Unit-3 Big Data Tools

Big Data (CS-3032)

Kalinga Institute of Industrial Technology
Deemed to be University
Bhubaneswar-751024

School of Computer Engineering

3 Credit

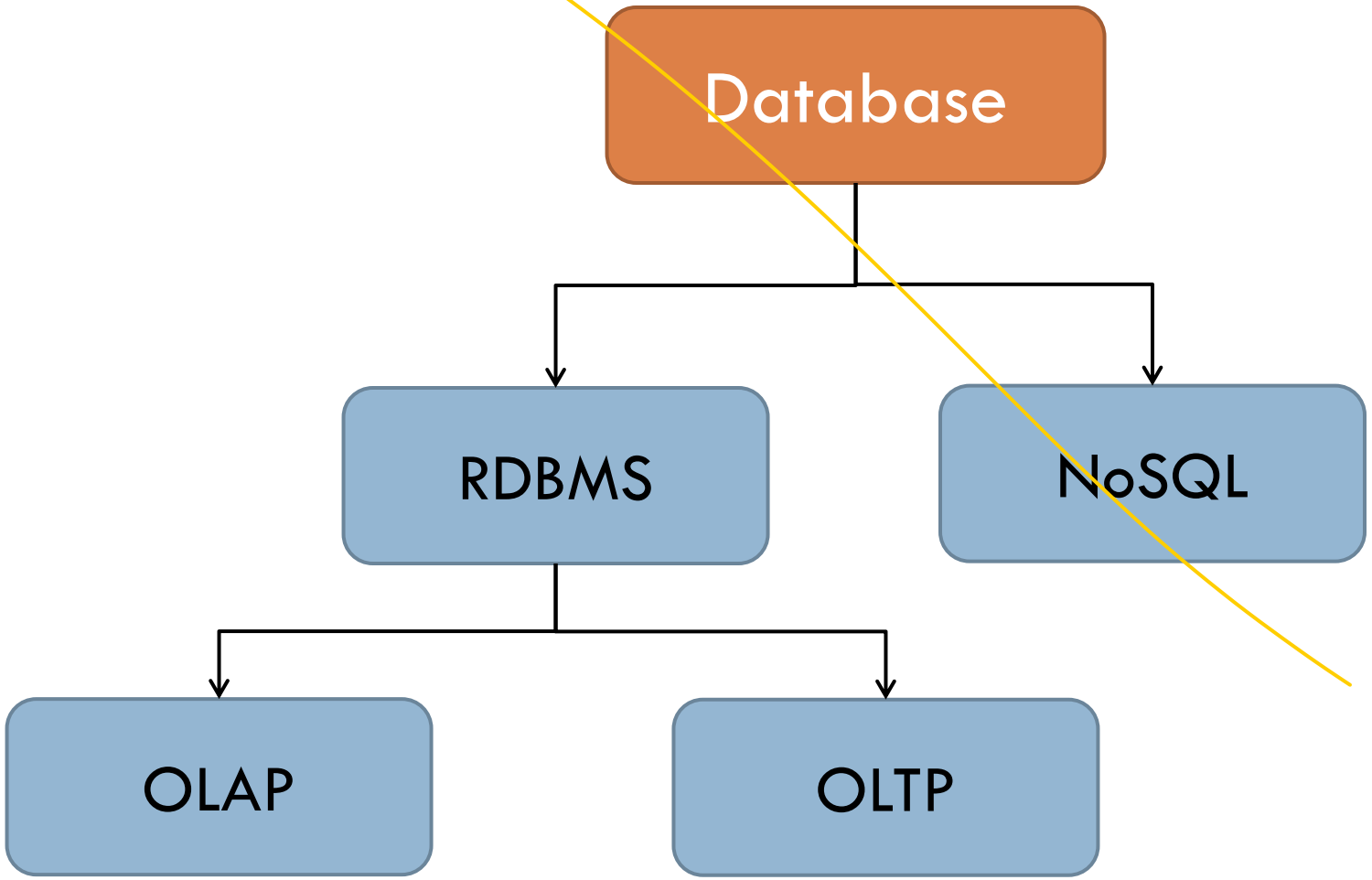
Lecture Note

Course Contents

Sr #	Major and Detailed Coverage Area	Hrs
3	**Big Data Tools**	8
	NOSQL, MapReduce – Hadoop, HDFS, Hive, MapR – Hadoop -YARN - Pig and PigLatin, Jaql - Zookeeper - HBase, Cassandra- Oozie, Lucene- Avro, Mahout. Hadoop Distributed file systems.	

NoSQL

Database

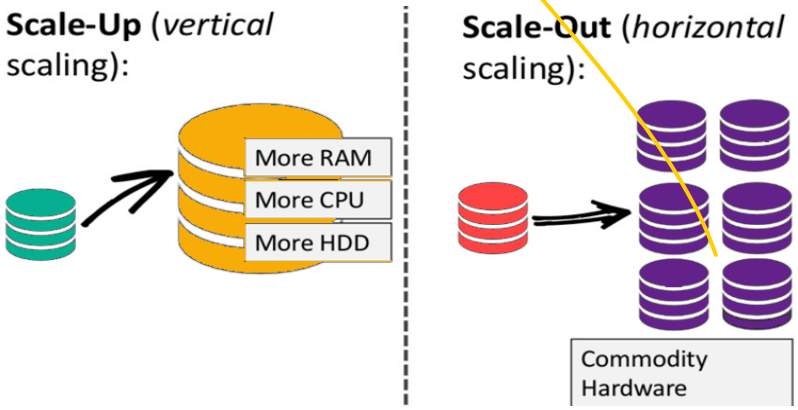


OLTP vs. OLAP

OLTP	OLAP
Many short transactions	Long transactions, complex queries
Example: - Update account balance - Add book to shopping cart - Enroll in course	Example: - Count the classes with fewer than 10 classes - Report total sales for each dept in each month
Queries touch small amounts of data (few records)	Queries touch large amounts of data
Updates are frequent	Updates are infrequent
Concurrency is biggest performance problem	Individual queries can require lots of resources

NoSQL

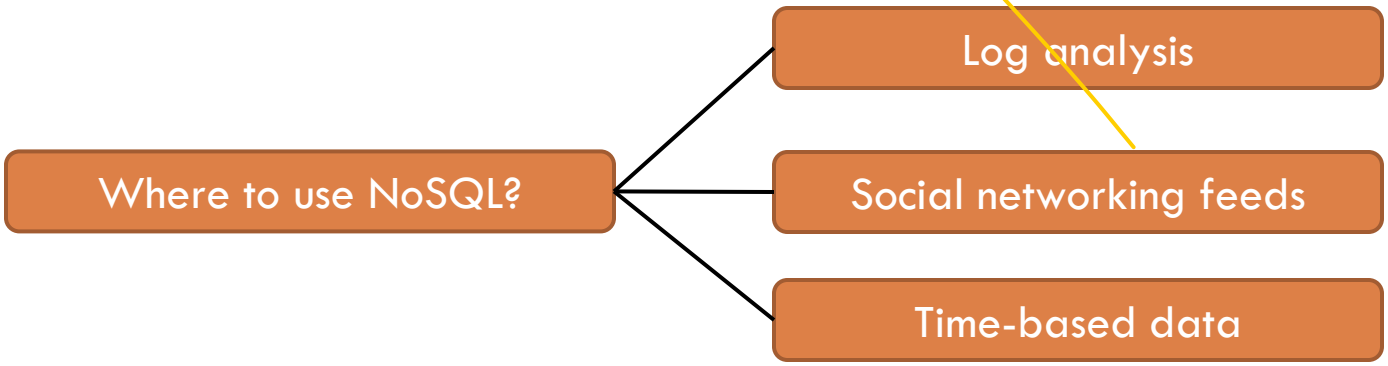
- NoSQL database stands for "Not Only SQL" or "Not SQL."
- It is a non-relational database, that does not require a fixed schema, and avoids joins.
- It is used for distributed data stores and specifically targeted for Big Data, for example Google or Facebook which collects terabytes of data every day for their users.
- 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, and unstructured data.
- It adhere to Brewer's CAP theorem.
- The tables are stored as ASCII files and each field is separated by tabs
- The data scale horizontally.



Why and Uses of NoSQL

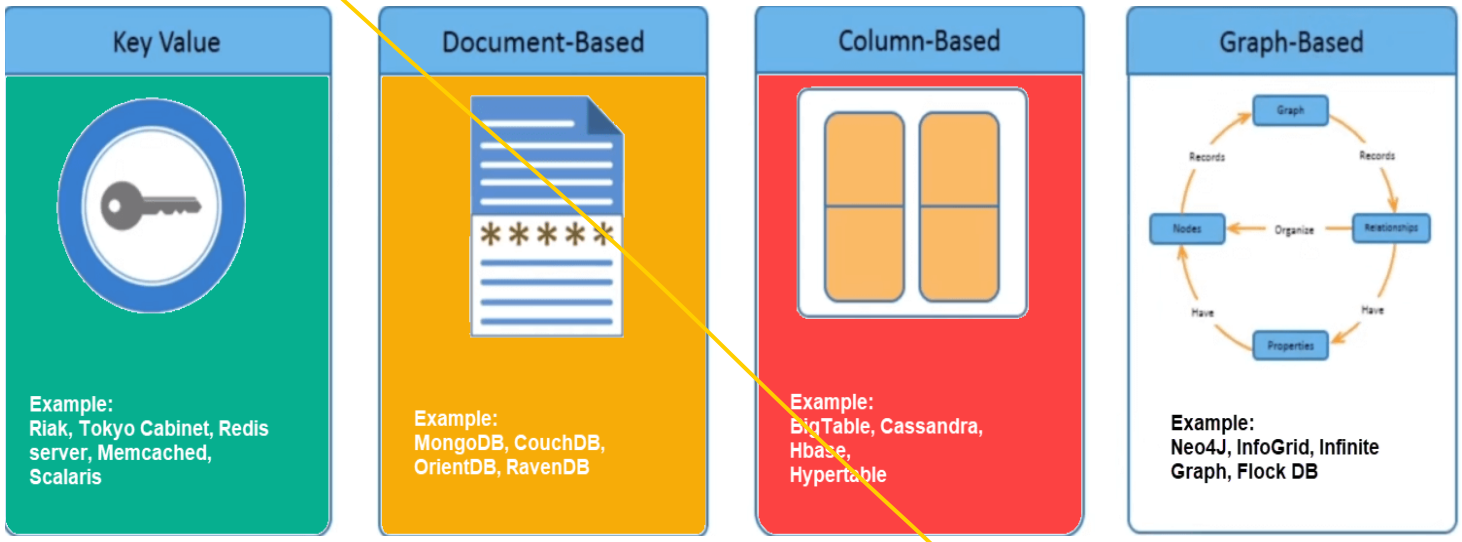
Why: In today's time data is becoming easier to access and capture through third parties such as Facebook, Google+ and others. Personal user information, social graphs, geo location data, user-generated content and machine logging data are just a few examples where the data has been increasing exponentially. To avail the above service properly, it is required to process huge amount of data which SQL databases were never designed. The evolution of NoSql databases is to handle these huge data properly.

