UNIT V

FRAMEWORKS AND VISUALIZATION

MapReduce

MapReduce is a functional programming paradigm that is well suited to handling parallel processing of huge data sets distributed across a large number of computers.

MapReduce is the application paradigm supported by Hadoop and the infrastructure presented in this article.

MapReduce works in two steps:

1. **Map**: The map step essentially solves a small problem: Hadoop's partitioner divides the problem into small workable subsets and assigns those to map processes to solve.

2. **Reduce**: The reducer combines the results of the mapping processes and forms the output of the MapReduce operation.

Maps specific keys to specific values.

For example, if we were to count the number of times each word appears in a book, our MapReduce application would output each word as a key and the value as the number of times it is seen.

Or more specifically, the book would probably be broken up into sentences or paragraphs, and the Map step would return each word mapped either to the number of times it appears in the sentence (or to "1" for each occurrence of every word) and then the reducer would combine the keys by adding their values together.

Prior to submitting your job to Hadoop, you would first load your data into Hadoop. It would then distribute your data, in blocks, to the various slave nodes in its cluster. Then when you did submit your job to Hadoop, it would distribute your code to the slave nodes and have each map and reduce task process data on that slave node. Your map task would iterate over every word in the data block passed to it (assuming a sentence in this example), and output the word as the key and the value as “1”.

The reduce task would then receive all instances of values mapped to a particular key; for example, it may have 1,000 values of “1” mapped to the work “apple”, which would mean that there are 1,000 apples in the text. The reduce task sums up all of the values and outputs that as its result.

Then your Hadoop job would be set up to handle all of the output from the various reduce tasks.

An example of MapReduce

1. Let’s look at a simple example. Assume you have five files, and each file contains two columns (a key and a value in Hadoop terms) that represent a city and the corresponding temperature recorded in that city for the various measurement days.In this example, city is the key and temperature is the value.

Toronto, 20

Whitby, 25

New York, 22

Rome, 32

Toronto, 4

Rome, 33

New York, 18

Out of all the data we have collected, we want to find the maximum temperature for each city across all of the data files (note that each file might have the same city represented multiple times). Using the MapReduce framework, we can break this down into five map tasks, where each mapper works on one of the five files and the mapper task goes through the data and returns the maximum temperature for each city. For example, the results produced from one mapper task for the data above would look like this:

(Toronto, 20) (Whitby, 25) (New York, 22) (Rome, 33)

Let's assume the other four mapper tasks (working on the other four files not shown here) produced the following intermediate results:

(Toronto, 18) (Whitby, 27) (New York, 32) (Rome, 37)(Toronto, 32) (Whitby, 20) (New York, 33) (Rome, 38)(Toronto, 22) (Whitby, 19) (New York, 20) (Rome, 31)(Toronto, 31) (Whitby, 22) (New York, 19) (Rome, 30)

All five of these output streams would be fed into the reduce tasks, which combine the input results and output a single value for each city, producing a final result set as follows: (Toronto, 32) (Whitby, 27) (New York, 33) (Rome, 38)

An example of MapReduce:

The typical introductory program or 'Hello World' for Hadoop is a word count program. Word count programs or functions do a few things: 1) look at a file with words in it, 2) determine what words are contained in the file, and 3) count how many times each word shows up and potentially rank or sort the results. For example, you could run a word count function on a 200 page book about software programming to see how many times the word "code" showed up and what other words were more or less common. A word count program like this is considered to be a simple program.

The word counting problem becomes more complex when we want it to run a word count function on 100,000 books, 100 million web pages, or many terabytes of data instead of a single file. For this volume of data, we need a framework like MapReduce to help us by applying the principle of divide and conquer—MapReduce basically takes each chapter of each book, gives it to a different machine to count, and then aggregates the results on another set of machines. The MapReduce workflow for such a word count function would follow the steps as shown in the diagram below:

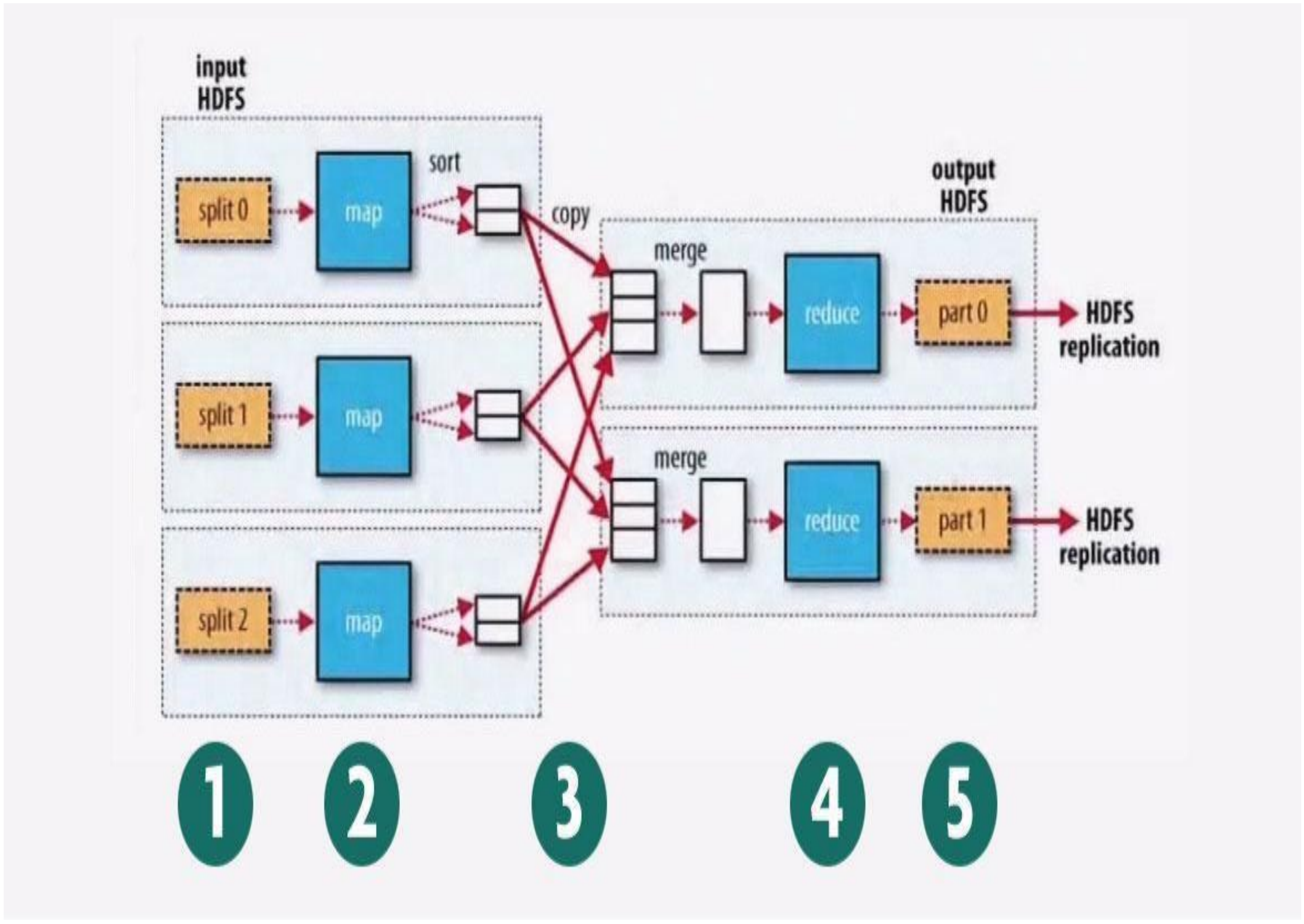
1. The system takes input from a file system and splits it up across separate Map nodes

2. The Map function or code is run and generates an output for each Map node—in the word count function, every word is listed and grouped by word per node

3. This output represents a set of intermediate key-value pairs that are moved to Reduce nodes as input

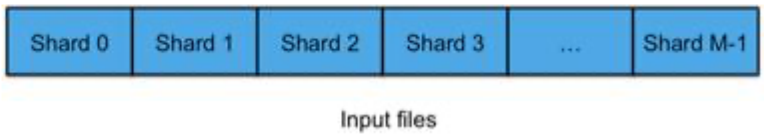
4. The Reduce function or code is run and generates an output for each Reduce node—in the word count example, the reduce function sums the number of times a group of words or keys occurs

5. The system takes the outputs from each node to aggregate a final view .



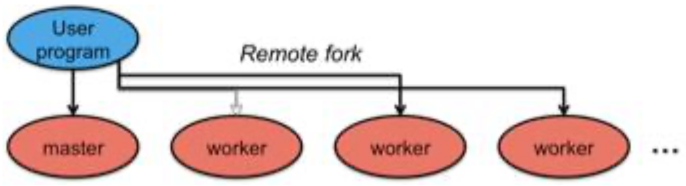
MapReduce is implemented in a master/worker configuration, with one master serving as the coordinator of many workers. A worker may be assigned a role of either a *map worker* or a *reduce worker*.

