retrieved March 23, 2017, from https://data-artisans.com/kafka-flink-a-practical-how-to/