Uses:



Types of NoSQL Database

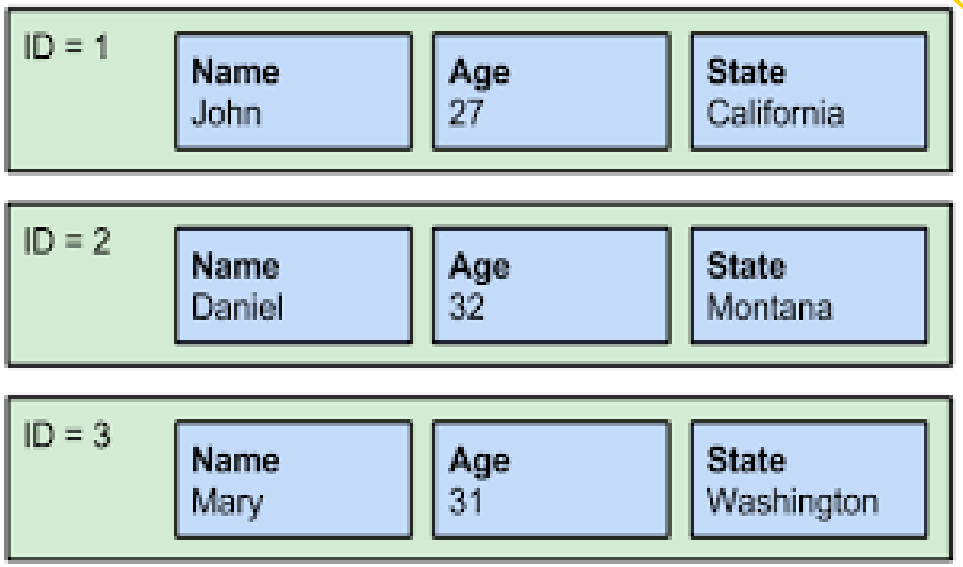
There are mainly four categories of NoSQL databases. Each of these categories has its unique attributes and limitations.



Performance	High	High	High	Variable
Scalability	High	High	Moderate	Minimal
Flexibility	High	Variable (high)	High	Variable (low)
Functionality	Variable	Variable	Variable	Graph Theory

Key Value

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, string, etc. It is one of the most basic types of NoSQL databases. 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 examples of key-value store.



SQL

ID	Name	Age	State
1	John	27	California

NoSQL – Key Value

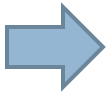
Key (i.e. ID)	Values
1	Name: John Age:27 State: California

Document-Based

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 document type is mostly used for CMS (Content Management 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.