Step 1. Split input



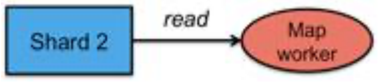
The first step, and the key to massive parallelization in the next step, is to split the input into multiple pieces. Each piece is called a split, or shard. For *M* map workers, we want to have *M* shards, so that each worker will have something to work on. The number of workers is mostly a function of the amount of machines we have at our disposal.

The MapReduce library of the user program performs this split. The actual form of the split may be specific to the location and form of the data. MapReduce allows the use of custom readers to split a collection of inputs into shards, based on specific format of the files. **Step 2. Fork processes**



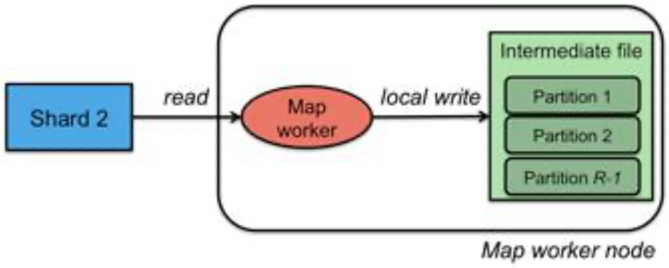
The next step is to create the master and the workers. The master is responsible for dispatching jobs to workers, keeping track of progress, and returning results. The master picks idle workers and assigns them either a map task or a reduce task. A map task works on a single shard of the original data. A reduce task works on intermediate data generated by the map tasks. In all, there will be *M* map tasks and *R* reduce tasks. The number of reduce tasks is the number of partitions defined by the user. A worker is sent a message by the master identifying the program (map or reduce) it has to load and the data it has to read.

Step 3. Map



Each *map* task reads from the input shard that is assigned to it. It parses the data and generates *(key,* *value)* pairs for data of interest. In parsing the input, the *map* function is likely to get rid of a lot of data that is of no interest. By having many map workers do this in parallel, we can linearly scale the performance of the task of extracting data.

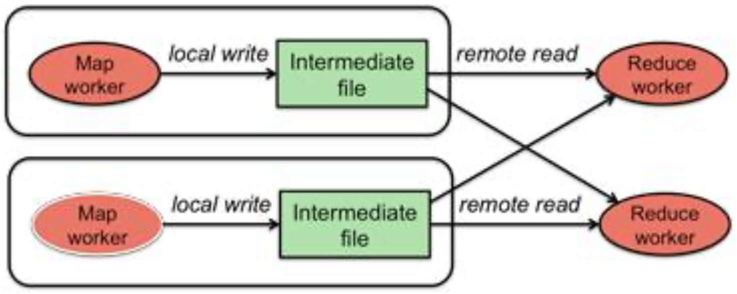
Step 4: Map worker: Partition



The stream of *(key, value)* pairs that each worker generates is buffered in memory and periodically stored on the local disk of the map worker. This data is partitioned into *R* regions by a partitioning function.

The partitioning function is responsible for deciding which of the *R* reduce workers will work on a specific key. The default partitioning function is simply a hash of *key* modulo *R* but a user can replace this with a custom partition function if there is a need to have certain keys processed by a specific reduce worker.

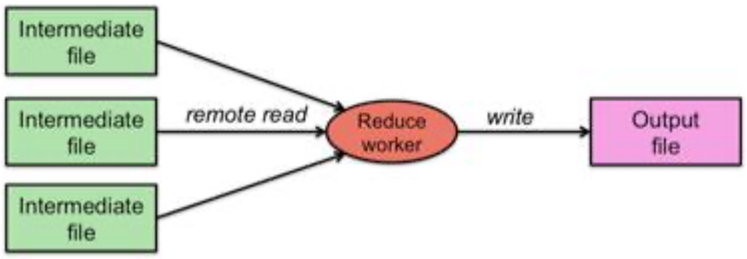
Step 5: Reduce: Sort (Shuffle)



When all the map workers have completed their work, the master notifies the reduce workers to start working. The first thing a reduce worker needs to is to get the data that it needs to present to the user's *reduce* function. The reduce worker contacts every map worker via remote procedure calls to get the *(key, value)* data that was targeted for its partition. This data is then sorted by the keys. Sorting is needed since it will usually be the case that there are many occurrences of the same key and many keys will map to the same reduce worker (same partition). After sorting, all occurrences of the same key are grouped together so that it is easy to grab all the data that is associated with a single key.

This phase is sometimes called the shuffle phase.

Step 6: Reduce function



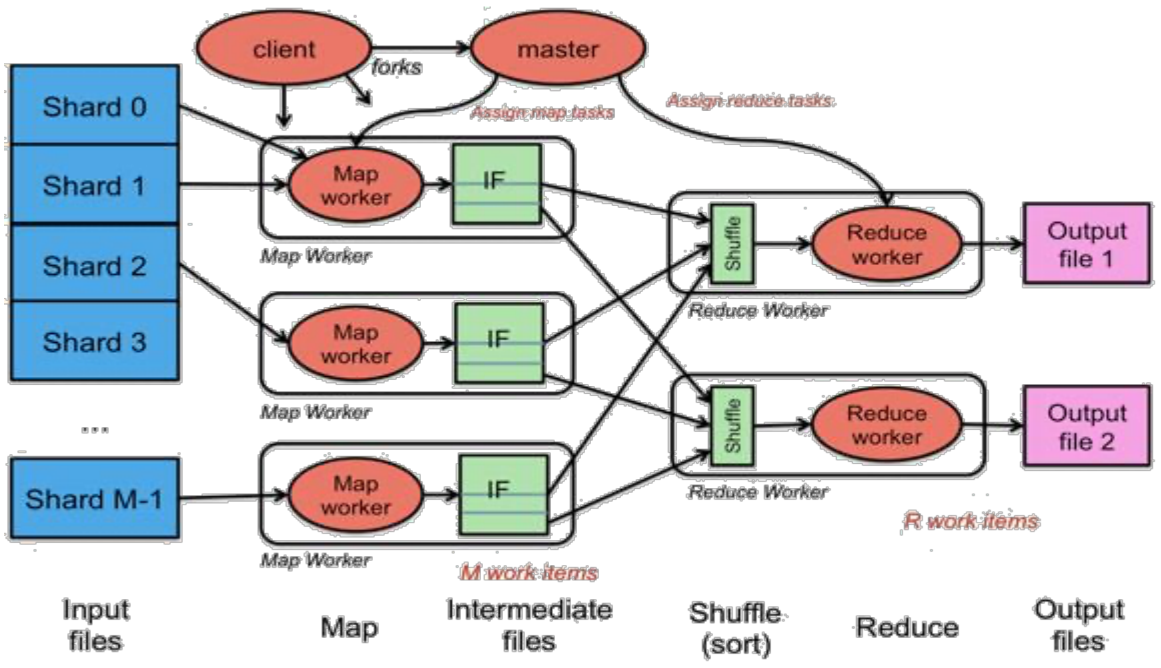
With data sorted by keys, the user's *Reduce* function can now be called. The reduce worker calls the *Reduce* function once for each unique key. The function is passed two parameters: the key and the list of intermediate values that are associated with the key.

The *Reduce* function writes output sent to file.

Step 7: Done!

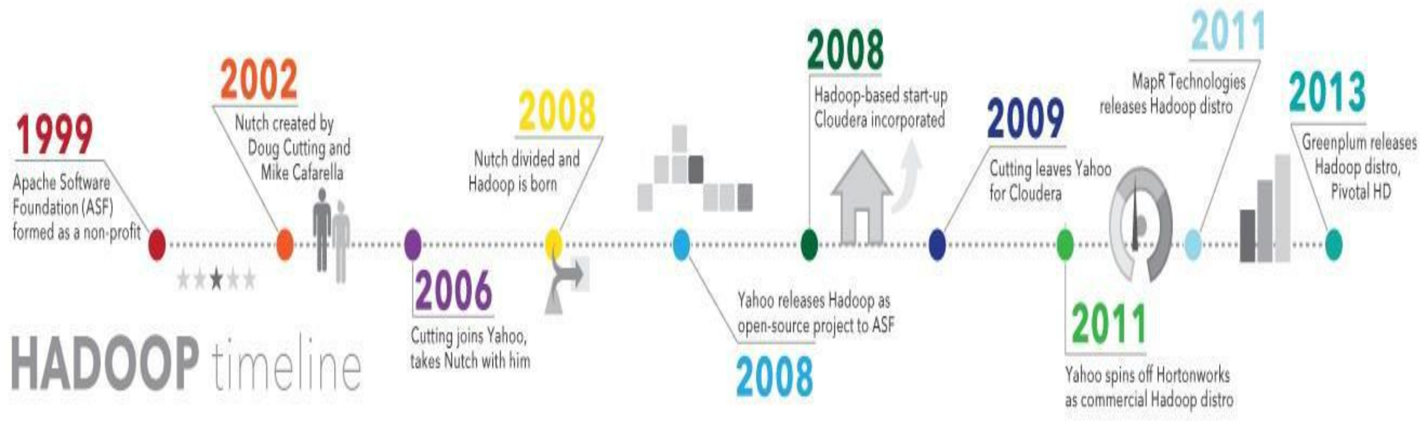
When all the reduce workers have completed execution, the master passes control back to the user program.

The client library initializes the shards and creates map workers, reduce workers, and a master. Map workers are assigned a shard to process. If there are more shards than map workers, a map worker will be assigned another shard when it is done. Map workers invoke the user's*Map* function to parse the data and write intermediate *(key, value)* results onto their local disks. This intermediate data is partitioned into *R* partitions according to a partioning function. Each of *R* reduce workers contacts all of the map workers and gets the set of *(key, value)* intermediate data that was targeted to its partition. It then calls the user's *Reduce* function once for each unique key and gives it a list of all values that were generated for that key. The *Reduce* function writes its final output to a file that the user's program can access once MapReduce has completed.



Hadoop

As the World Wide Web grew in the late 1900s and early 2000s, search engines and indexes were created to help locate relevant information amid the text-based content. In the early years, search results were returned by humans. But as the web grew from dozens to millions of pages, automation was needed. Web crawlers were created, many as university-led research projects, and search engine start-ups took off (Yahoo, AltaVista, etc.).


1999
Apache Software Foundation (ASF) formed as a non-profit
2002
Nutch created by Doug Cutting and Mike Cafarella
2006
Cutting joins Yahoo, takes Nutch with him
2008
Nutch divided and Hadoop is born
2008
Yahoo releases Hadoop as open-source project to ASF
2008
Hadoop-based start-up Cloudera incorporated
2009
Cutting leaves Yahoo for Cloudera
2011
Yahoo spins off Hortonworks as commercial Hadoop distro
2011
MapR Technologies releases Hadoop distro
2013
Greenplum releases Hadoop distro, Pivotal HD
HADOOP timeline

One such project was an open-source web search engine called Nutch – the brainchild of Doug Cutting and Mike Cafarella. They wanted to return web search results faster by distributing data and calculations across different computers so multiple tasks could be accomplished simultaneously. During this time, another search engine project called Google was in progress. It was based on the same concept – storing and processing data in a distributed, automated way so that relevant web search results could be returned faster.

In 2006, Cutting joined Yahoo and took with him the Nutch project as well as ideas based on Google's early work with automating distributed data storage and processing. The Nutch project was divided – the web crawler portion remained as Nutch and the distributed computing and processing portion became Hadoop

(named after Cutting's son's toy elephant). In 2008, Yahoo released Hadoop as an open-source project. Today, Hadoop’s framework and ecosystem of technologies are managed and maintained by the non-profit Apache Software Foundation (ASF), a global community of software developers and contributors.

Hadoop

In the evolution of data processing, we moved from flat files to relational databases and from relational databases to NoSQL databases. Essentially, as the amount of captured data increased, so did our needs, and traditional patterns no longer sufficed. The databases of old worked well with data that measured in megabytes and gigabytes, but now that companies realize “data is king,” the amount of captured data is measured in terabytes and petabytes. Even with NoSQL data stores, the question remains: How do we analyze that amount of data?

Hadoop is an open-source framework for developing and executing distributed applications that process very large amounts of data. Hadoop is meant to run on large clusters of commodity machines, which can be machines in your data center that you’re not using or even Amazon EC2 images. The danger, of course, in running on commodity machines is how to handle failure. Hadoop is architected with the assumption that hardware will fail and as such, it can gracefully handle most failures. Furthermore, its architecture allows it to scale nearly linearly, so as processing capacity demands increase, the only constraint is the amount of budget you have to add more machines to your cluster.

Hadoop Architecture

At a high-level, Hadoop operates on the philosophy of pushing analysis code close to the data it is intended to analyze rather than requiring code to read data across a network. As such, Hadoop provides its own file system, aptly named *Hadoop File System* or *HDFS*. When you upload your data to the HDFS, Hadoop will partition your data across the cluster (keeping multiple copies of it in case your hardware fails), and then it can deploy your code to the machine that contains the data upon which it is intended to operate.

Like many NoSQL databases, HDFS organizes data by keys and values rather than relationally. In other words, each piece of data has a unique key and a value associated with that key. Relationships between keys, if they exist, are defined in the application, not by HDFS. And in practice, you’re going to have to think about your problem domain a bit differently in order realize the full power of Hadoop (see the next section on MapReduce).

The components that comprise Hadoop are:

1. **HDFS**: The Hadoop file system is a distributed file system designed to hold huge amounts of data across multiple nodes in a cluster (where huge can be defined as files that are 100+ terabytes in size!) Hadoop provides both an API and a command-line interface to interacting with HDFS.

2. **MapReduce Application**: The next section reviews the details of MapReduce, but in short, MapReduce is a functional programming paradigm for analyzing a single record in your HDFS. It then assembles the results into a consumable solution. The Mapper is responsible for the data processing step, while the Reducer receives the output from the Mappers and sorts the data that applies to the same key.

3. **Partitioner**: The partitioner is responsible for dividing a particular analysis problem into workable chunks of data for use by the various Mappers. The Hash Partioner is a partitioner that divides work up by "rows" of data in the HDFS, but you are free to create your own custom partitioner if you need to divide your data up differently.

4. **Combiner**: If, for some reason, you want to perform a local reduce that combines data before sending it back to Hadoop, then you'll need to create a combiner. A combiner performs the reduce step, which groups values together with their keys, but on a single node before returning the key/value pairs to Hadoop for proper reduction.

5. **InputFormat**: Most of the time the default readers will work fine, but if your data is not formatted in a standard way, such as "key, value" or "key *tab+ value", then you will need to create a custom InputFormat implementation.

6. **OutputFormat**: Your MapReduce applications will read data in some InputFormat and then write data out through an OutputFormat. Standard formats, such as "key *tab+ value", are supported out of the box, but if you want to do something else, then you need to create your own OutputFormat implementation.

7. Additionally, Hadoop applications are deployed to an infrastructure that supports its high level of scalability and resilience. These components include:

8. **NameNode**: The NameNode is the master of the HDFS that controls slave DataNode daemons; it understands where all of your data is stored, how the data is broken into blocks, what nodes those blocks are deployed to, and the overall health of the distributed filesystem. In short, it is the most important node in the entire Hadoop cluster. Each cluster has one NameNode, and the NameNode is a single-point of failure in a Hadoop cluster.
9. **Secondary NameNode**: The Secondary NameNode monitors the state of the HDFS cluster and takes "snapshots" of the data contained in the NameNode. If the NameNode fails, then the Secondary NameNode can be used in place of the NameNode. This does require human intervention, however, so there is no automatic failover from the NameNode to the Secondary NameNode, but having the Secondary NameNode will help ensure that data loss is minimal. Like the NameNode, each cluster has a single Secondary NameNode.

10. **DataNode**: Each slave node in your Hadoop cluster will host a DataNode. The DataNode is responsible for performing data management: It reads its data blocks from the HDFS, manages the data on each physical node, and reports back to the NameNode with data management status.

11. **JobTracker**: The JobTracker daemon is your liaison between your application and Hadoop itself. There is one JobTracker configured per Hadoop cluster and, when you submit your code to be executed on the Hadoop cluster, it is the JobTracker's responsibility to build an execution plan. This execution plan includes determining the nodes that contain data to operate on, arranging nodes to correspond with data, monitoring running tasks, and relaunching tasks if they fail.

12. **TaskTracker**: Similar to how data storage follows the master/slave architecture, code execution also follows the master/slave architecture. Each slave node will have a TaskTracker daemon that is responsible for executing the tasks sent to it by the JobTracker and communicating the status of the job (and a heartbeat) with the JobTracker.

.

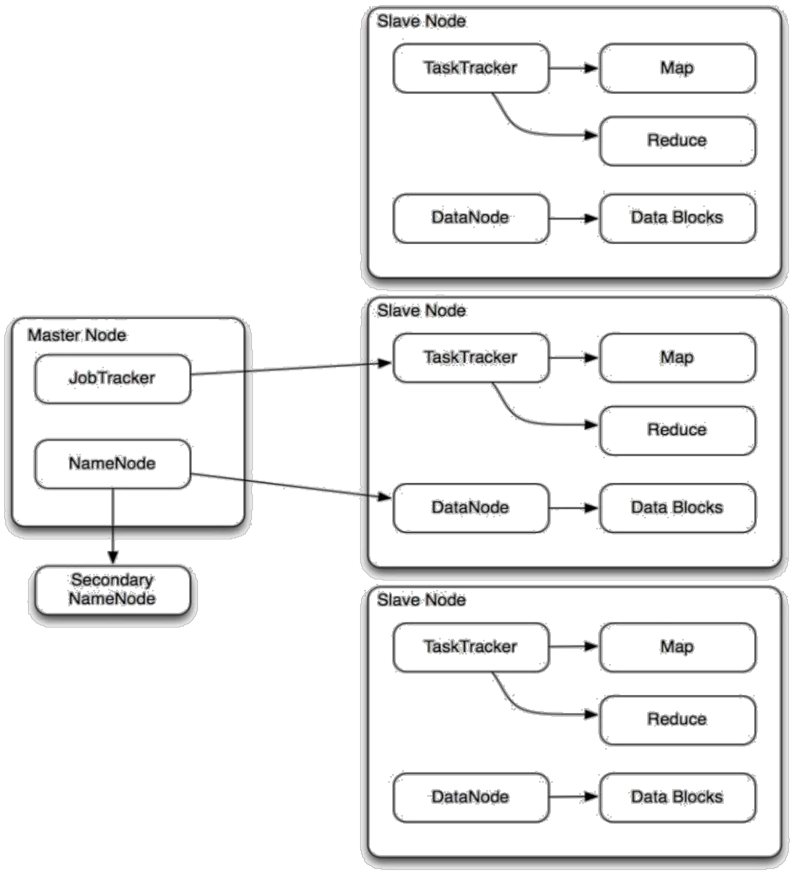


Figure 1 Hadoop application and infrastructure interactions

The master node contains two important components: the **NameNode**, which manages the cluster and is in charge of all data, and the **JobTracker**, which manages the code to be executed and all of the TaskTracker daemons. Each slave node has both a TaskTracker daemon as well as a DataNode: the TaskTracker receives its instructions from the JobTracker and executes map and reduce processes, while the DataNode receives its data from the NameNode and manages the data contained on the slave node. And of course there is a Secondary NameNode listening to updates from the NameNode.