SQL

ID	Name	Age	State
1	John	27	California



NoSQL – Document-Based

Key (ID)	Value (JSON)
1	{ "Name": John "Age":27 "State": California }

JSON vs. XML format

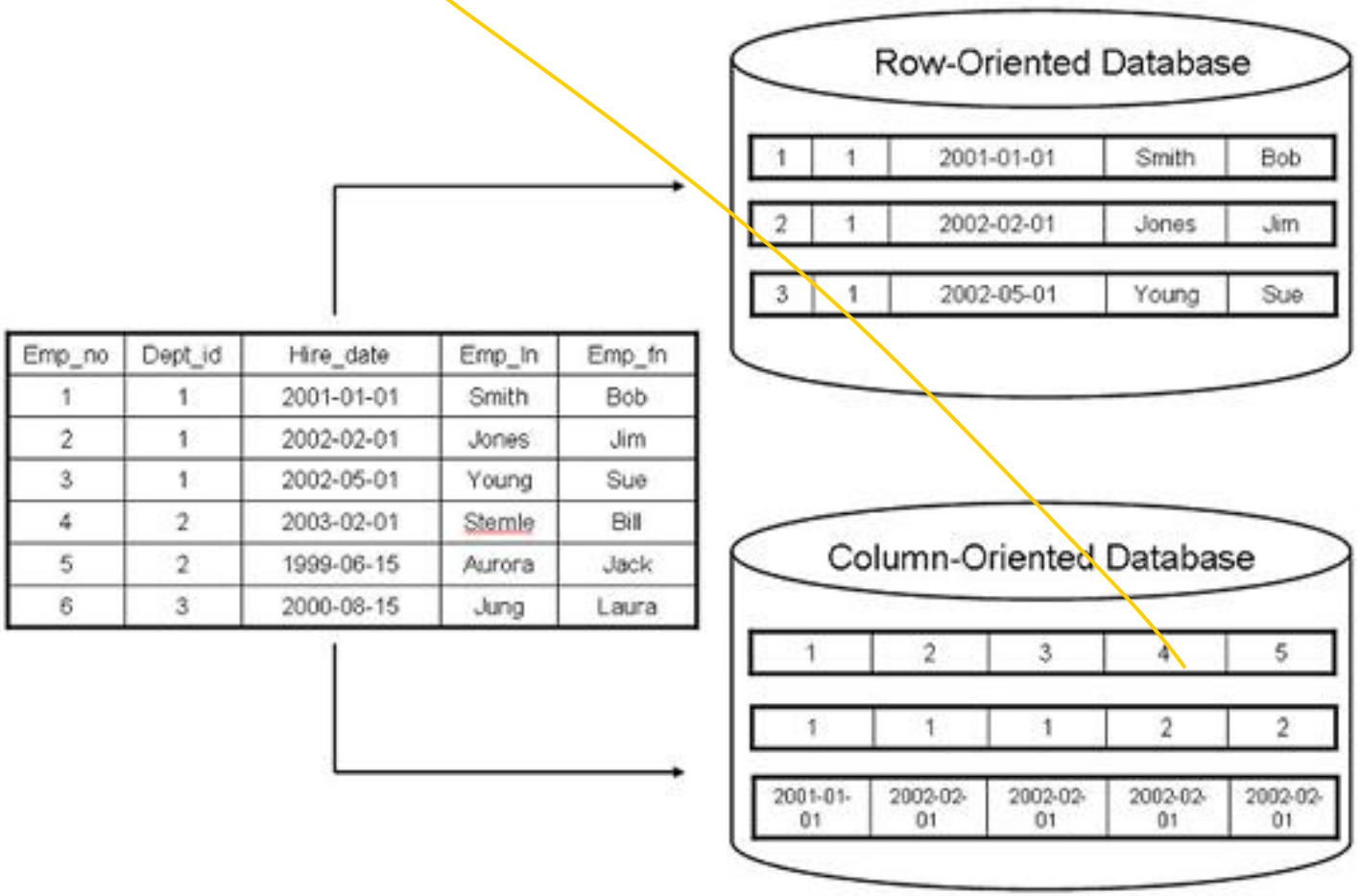
JSON

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
}
```

XML

```xml
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>
      <type>home</type>
      <number>212 555-1234</number>
    </phoneNumber>
    <phoneNumber>
      <type>fax</type>
      <number>646 555-4567</number>
    </phoneNumber>
  </phoneNumbers>
</person>
```

Column-Oriented vs. Row-Oriented Database

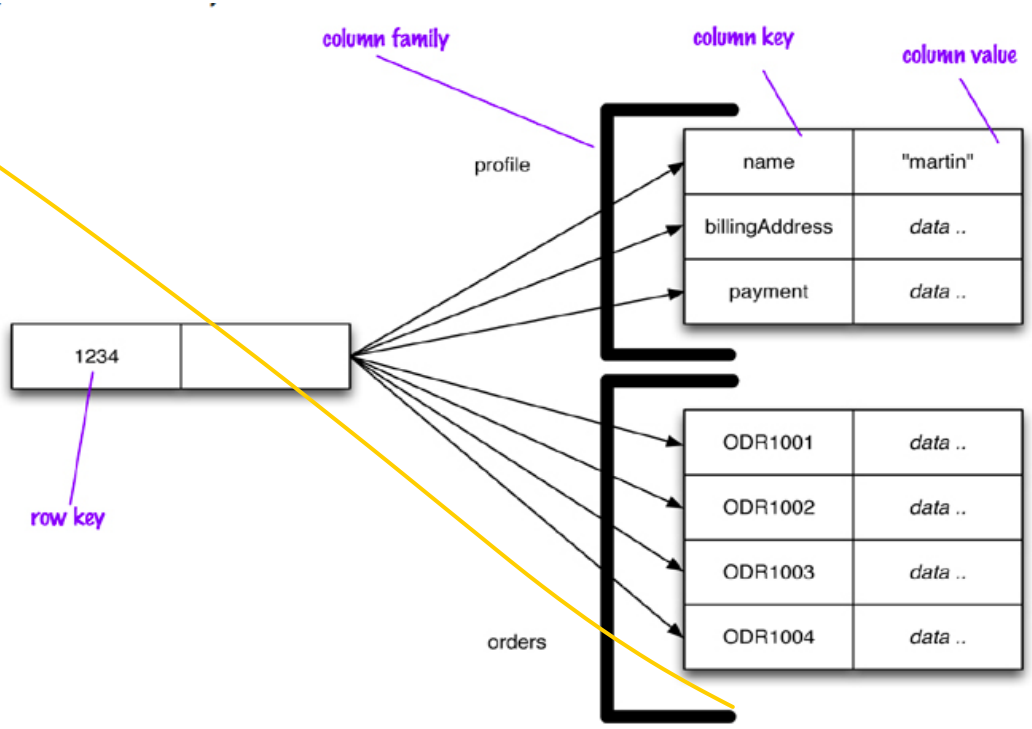


Column-Oriented vs. Row-Oriented Database cont'd

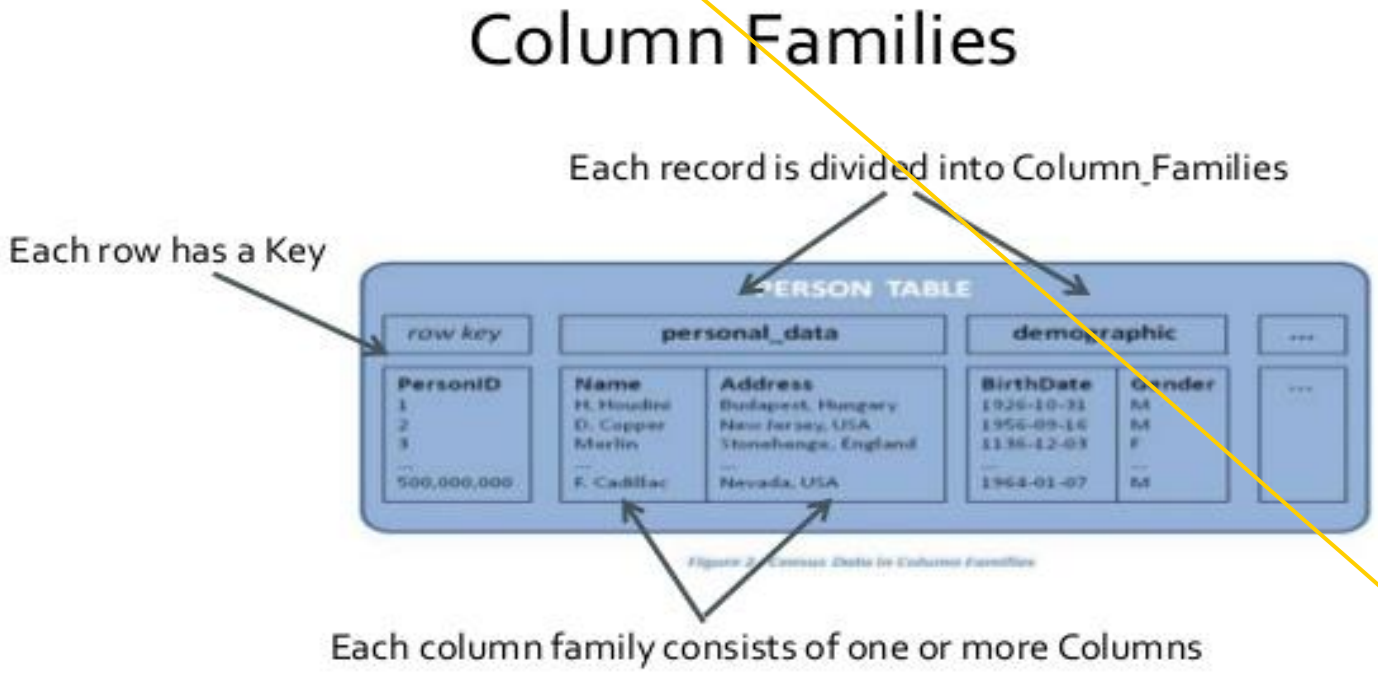
Row Oriented	Column Oriented
Data is stored and retrieved one row at a time and hence could read unnecessary data if some of the data in a row are required.	In this type of data stores, data are stored and retrieve in columns and hence it can only able to read only the relevant data if required.
Records in Row Oriented Data stores are easy to read and write.	In this type of data stores, read and write operations are slower as compared to row-oriented.
Row-oriented data stores are best suited for online transaction system.	Column-oriented stores are best suited for online analytical processing.
These are not efficient in performing operations applicable to the entire datasets and hence aggregation in row-oriented is an expensive job or operations.	These are efficient in performing operations applicable to the entire dataset and hence enables aggregation over many rows and columns.

Column-Based Database

Column-oriented databases work on column family and based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously. They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column. Such NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs etc.

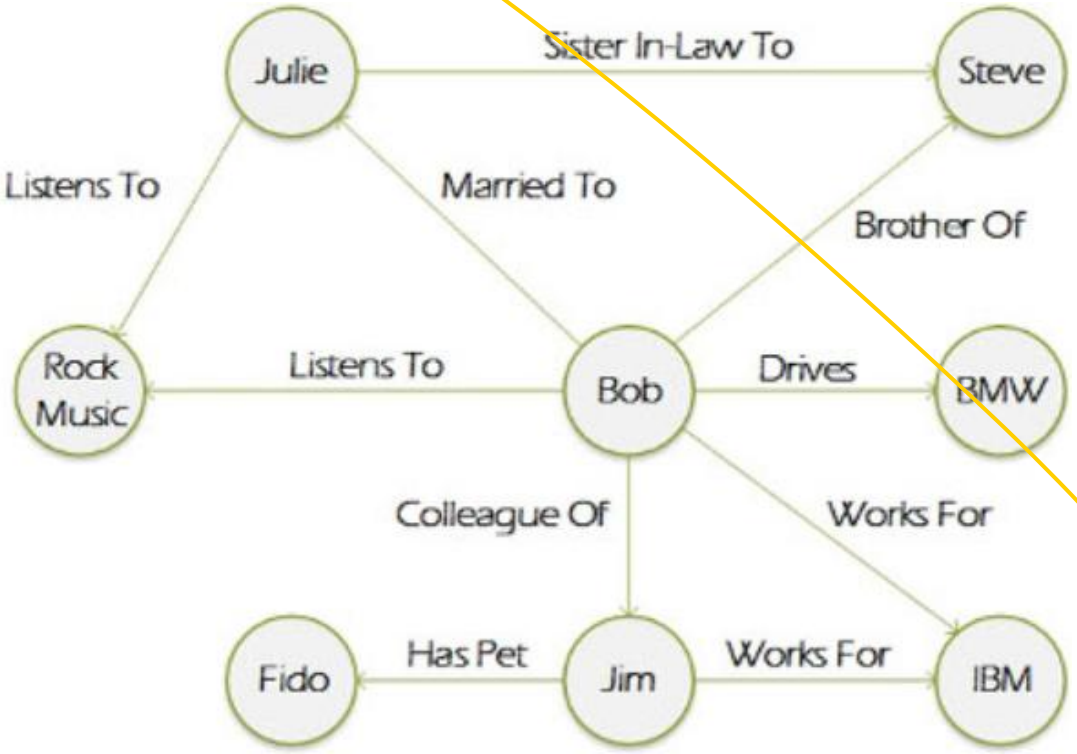


Column-Based cont'd



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



Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- Provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- NoSQL databases don't need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

SQL vs. NoSQL

SQL	NoSQL
Relational database	Non-relational, distributed database
Relational model	Model-less approach
Pre-defined schema	Dynamic schema for unstructured data
Table based databases	Document-based or graph-based or wide column store or key-value pairs databases
Vertically scalable (by increasing system resources)	Horizontally scalable (by creating a cluster of commodity machines)
Uses SQL	Uses UnQL (Unstructured Query Language)
Not preferred for large datasets	Largely preferred for large datasets
Not a best fit for hierarchical data	Best fit for hierarchical storage as it follows the key-value pair of storing data similar to JSON
Emphasis on ACID properties	Follows Brewer's CAP theorem

SQL vs. NoSQL cont'd

SQL	NoSQL
Excellent support from vendors	Relies heavily on community support
Supports complex querying and data keeping needs	Does not have good support for complex querying
Can be configured for strong consistency	Few support strong consistency (e.g., MongoDB), few others can be configured for eventual consistency (e.g., Cassandra)