What is Apache Pig?

Pig is a high-level programming language useful for analyzing large data sets. Pig was a result of development effort at Yahoo!

In a MapReduce framework, programs need to be translated into a series of Map and Reduce stages. However, this is not a programming model which data analysts are familiar with. So, in order to bridge this gap, an abstraction called Pig was built on top of Hadoop.

Apache Pig enables people to focus more on **analyzing bulk data sets and to spend less time writing Map-Reduce programs.** Similar to Pigs, who eat anything, the Apache Pig programming language is designed to work upon any kind of data. That's why the name, Pig!

FEATURES OF APACHE PIG IN BIG DATA

Apache Pig accompanies the following highlights:

1. User-defined Functions: Pig in big data gives the ability to make UDFs in other programming languages like Java and embed or invoke them in Pig Scripts.

2. Handles a wide range of data: Apache Pig examines a wide range of data, both unstructured as well as structured. It stores the outcomes in the Hadoop Distributed File System.

3. Rich set of operators: It gives numerous operators to perform tasks like a filter, sort, join, and so on.

4. Extensibility: Using the current operators, clients can build up their capacities to write, process, and read data.

5. The simplicity of programming: Pig Latin is like Structured Query Language and it is not difficult to compose a Pig scripting on the off chance that you are acceptable at Structured Query Language.

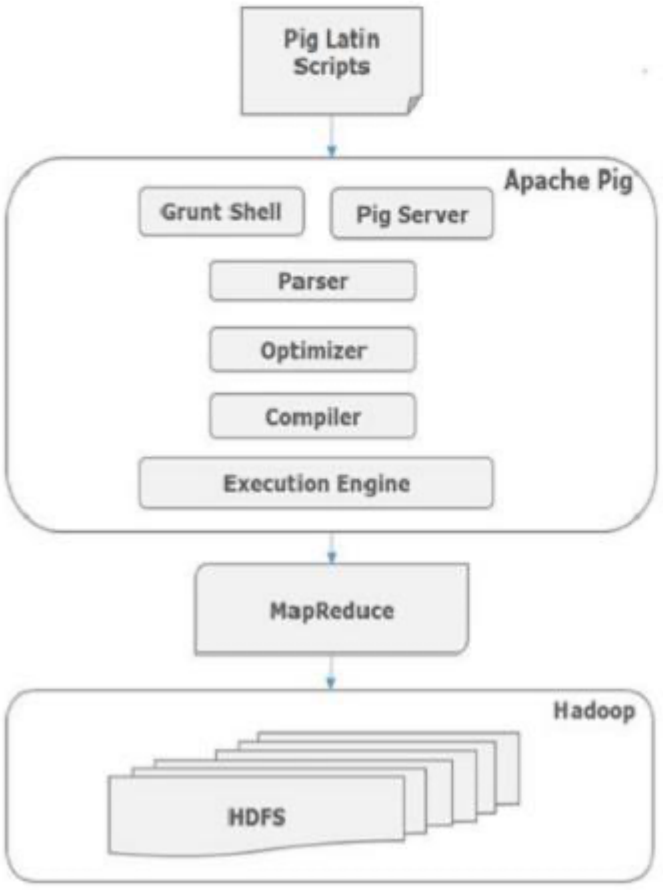
6. Optimization opportunities: The assignments in Apache Pig enhance their execution naturally, so the software engineers need to focus just on the semantics of the language.

Apache Pig - Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.



Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG (directed acyclic graph), which represents the Pig Latin statements and logical operators.

In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.

Optimizer

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

Compiler

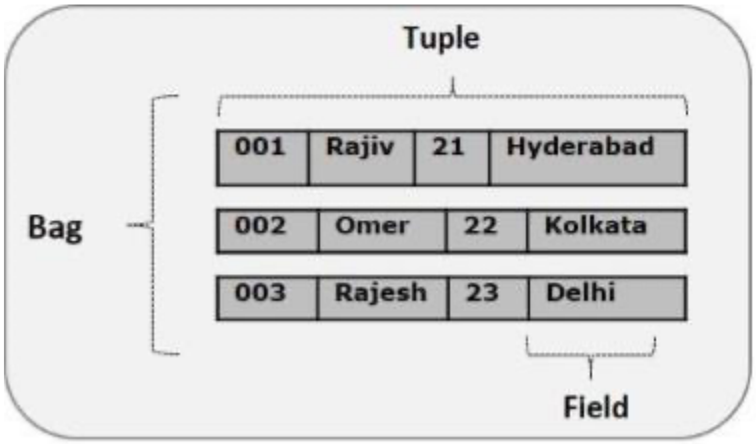
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

Pig Latin Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.



Atom

Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

Example − 'raja' or '30'

Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

Example − (Raja, 30)

Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by '{}'. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Example − {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner**

bag. **Example** − {Raja, 30, **{9848022338, raja@gmail.com,}**} **Map**

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by '[]'

Example − [name#Raja, age#30]

Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

Apache Pig	MapReduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.
There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

Apache Pig Vs Hive

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

Apache Pig	Hive
Apache Pig uses a language called **Pig Latin**. It was originally created at **Yahoo**.	Hive uses a language called **HiveQL**. It was originally created at **Facebook**.
Pig Latin is a data flow language.	HiveQL is a query processing language.
Pig Latin is a procedural language and it fits in pipeline paradigm.	HiveQL is a declarative language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

HIVE

Pig, which is a scripting language with a focus on dataflows. Hive provides a database query interface to Apache Hadoop

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

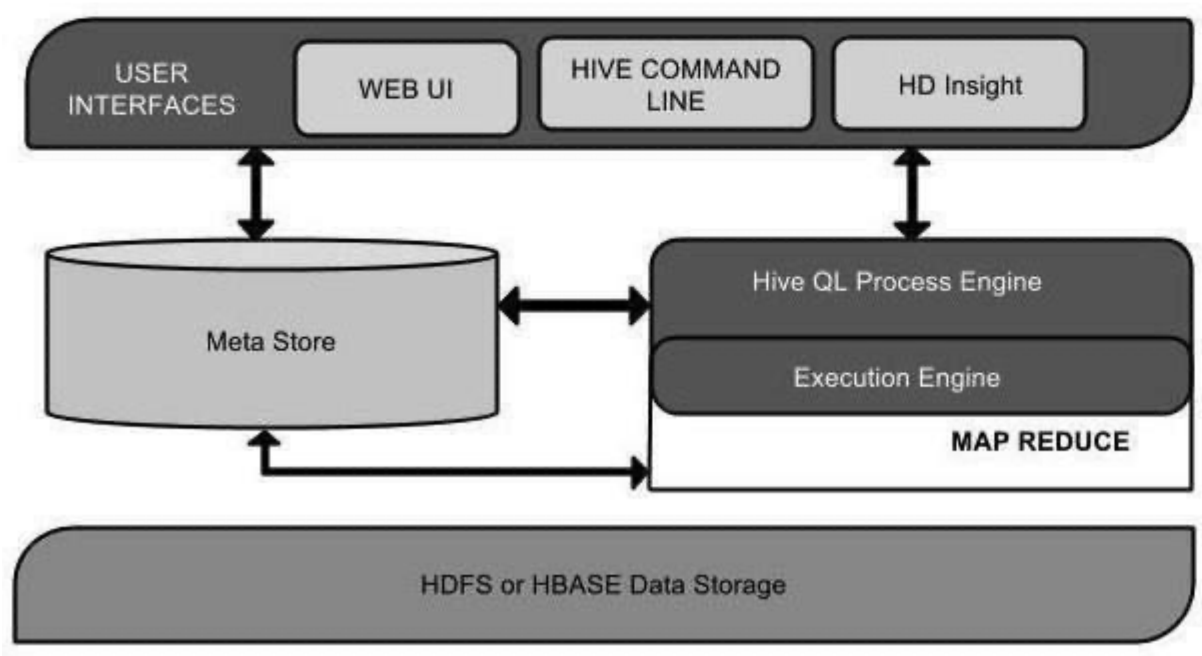
- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:

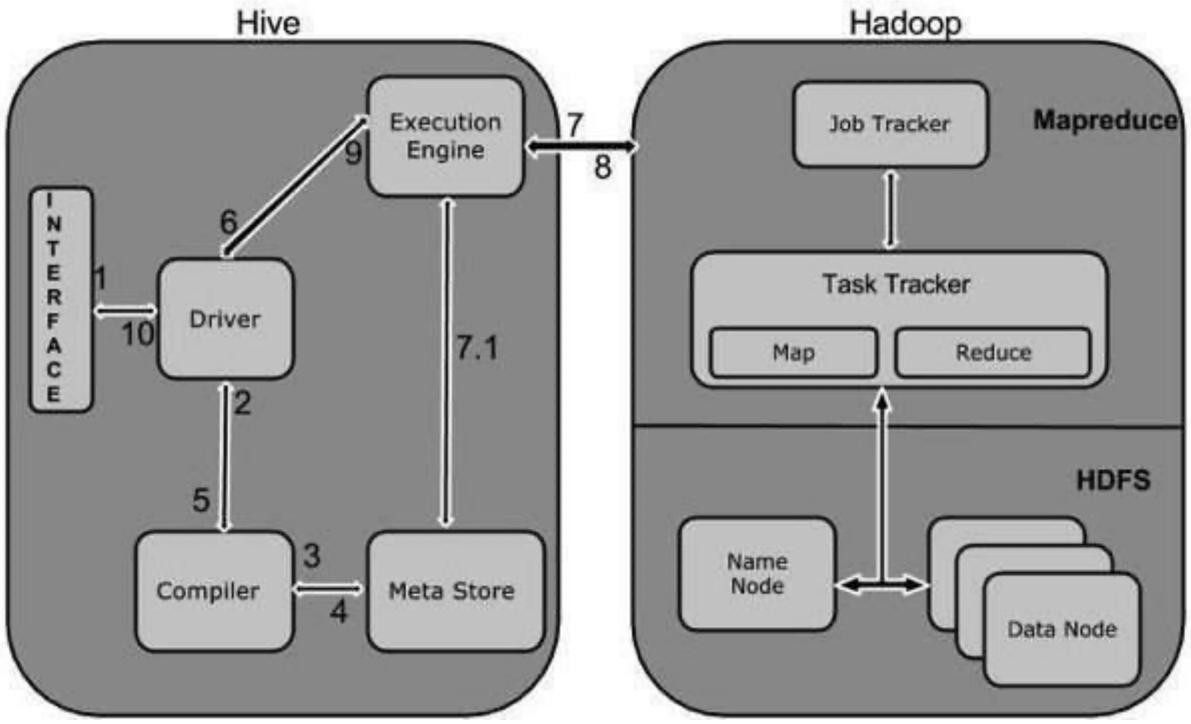


This component diagram contains different units. The following table describes each unit:

Unit Name	**Operation**
User Interface	Hive is data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.

2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata**
	The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.

7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops** Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

Hbase

Since 1970, RDBMS is the solution for data storage and maintenance related problems. After the advent of big data, companies realized the benefit of processing big data and started opting for solutions like Hadoop.

Hadoop uses distributed file system for storing big data, and MapReduce to process it. Hadoop excels in storing and processing of huge data of various formats such as arbitrary, semi-, or even unstructured.

Limitations of Hadoop

Hadoop can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs.

A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

Hadoop Random Access Databases

Applications such as HBase, Cassandra, couchDB, Dynamo, and MongoDB are some of the databases that store huge amounts of data and access the data in a random manner.

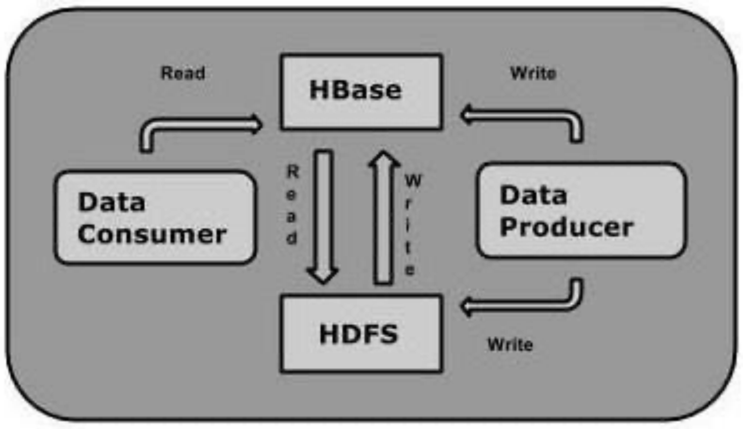
What is HBase?

HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.

HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).

It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.

One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.



HBase and HDFS

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.

Storage Mechanism in HBase

HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:

- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Given below is an example schema of table in HBase.

Rowid	**Column Family**			**Column Family**			**Column Family**			**Column Family**		
	col1	**col2**	**col3**	**col1**	**col2**	**col3**	**col1**	**col2**	**col3**	**col1**	**col2**	**col3**

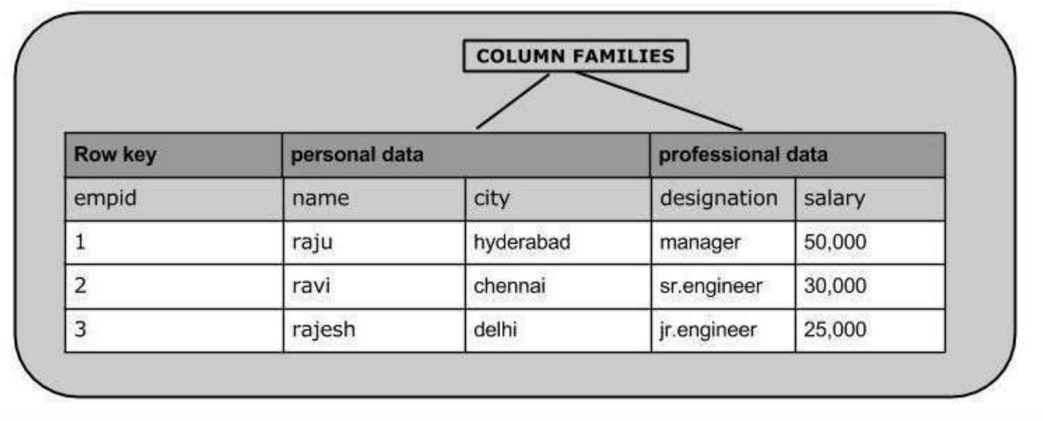
1												
2												
3												

Column Oriented and Row Oriented

Column-oriented databases are those that store data tables as sections of columns of data, rather than as rows of data. Shortly, they will have column families.

Row-Oriented Database	Column-Oriented Database
It is suitable for Online Transaction Process (OLTP).	It is suitable for Online Analytical Processing (OLAP).
Such databases are designed for small number of rows and columns.	Column-oriented databases are designed for huge tables.

The following image shows column families in a column-oriented database:



Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

HBase and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

Features of HBase

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters.

Where to Use HBase

- Apache HBase is used to have random, real-time read/write access to Big Data.
- It hosts very large tables on top of clusters of commodity hardware.
- Apache HBase is a non-relational database modeled after Google's Bigtable. Bigtable acts up on Google File System, likewise Apache HBase works on top of Hadoop and HDFS.

Applications of HBase

- It is used whenever there is a need to write heavy applications.
- HBase is used whenever we need to provide fast random access to available data.
- Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

What is Sharding?

Sharding is a very important concept which helps the system to keep data into different resources according to the sharding process.
The word "**Shard**" means "**a small part of a whole**". Hence Sharding means dividing a larger part into smaller parts.
In DBMS, Sharding is a type of DataBase partitioning in which a large DataBase is divided or partitioned into smaller data, also known as shards. These shards are not only smaller, but also faster and hence easily manageable.

Need for Sharding:
Consider a very large database whose sharding has not been done. For example, let's take a DataBase of a college in which all the student's record (present and past) in the whole college are maintained in a single database. So, it would contain very very large number of data, say 100, 000 records.

Now when we need to find a student from this Database, each time around 100, 000 transactions has to be done to find the student, which is very very costly.

Now consider the same college students records, divided into smaller data shards based on years. Now each data shard will have around 1000-5000 students records only. So not only the database became much more manageable, but also the transaction cost of each time also reduces by a huge factor, which is achieved by Sharding.

Hence this is why Sharding is needed.