Examples: Oracle, DB2, MySQL, MS SQL, PostgreSQL, etc.	Examples: MongoDB, HBase, Cassandra, Redis, Neo4j, CouchDB, Couchbase, Riak, etc.

Hadoop

Hadoop

Hadoop is an open-source project of the Apache Foundation. Apache Hadoop is written in Java and a collection of open-source software utilities that facilitate using a network of many computers to solve problems involving massive amounts of data and computation. It provides a software framework for distributed storage and processing of big data and uses Google's MapReduce and Google File System as its foundation.

Hadoop
Apache open-source software framework

Inspired by:
- Google MapReduce
- Google File System

Hadoop provides various tools and technologies, collectively termed as Hadoop ecosystem, to enable development and deployment of Big Data solutions. It accomplishes two tasks namely i) Massive data storage, and ii) Faster data processing.

Flood of data

Let's look at few stastics to get an idea of data gets generated every day, every minute, and every second.

- Every day
  - NYSE generates 1.5 billion shares and trade data
  - Facebook stores 2.7 billion comments and likes
  - Google processes about 24 petabytes of data
- Every minutes
  - Facebook users share nearly 2.5 million pieces of content.
  - Amazon generates over $ 80,000 in online sale
  - Twitter users tweet nearly 300,000 times.
  - Instagram users post nearly 220,000 new photos
  - Apple users download nearly 50,000 apps.
  - Email users send over 2000 million messages
  - YouTube users upload 72 hrs of new video content
- Every second
  - Banking applications process more than 10,000 credit card transactions.

Data Challenges

To process, analyze and made sense of these different kinds of data, a system is needed that scales and address the challenges as shown:

"I am flooded with data". How to store terabytes of mounting data?

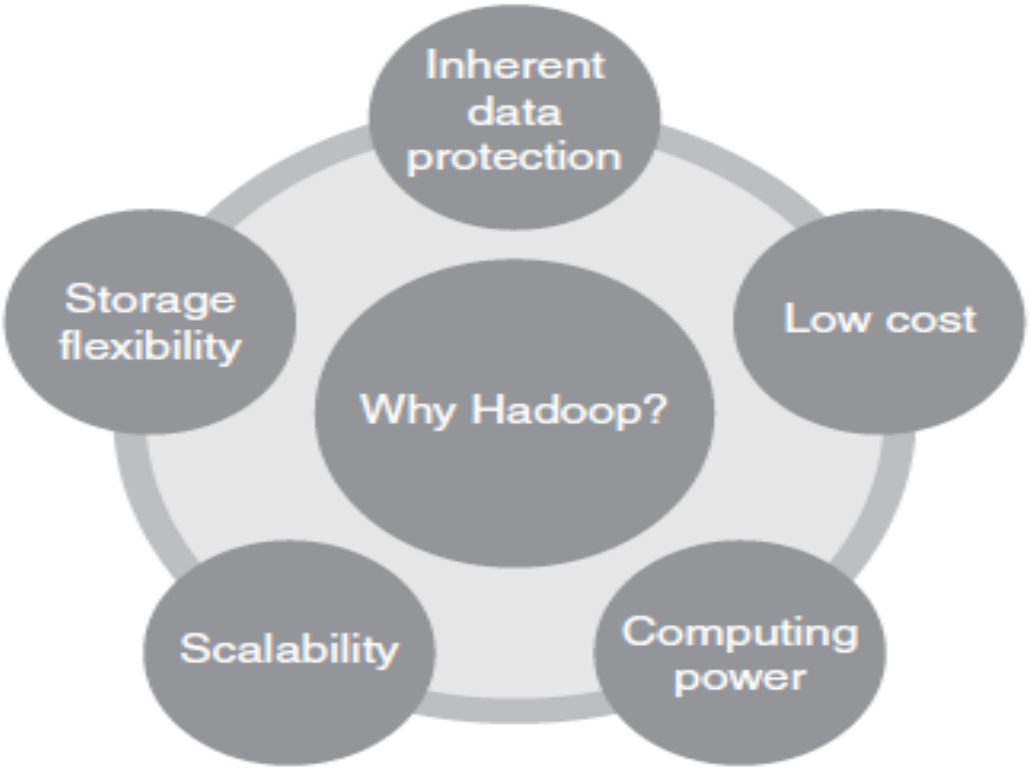
"I have data in various sources. I have data that rich in variety – structured, semi-structured and unstructured". How to work with data that is so very different?

"I need this data to be proceed quickly. My decision is pending". How to access the information quickly?

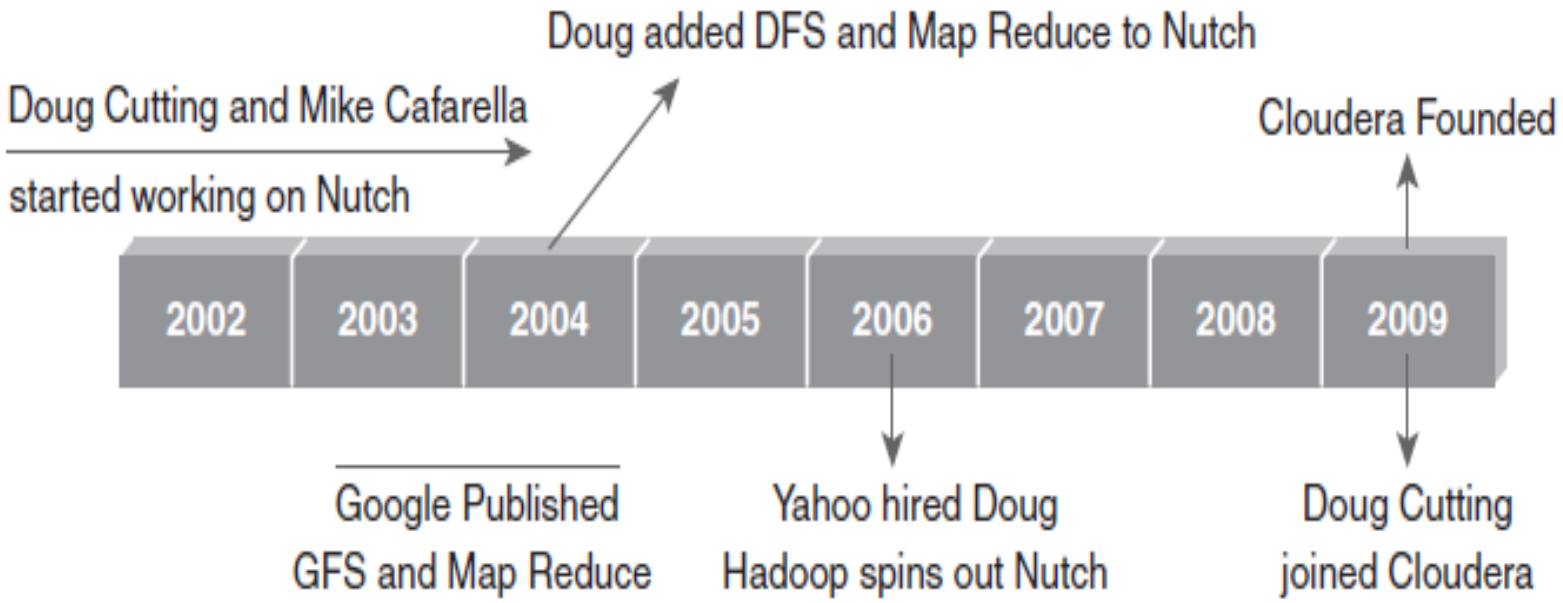
Why Hadoop

Its capability to handle massive amounts of data, different categories of data – fairly quickly.

Considerations

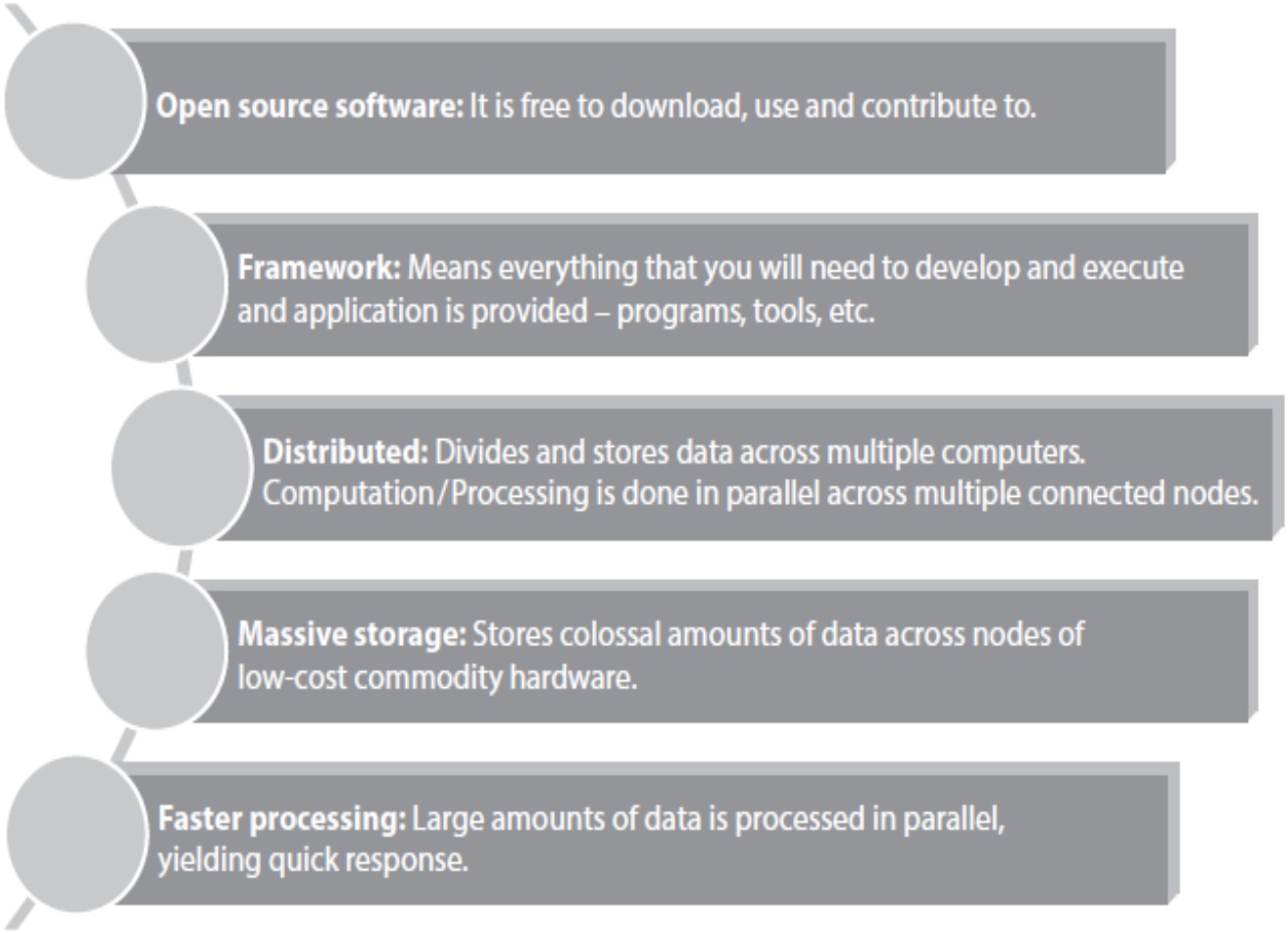


Hadoop History

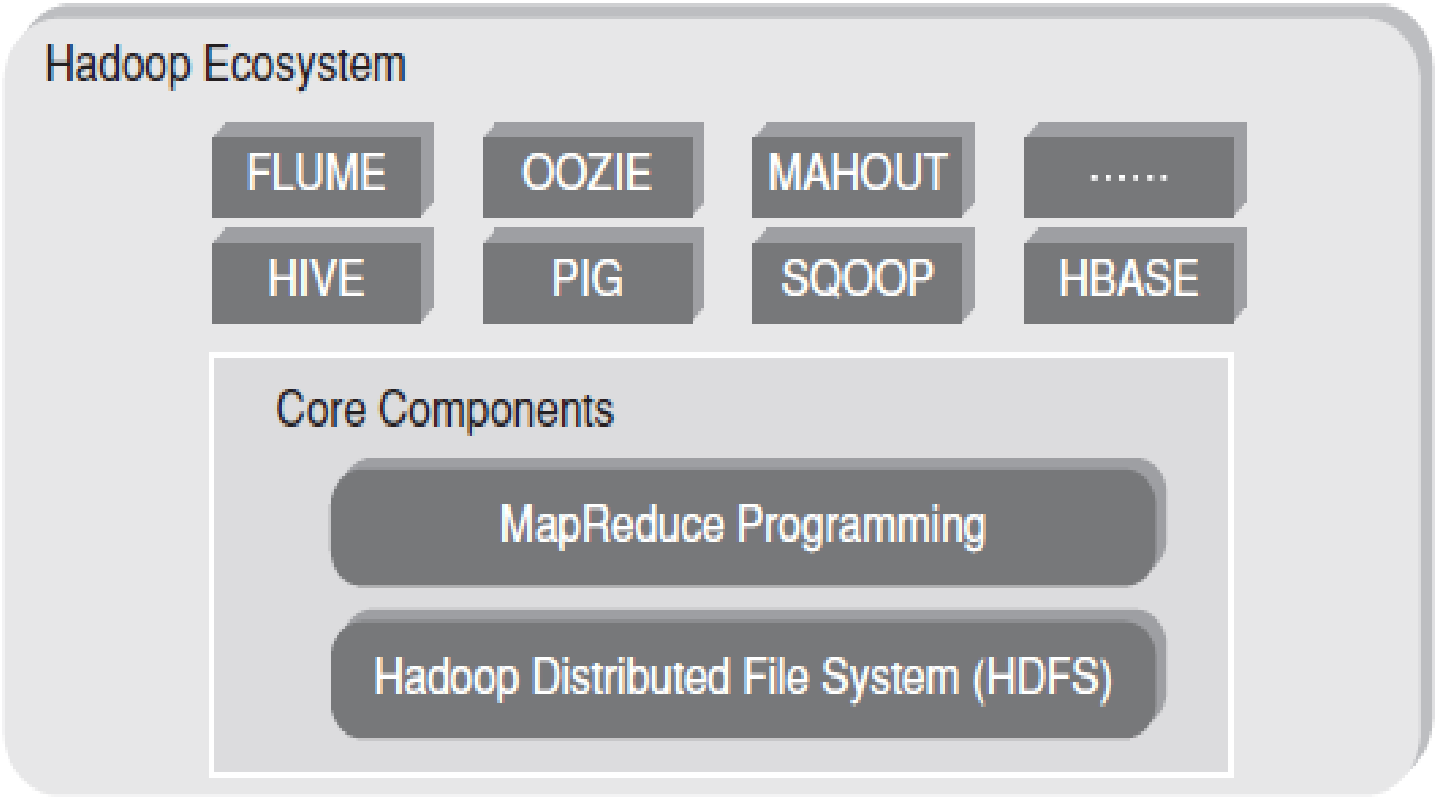


Hadoop was created by Doug Cutting, the creator of Apache Lucene (text search library). Hadoop was part of Apace Nutch (open-source web search engine of Yahoo project) and also part of Lucene project. The name Hadoop is not an acronym; it's a made-up name.

Key Aspects of Hadoop



Hadoop Components



Hadoop Components cont'd

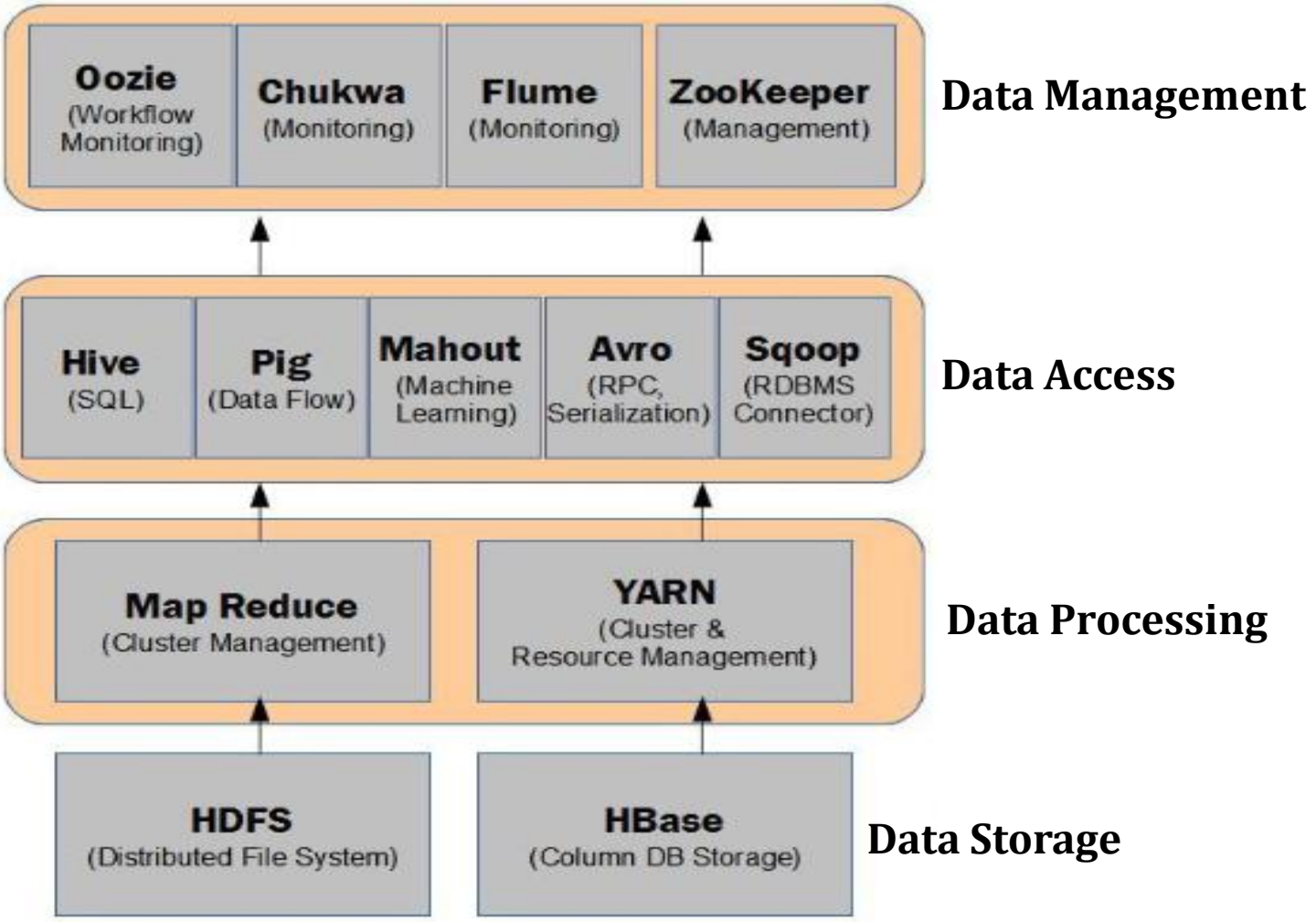
Hadoop Core Components:

- HDFS
  - Storage component
  - Distributed data across several nodes
  - Natively redundant
- MapReduce
  - Computational Framework
  - Splits a task across multiple nodes
  - Process data in parallel