Features of Sharding:
- Sharding makes the Database smaller
- Sharding makes the Database faster
- Sharding makes the Database much more easily manageable
- Sharding can be a complex operation sometimes
- Sharding reduces the transaction cost of the Database

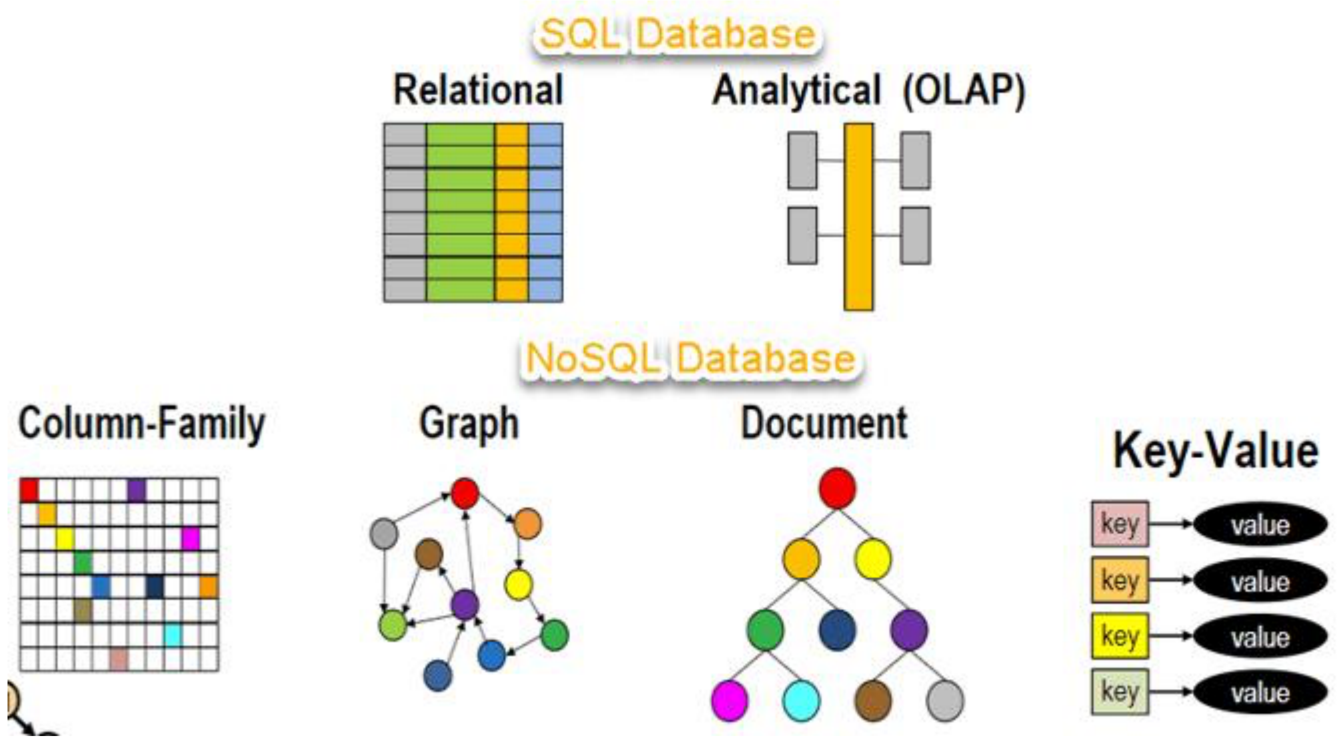
What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:

19.1M
155
What is Software Testing Why Testing is Important

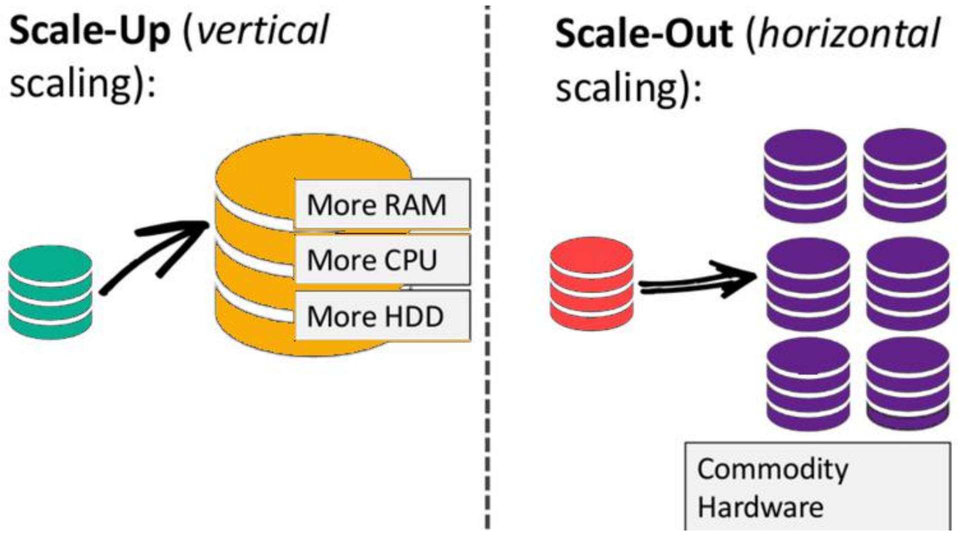


Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.

The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as "scaling out."



NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

Features of NoSQL

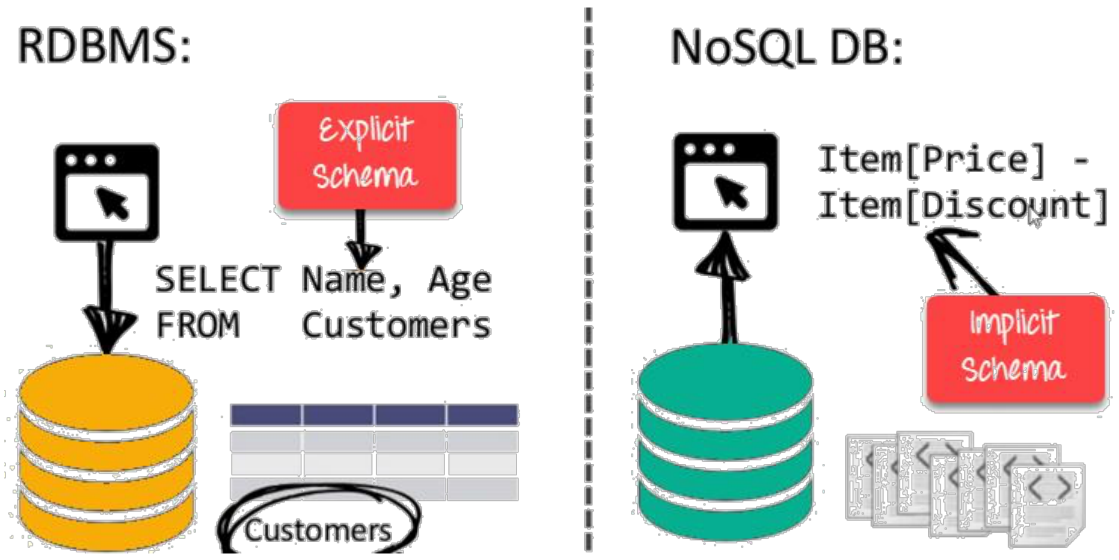
Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query

  planners, referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain



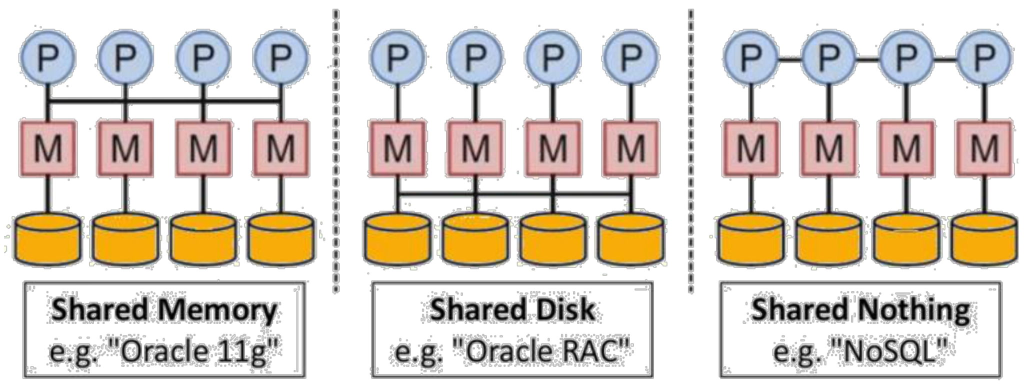
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.



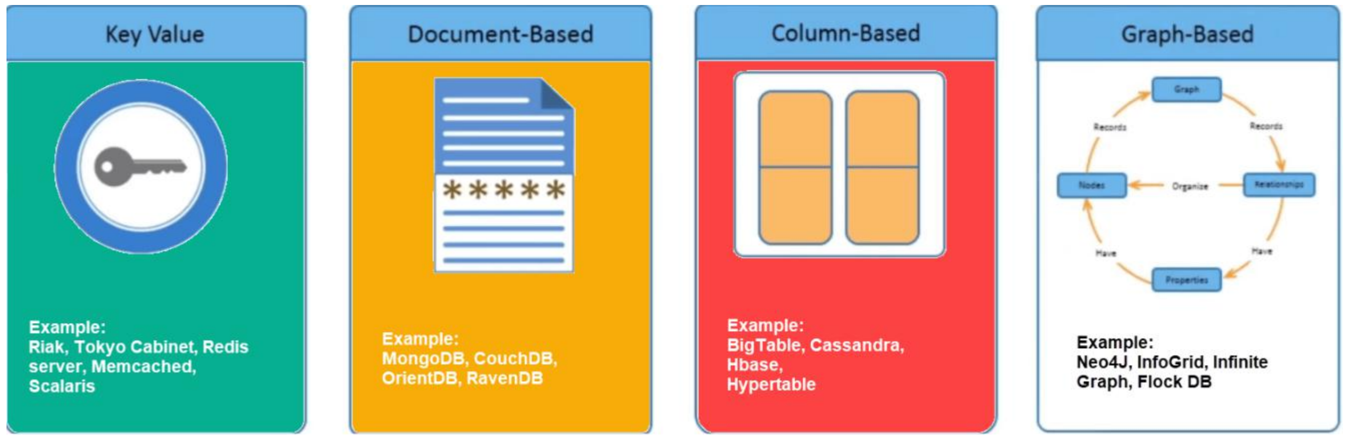
NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented

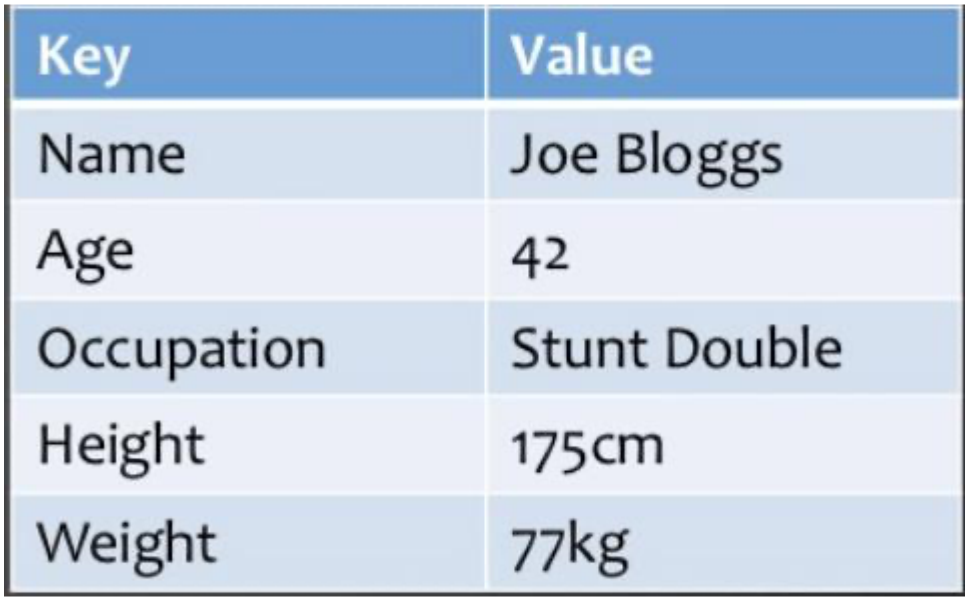

Key Value
Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris
Document-Based
Example:
MongoDB, CouchDB, OrientDB, RavenDB
Column-Based
Example:
BigTable, Cassandra, Hbase, Hypertable
Graph-Based
Graph
Records
Records
Nodes
Organize
Relationships
Have
Have
Properties
Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like "Website" associated with a value like "Guru99".



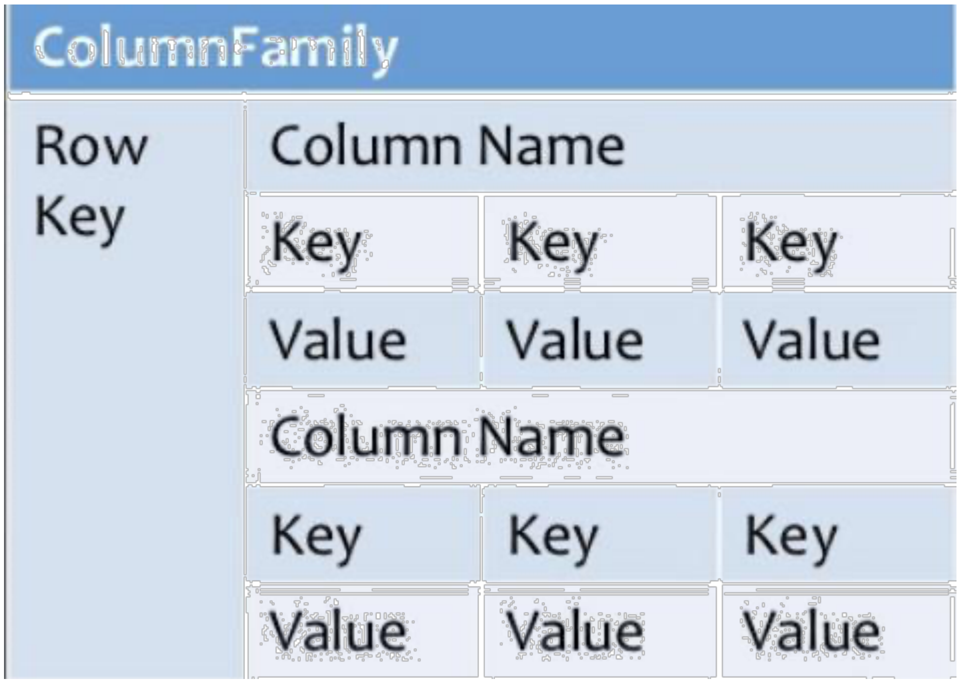
Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.



Column based NoSQL database

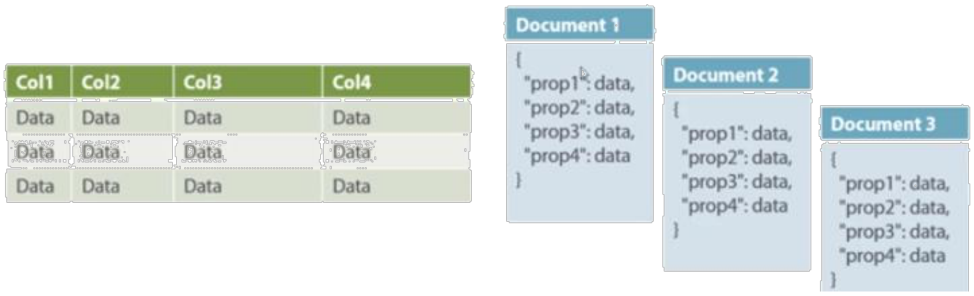
They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

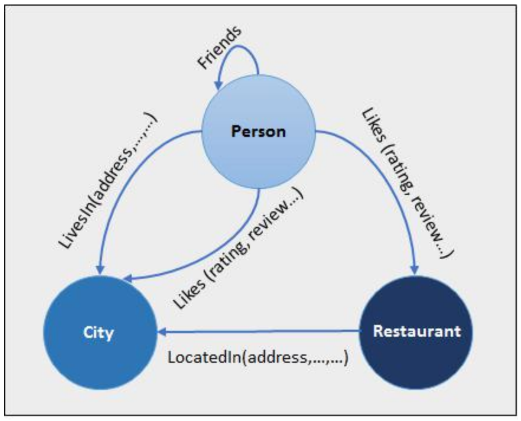
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.


Friends
Person
LivesIn(address,......)
Likes (rating, review...)
Likes (rating, review...)
City
Restaurant
LocatedIn(address,...,...)

Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

S3 (Simple Storage Service)

Amazon **S3** (Simple Storage Service) is an online service provided by Amazon.com, that allows web marketers, retailers and web-preneurs to store large amounts of data online.

S3 is **free to join**, and is a **pay-as-you-go** sevice, meaning you only ever pay for any of the hosting and bandwidth costs that you use, making it very attractive for start-up, agile and lean companies looking to **minimize costs**.

On top of this, the **fully scalable, fast** and **reliable** service provided by Amazon, makes it highly attractive to video producers and marketers all over the world.

Amazon offers S3 as a hosting system, with pricing dependent on the geographic location of the datacenter where you store your videos.

So Why Should You Use It?

1. **One of the Most Affordable Hosting Solutions on the Web**

   The obvious answer to this question is cost! Since you only ever pay for the storage and bandwidth you use, you don't have to contend with high end server costs, or pay for storage and bandwidth that you will never use. Your bill is always in line with the volume of your use.

2. **No Limits, Fully Scalable Solution**

   You should also consider the fact the Amazon S3 is fully scalable, and there are no limits to the amount of storage or bandwidth that you use. Conventional hosting companies apply limits to the majority of their plans, and once you hit them, you either get slapped with large, extra costs, or they simply suspend your account, putting your whole website out of action. Using Amazon S3 means you're no longer at the mercy of the hosting company.

3. **Reliability: S3 is provided by Amazon, a global leader in web services, with world class technical expertise**

   The S3 service is very reliable - there is currently a growing network of over 200,000 developers, it's being used by a multitude of companies of different sizes, from start-ups to Fortune 1000 companies - it's a widely tested system. This is backed up by a guarantee of 99.9% uptime from Amazon, and their service level agreement if the service ever falls below that.

4. **Protect your Own Server**
   Finally, the use of Amazon S3 allows you to reduce strain on the server hosting your website. Files such as video and audio have much larger file sizes than standard html files, and an influx of traffic to view a particular file can put undue strain on your server, and in some cases even knock it out. Outsourcing the hosting of your larger files to Amazon S3, means you can optimize your own server for hosting the website, and let Amazon worry about everything else.

The Hadoop Distributed File System (HDFS)

It is the primary data storage system used by Hadoop applications. HDFS employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters.

Hadoop itself is an open source distributed processing framework that manages data processing and storage for big data applications. HDFS is a key part of the many Hadoop ecosystem technologies. It provides a reliable means for managing pools of big data and supporting related big data analytics applications.