Hadoop Ecosystems: These are support projects to enhance the functionality of Hadoop Core components. The projects are as follows:

- Hive
- Pig
- Sqoop
- Flume
- Oozie
- Mahout
- HBase

Hadoop Ecosystem Elements at various Stages of Data Processing

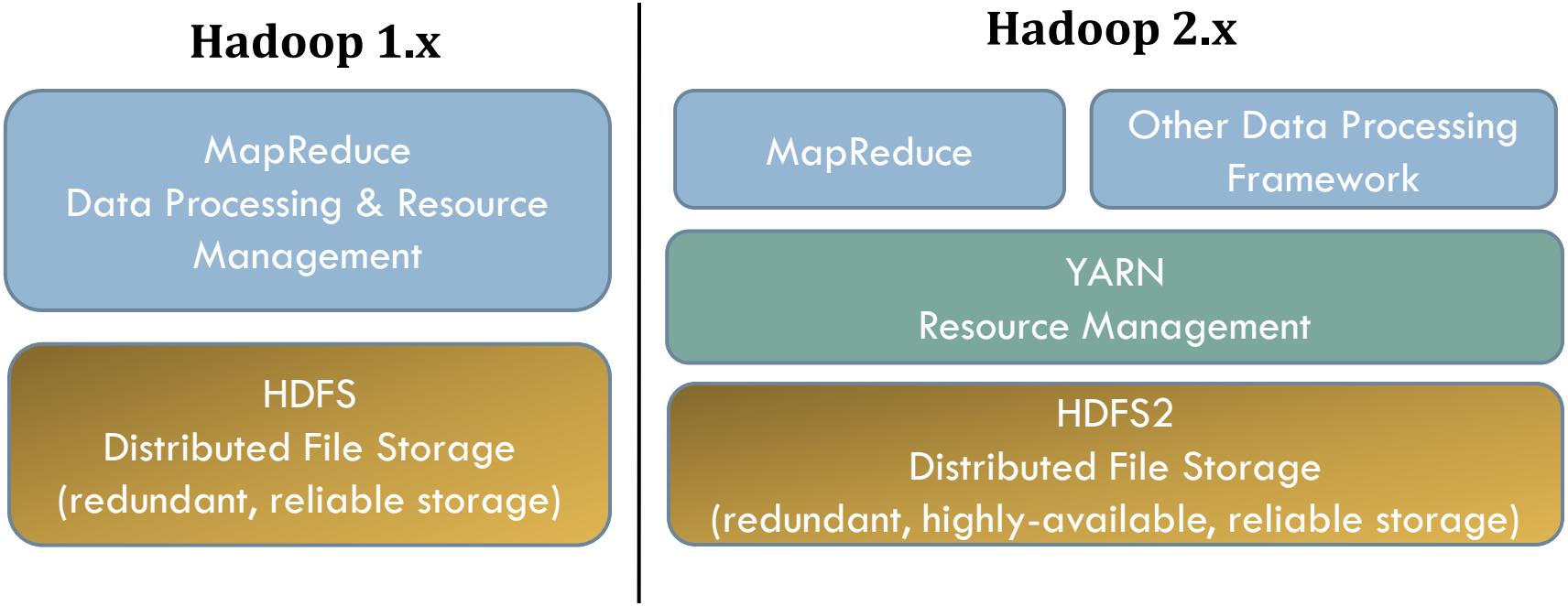


Version of Hadoop

There are 3 versions of Hadoop available:

- Hadoop 1.x
- Hadoop 2.x
- Hadoop 3.x

Hadoop 1.x vs. Hadoop 2.x

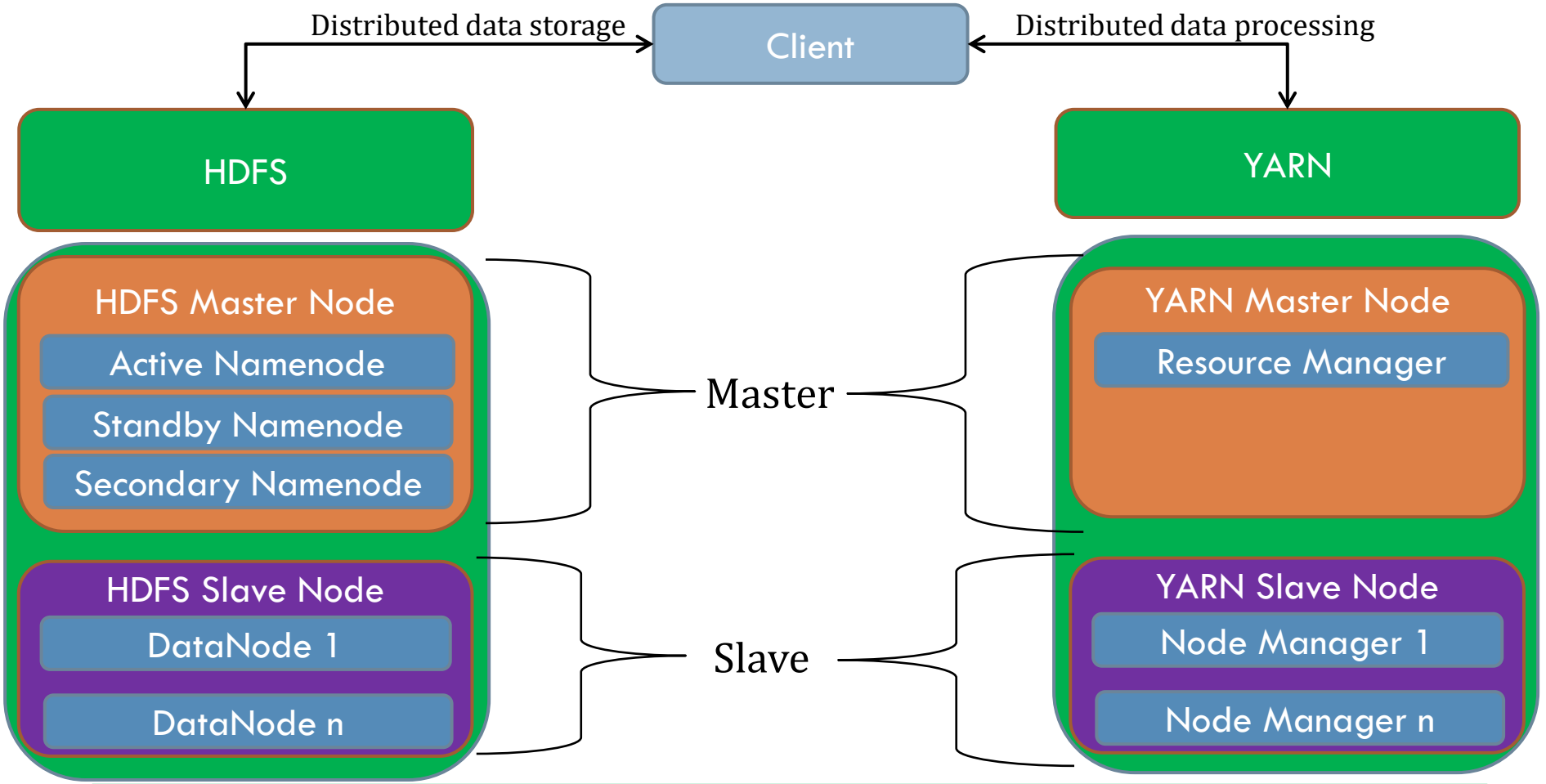


Hadoop 2.x vs. Hadoop 3.x

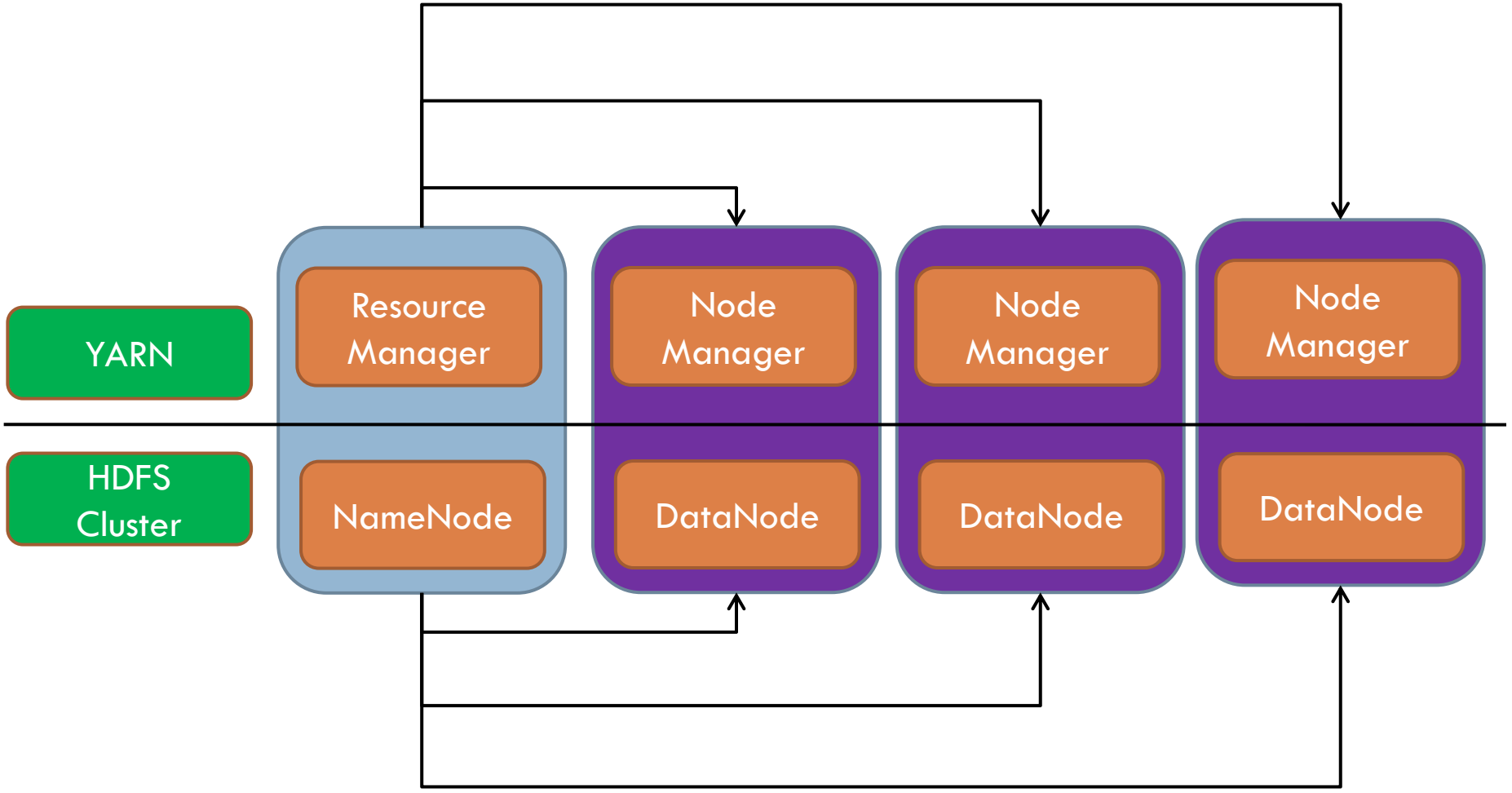
Characteristics	Hadoop 2.x	Hadoop 3.x
Minimum supported version of java	Java 7	Java 8
Fault tolerance	Handled by replication (which is wastage of space).	Handled by erasure coding
Data Balancing	Uses HDFS balancer	Uses Intra-data node balancer, which is invoked via the HDFS disk balancer CLI.
Storage Scheme	Uses 3X replication scheme. E.g. If there is 6 block so there will be 18 blocks occupied the space because of the replication scheme.	Support for erasure encoding in HDFS. E.g. If there is 6 block so there will be 9 blocks occupied the space 6 block and 3 for parity.
Scalability	Scale up to 10,000 nodes per cluster.	Scale more than 10,000 nodes per cluster.

High Level Hadoop 2.0 Architecture

Hadoop is distributed Master-Slave architecture.



High Level Hadoop 2.0 Architecture cont'd



Hadoop Distributors

The top 8 vendors offering Big Data Hadoop solution are:

- Integrated Hadoop Solution
  - Cloudera
  - HortonWorks
  - Amazon Web Services Elastic MapReduce Hadoop Distribution
  - Microsoft
  - MapR
  - IBM InfoSphere Insights
- Cloud-Based Hadoop Solution
  - Amazon Web Service
  - Google BigQuery

Hadoop HDFS

Hadoop HDFS

- The Hadoop Distributed File System (HDFS) is the primary data storage system used by Hadoop applications.
- HDFS holds very large amount of data and employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters.
- To store such huge data, the files are stored across multiple machines.
- These files are stored in redundant fashion to rescue the system from possible data losses in case of failure.
- It's run on commodity hardware.
- Unlike other distributed systems, HDFS is highly fault-tolerant and designed using low-cost hardware.

Hadoop HDFS Key points

Some key points of HDFS are as follows:

1. Storage component of Hadoop.
2. Distributed File System.
3. Modeled after Google File System.
4. Optimized for high throughput (HDFS leverages large block size and moves computation where data is stored).
5. One can replicate a file for a configured number of times, which is tolerant in terms of both software and hardware.
6. Re-replicates data blocks automatically on nodes that have failed.
7. Sits on top of native file system

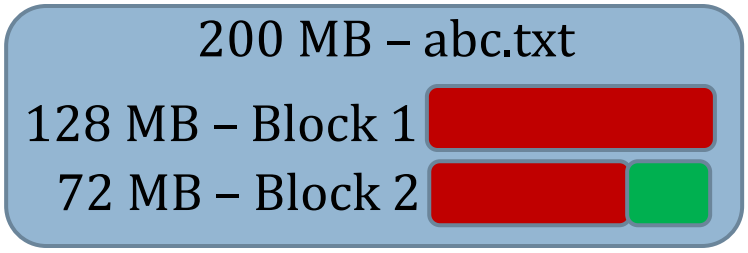
HDFS Daemons

Key components of HDFS are as follows:

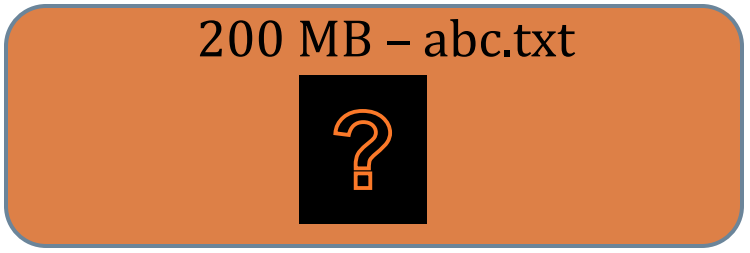
1. NameNode
2. DataNodes
3. Secondary NameNode
4. Standby NameNode

Blocks: Generally the user data is stored in the files of HDFS. HDFS breaks a large file into smaller pieces called **blocks**. **In other words, the minimum amount of data that HDFS can read or write is called a block.** By default the block size is 128 MB in Hadoop 2.x and 64 MB in Hadoop 1.x. But it can be increased as per the need to change in HDFS configuration.

Hadoop 2.X



Hadoop 1.X

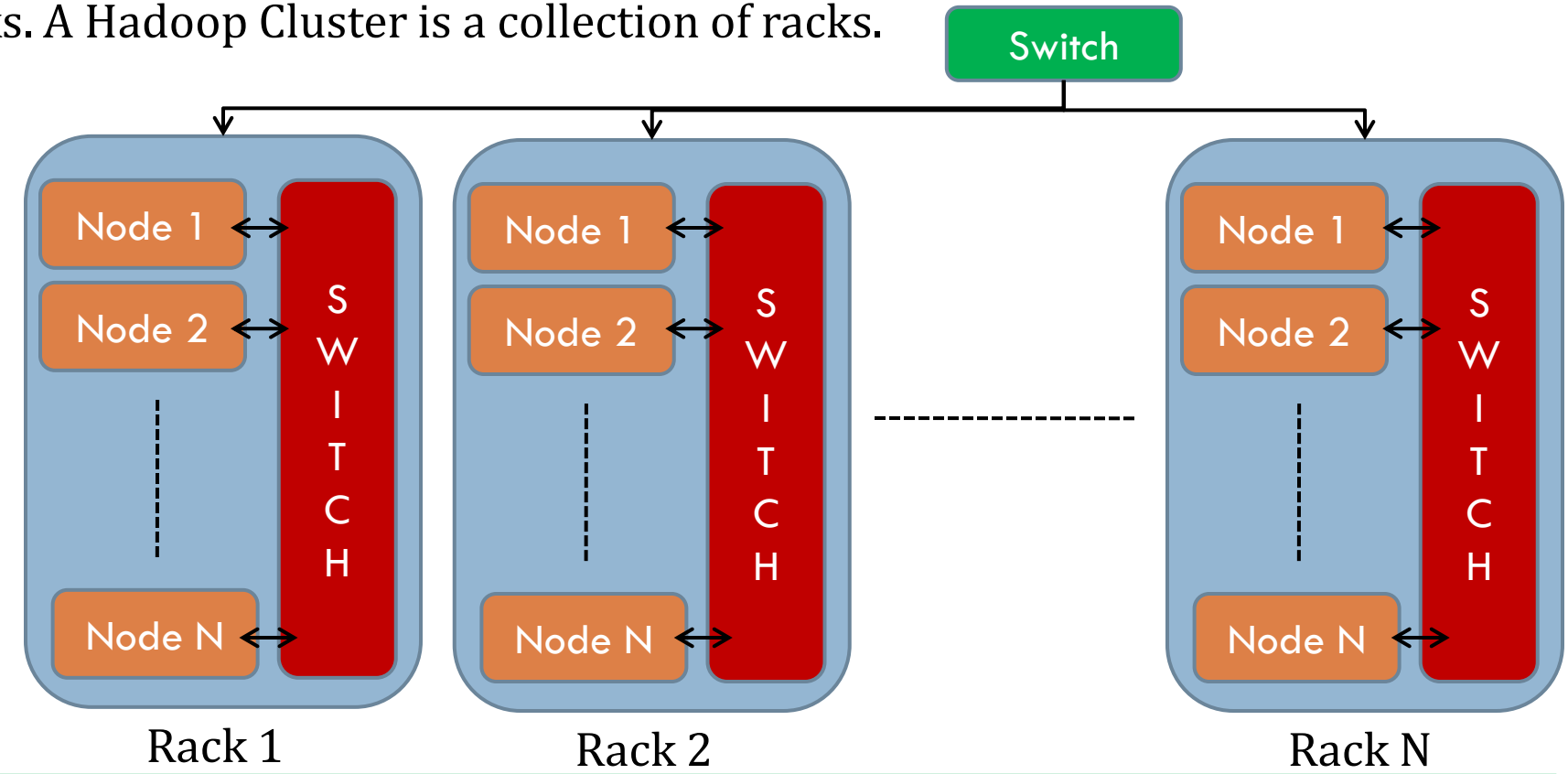


Why block size is large?

1. Reduce the cost of seek time and
2. Proper usage of storage space

Rack

A rack is a collection of 30 or 40 nodes that are physically stored close together and are all connected to the same network switch. Network bandwidth between any two nodes in rack is greater than bandwidth between two nodes on different racks. A Hadoop Cluster is a collection of racks.



NameNode

1. NameNode is the centerpiece of HDFS.
2. NameNode is also known as the Master.
3. NameNode only stores the metadata of HDFS – the directory tree of all files in the file system, and tracks the files across the cluster.
4. NameNode does not store the actual data or the dataset. The data itself is actually stored in the DataNodes
5. NameNode knows the list of the blocks and its location for any given file in HDFS. With this information NameNode knows how to construct the file from blocks.
6. NameNode is usually configured with a lot of memory (RAM).
7. NameNode is so critical to HDFS and when the NameNode is down, HDFS/Hadoop cluster is inaccessible and considered down.
8. NameNode is a single point of failure in Hadoop cluster.

Configuration

Processors: 2 Quad Core CPUs running @ 2 GHz
RAM: 128 GB
Disk: 6 x 1TB SATA
Network: 10 Gigabit Ethernet

NameNode Metadata

1. Metadata stored about the file consists of file name, file path, number of blocks, block Ids, replication level.
2. This metadata information is stored on the local disk. Namenode uses two files for storing this metadata information.
   - FsImage
   - EditLog
3. NameNode in HDFS also keeps in it's memory, location of the DataNodes that store the blocks for any given file. Using that information Namenode can reconstruct the whole file by getting the location of all the blocks of a given file.

Example

(File Name, numReplicas, rack-ids, machine-ids, block-ids, …)
/user/in4072/data/part-0, 3, r:3, M3, {1, 3}, …
/user/in4072/data/part-1, 3, r:2, M1, {2, 4, 5}, …
/user/in4072/data/part-2, 3, r:1, M2, {6, 9, 8}, …

DataNode

1. DataNode is responsible for storing the actual data in HDFS.
2. DataNode is also known as the Slave
3. NameNode and DataNode are in constant communication.
4. When a DataNode starts up it announce itself to the NameNode along with the list of blocks it is responsible for.
5. When a DataNode is down, it does not affect the availability of data or the cluster. NameNode will arrange for replication for the blocks managed by the DataNode that is not available.
6. DataNode is usually configured with a lot of hard disk space. Because the actual data is stored in the DataNode.

Configuration

Processors: 2 Quad Core CPUs running @ 2 GHz
RAM: 64 GB
Disk: 12-24 x 1TB SATA
Network: 10 Gigabit Ethernet

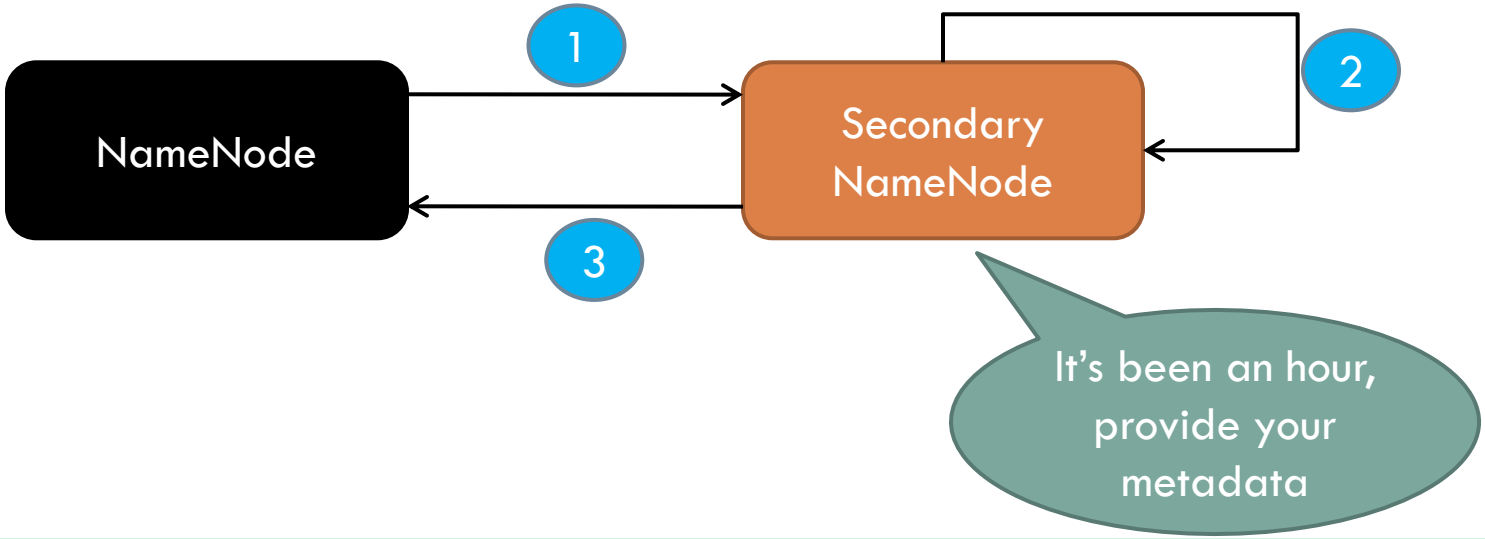
Secondary NameNode

1. Secondary NameNode in Hadoop is more of a helper to NameNode, it is not a backup NameNode server which can quickly take over in case of NameNode failure.
2. EditLog– All the file write operations done by client applications are first recorded in the EditLog.