How does HDFS work?

HDFS enables the rapid transfer of data between compute nodes. At its outset, it was closely coupled with MapReduce, a framework for data processing that filters and divides up work among the nodes in a cluster, and it organizes and condenses the results into a cohesive answer to a query. Similarly, when HDFS takes in data, it breaks the information down into separate blocks and distributes them to different nodes in a cluster.

With HDFS, data is written on the server once, and read and reused numerous times after that. HDFS has a primary NameNode, which keeps track of where file data is kept in the cluster.

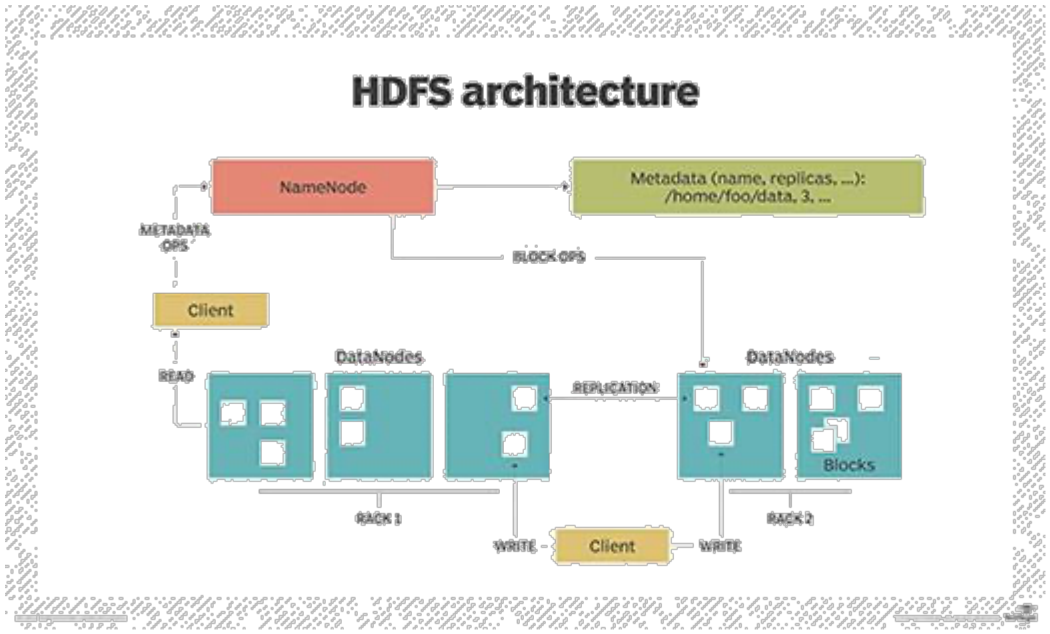
HDFS also has multiple DataNodes on a commodity hardware cluster -- typically one per node in a cluster. The DataNodes are generally organized within the same rack in the data center. Data is broken down into separate blocks and distributed among the various DataNodes for storage. Blocks are also replicated across nodes, enabling highly efficient parallel processing.

The NameNode knows which DataNode contains which blocks and where the DataNodes reside within the machine cluster. The NameNode also manages access to the files, including reads, writes, creates, deletes and the data block replication across the DataNodes.

The NameNode operates in conjunction with the DataNodes. As a result, the cluster can dynamically adapt to server capacity demand in real time by adding or subtracting nodes as necessary.

The DataNodes are in constant communication with the NameNode to determine if the DataNodes need to complete specific tasks. Consequently, the NameNode is always aware of the status of each DataNode. If the NameNode realizes that one DataNode isn't working properly, it can immediately reassign that DataNode's task to a different node containing the same data block. DataNodes also communicate with each other, which enables them to cooperate during normal file operations.

Moreover, the HDFS is designed to be highly fault-tolerant. The file system replicates -- or copies -- each piece of data multiple times and distributes the copies to individual nodes, placing at least one copy on a different server rack than the other copies.



APACHE SOFTWARE FOUNDATION
HDFS architecture centers on commanding NameNodes that hold metadata and DataNodes that store information in blocks. Working at the heart of Hadoop, HDFS can replicate data at great scale.

HDFS architecture, NameNodes and DataNodes

HDFS uses a primary/secondary architecture. The HDFS cluster's NameNode is the primary server that manages the file system namespace and controls client access to files. As the central component of the Hadoop Distributed File System, the NameNode maintains and manages the file system namespace and provides clients with the right access permissions. The system's DataNodes manage the storage that's attached to the nodes they run on.

HDFS exposes a file system namespace and enables user data to be stored in files. A file is split into one or more of the blocks that are stored in a set of DataNodes. The NameNode performs file system namespace operations, including opening, closing and renaming files and directories. The NameNode also governs the mapping of blocks to the DataNodes. The DataNodes serve read and write requests from the clients of the file system. In addition, they perform block creation, deletion and replication when the NameNode instructs them to do so.

HDFS supports a traditional hierarchical file organization. An application or user can create directories and then store files inside these directories. The file system namespace hierarchy is like most other file systems -- a user can create, remove, rename or move files from one directory to another.

The NameNode records any change to the file system namespace or its properties. An application can stipulate the number of replicas of a file that the HDFS should maintain. The NameNode stores the number of copies of a file, called the replication factor of that file.

Features of HDFS

There are several features that make HDFS particularly useful, including:

- **Data replication.** This is used to ensure that the data is always available and prevents data loss. For example, when a node crashes or there is a hardware failure, replicated data can be pulled from elsewhere within a cluster, so processing continues while data is recovered.
- **Fault tolerance and reliability.** HDFS' ability to replicate file blocks and store them across nodes in a large cluster ensures fault tolerance and reliability.
- **High availability.** As mentioned earlier, because of replication across notes, data is available even if the NameNode or a DataNode fails.
- **Scalability.** Because HDFS stores data on various nodes in the cluster, as requirements increase, a cluster can scale to hundreds of nodes.
- **High throughput.** Because HDFS stores data in a distributed manner, the data can be processed in parallel on a cluster of nodes. This, plus data locality (see next bullet), cut the processing time and enable high throughput.
- **Data locality.** With HDFS, computation happens on the DataNodes where the data resides, rather than having the data move to where the computational unit is. By minimizing the distance between the data and the computing process, this approach decreases network congestion and boosts a system's overall throughput.

What are the benefits of using HDFS?

There are five main advantages to using HDFS, including:

1. **Cost effectiveness.** The DataNodes that store the data rely on inexpensive off-the-shelf hardware, which cuts storage costs. Also, because HDFS is open source, there's no licensing fee.
2. **Large data set storage.** HDFS stores a variety of data of any size -- from megabytes to petabytes -- and in any format, including structured and unstructured data.
3. **Fast recovery from hardware failure.** HDFS is designed to detect faults and automatically recover on its own.
4. **Portability.** HDFS is portable across all hardware platforms, and it is compatible with several operating systems, including Windows, Linux and Mac OS/X.
5. **Streaming data access.** HDFS is built for high data throughput, which is best for access to streaming data.

HDFS use cases and examples

The Hadoop Distributed File System emerged at Yahoo as a part of that company's online ad placement and search engine requirements. Like other web-based companies, Yahoo juggled a variety of applications that were accessed by an increasing number of users, who were creating more and more data.

EBay, Facebook, LinkedIn and Twitter are among the companies that used HDFS to underpin big data analytics to address requirements similar to Yahoo's.

HDFS has found use beyond meeting ad serving and search engine requirements. The New York Times used it as part of large-scale image conversions, Media6Degrees for log processing and machine learning, LiveBet for log storage and odds analysis, Joost for session analysis, and Fox Audience Network for log analysis and data mining. HDFS is also at the core of many open source data lakes.

More generally, companies in several industries use HDFS to manage pools of big data, including:

- **Electric companies.** The power industry deploys phasor measurement units (PMUs) throughout their transmission networks to monitor the health of smart grids. These high-speed sensors measure current and voltage by amplitude and phase at selected transmission stations. These companies analyze PMU data to detect system faults in network segments and adjust the grid accordingly. For instance, they might switch to a backup power source or perform a load adjustment. PMU networks clock thousands of records per second, and consequently, power companies can benefit from inexpensive, highly available file systems, such as HDFS.
- **Marketing.** Targeted marketing campaigns depend on marketers knowing a lot about their target audiences. Marketers can get this information from several sources, including CRM systems, direct mail responses, point-of-sale systems, Facebook and Twitter. Because much of this data is unstructured, an HDFS cluster is the most cost-effective place to put data before analyzing it.
- **Oil and gas providers.** Oil and gas companies deal with a variety of data formats with very large data sets, including videos, 3D earth models and machine sensor data. An HDFS cluster can provide a suitable platform for the big data analytics that's needed.
- **Research.** Analyzing data is a key part of research, so, here again, HDFS clusters provide a cost-effective way to store, process and analyze large amounts of data.

HDFS data replication

Data replication is an important part of the HDFS format as it ensures data remains available if there's a node or hardware failure. As previously mentioned, the data is divided into blocks and replicated across numerous nodes in the cluster. Therefore, when one node goes down, the user can access the data that was on that node from other machines. HDFS maintains the replication process at regular intervals.