3. FsImage– This file has the complete information about the file system metadata when the NameNode starts. All the operations after that are recorded in EditLog.
4. When the NameNode is restarted it first takes metadata information from the FsImage and then apply all the transactions recorded in EditLog. NameNode restart doesn't happen that frequently so EditLog grows quite large. That means merging of EditLog to FsImage at the time of startup takes a lot of time keeping the whole file system offline during that process.
5. Secondary NameNode take over this job of merging FsImage and EditLog and keep the FsImage current to save a lot of time. Its main function is to check point the file system metadata stored on NameNode.

Secondary NameNode cont'd

The process followed by Secondary NameNode to periodically merge the fsimage and the edits log files is as follows:

1. Secondary NameNode pulls the latest FsImage and EditLog files from the primary NameNode.
2. Secondary NameNode applies each transaction from EditLog file to FsImage to create a new merged FsImage file.
3. Merged FsImage file is transferred back to primary NameNode.



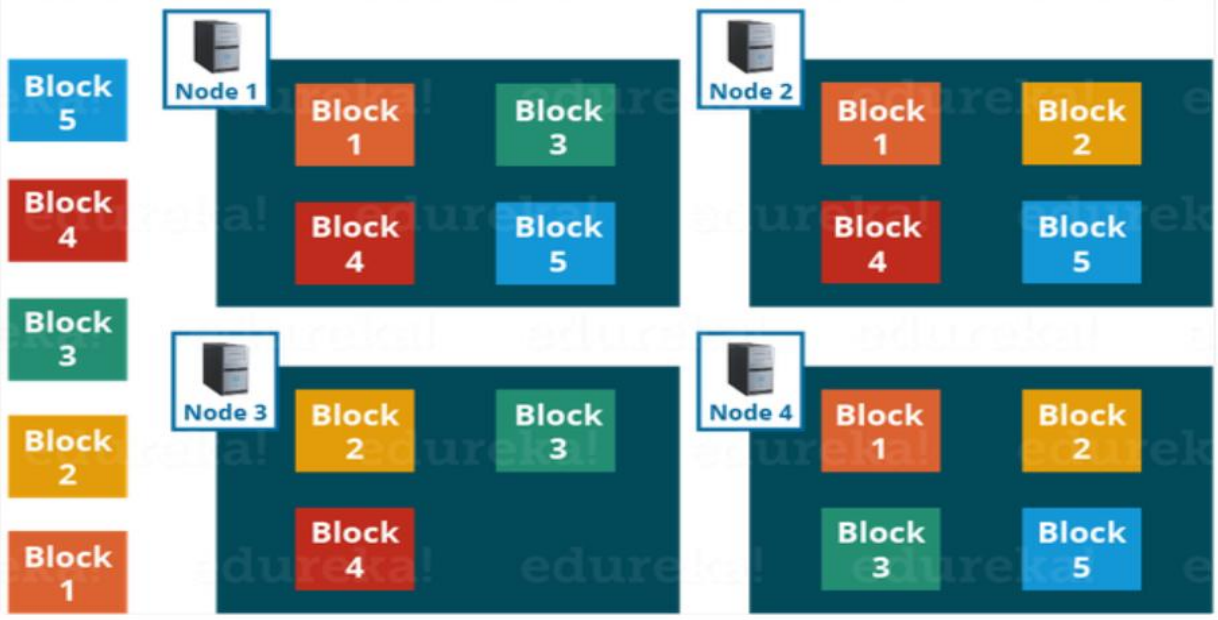
Standby NameNode

With Hadoop 2.0, built into the platform, HDFS now has **automated failover** with a **hot standby**, with full stack resiliency.

1. **Automated Failover:** Hadoop pro-actively detects NameNode host and process failures and will automatically switch to the standby NameNode to maintain availability for the HDFS service. There is no need for human intervention in the process – System Administrators can sleep in peace!
2. **Hot Standby**: Both Active and Standby NameNodes have up to date HDFS metadata, ensuring seamless failover even for large clusters – which means no downtime for your HDP cluster!
3. **Full Stack Resiliency:** The entire Hadoop stack (MapReduce, Hive, Pig, HBase, Oozie etc.) has been certified to handle a NameNode failure scenario without losing data or the job progress. This is vital to ensure long running jobs that are critical to complete on schedule will not be adversely affected during a NameNode failure scenario.

Replication

HDFS provides a reliable way to store huge data in a distributed environment as data blocks. The blocks are also replicated to provide fault tolerance. The default replication factor is 3 which is configurable. Therefore, if a file to be stored of 128 MB in HDFS using the default configuration, it would occupy a space of 384 MB (3*128 MB) as the blocks will be replicated three times and each replica will be residing on a different DataNode.

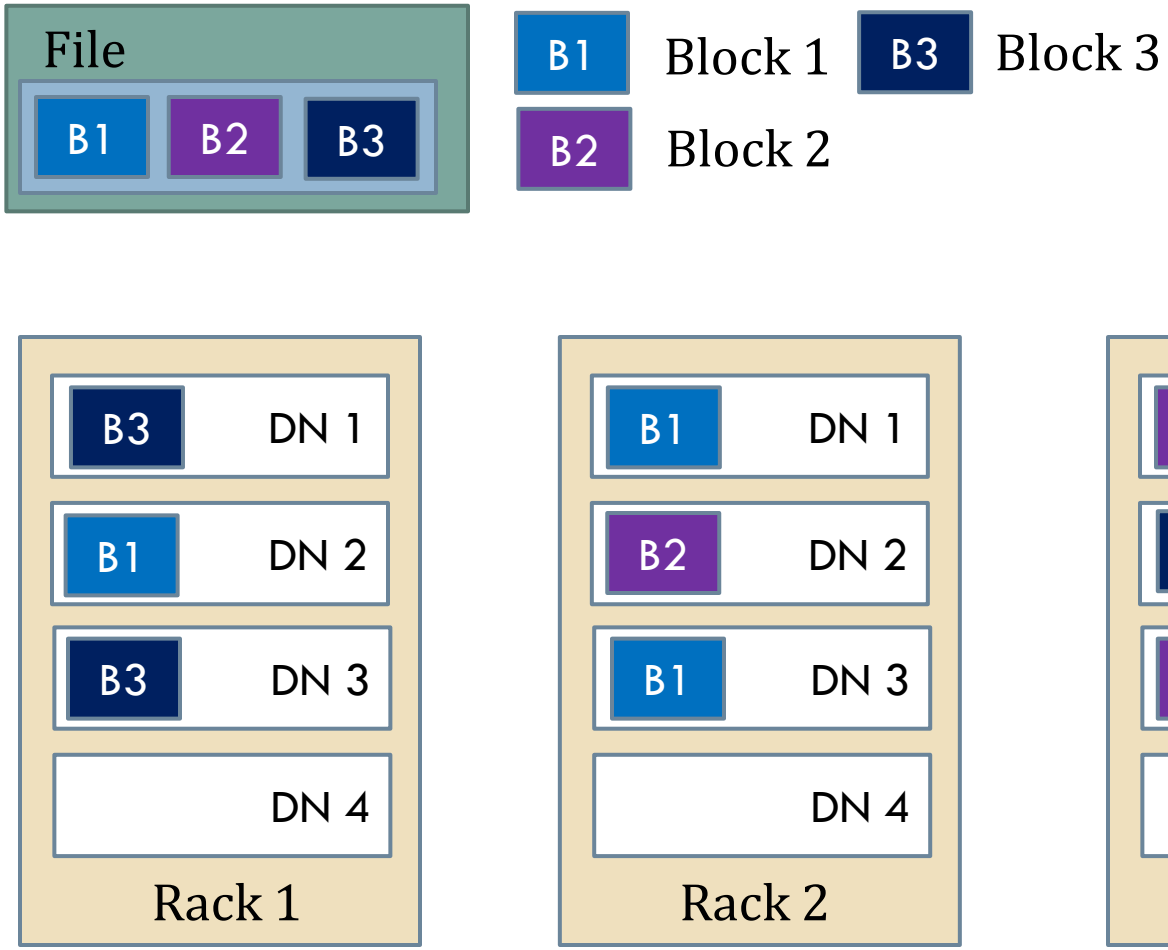


Rack Awareness

All machines in rack are connected using the same network switch and if that network goes down then all machines in that rack will be out of service. Thus the rack is down. Rack Awareness was introduced by Apache Hadoop to overcome this issue. In Rack Awareness, NameNode chooses the DataNode which is closer to the same rack or nearby rack. NameNode maintains Rack ids of each DataNode to achieve rack information. Thus, this concept chooses DataNodes based on the rack information. NameNode in Hadoop makes ensures that all the replicas should not stored on the same rack or single rack. Default replication factor is 3. Therefore according to Rack Awareness Algorithm:

- When a Hadoop framework creates new block, it places first replica on the local node, and place a second one in a different rack, and the third one is on different node on same remote node.
- When re-replicating a block, if the number of existing replicas is one, place the second on a different rack.
- When number of existing replicas are two, if the two replicas are in the same rack, place the third one on a different rack.

Rack Awareness & Replication

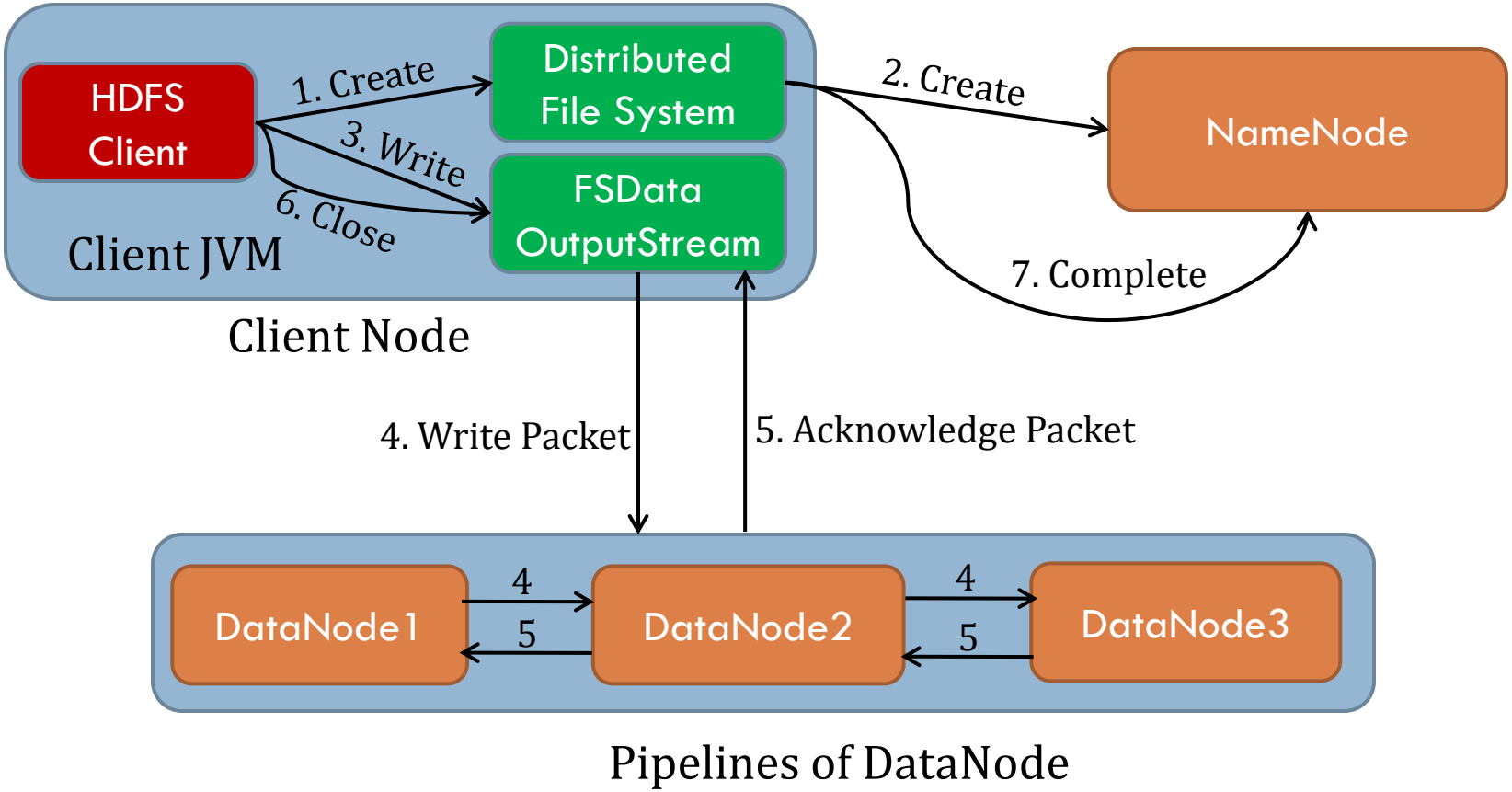


Rack Awareness Advantages

- **Provide higher bandwidth and low latency** – This policy maximizes network bandwidth by transferring block within a rack rather than between racks. The YARN is able to optimize MapReduce job performance by assigning tasks to nodes that are closer to their data in terms of network topology.
- **Provides data protection against rack failure** – Namenode assign the block replicas of 2nd And 3rd Block to nodes in different rack from the first replica. Thus, it provides data protection even against rack failure. However, this is possible only if Hadoop was configured with knowledge of its rack configuration.
- **Minimize the writing cost and Maximize read speed** – Rack awareness, policy places read/write requests to replicas which are in the same rack. Thus, this minimizes writing cost and maximizes reading speed.

Anatomy of File Write

HDFS follow **Write once Read many** models. So files can't be edited that are already stored in HDFS, but data can be appended by reopening the file.



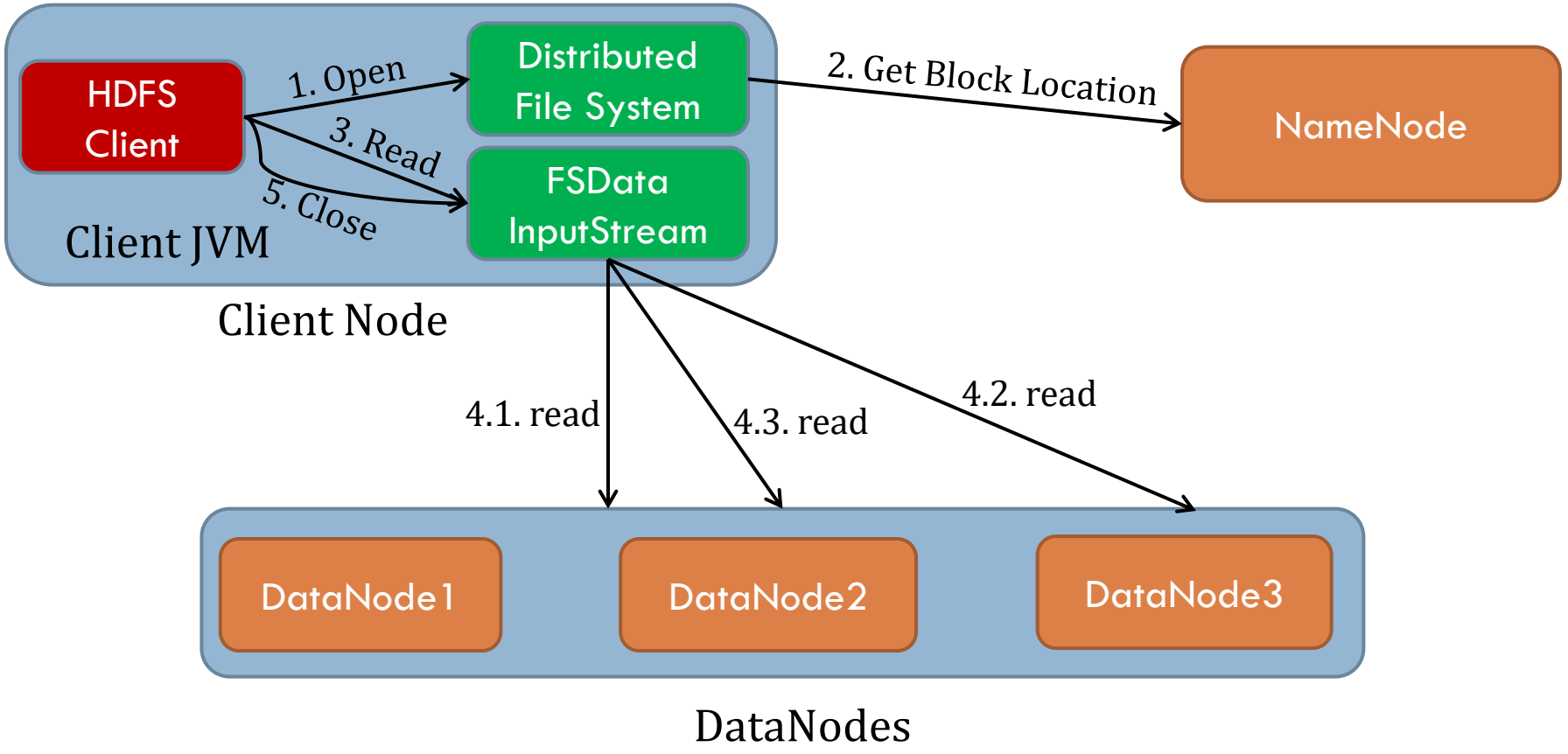
Anatomy of File Write cont'd

1. The client calls create function on DistributedFileSystem (a class extends from FileSystem) to create a file.
2. The RPC call to the NameNode happens through the DistributedFileSystem to create a new file. The NameNode performs various checks (existence of the file) to create a new file. Initially, the NameNode creates a file without associating any data blocks to the file. The DistributedFileSystem returns an FSDataOutputStream (i.e. class instance) to the client to perform write.
3. As the client writes data, data is split into packets by DFSOutputStream (i.e. a class), which is then written to the internal queue, called data queue. DataStreamer (i.e. a class) consumes the data queue. The DataStreamer requests the NameNode to allocate new blocks by selecting a list of suitable DataNodes to store replicas. The list of DataNodes makes a pipeline. With the default replication factor of 3, there will be 3 nodes in the pipeline for the first block.

Anatomy of File Write cont'd

4. DataStreamer streams the packets to first DataNode in the pipeline. It stores packet and forwards it to the second DataNode in the pipeline. In the sameway, the cecond DataNode stores the packet and forwards to the third DataNode in the pipeline.
5. In additional to the internal queue, DFSOutputStream also manages an "Ack queue" of packets that are waiting for the acknowledgement by DataNodes. A packet is removed from the "Ack Queue" only if it is acknowledged by all the DataNodes in the pipeline.
6. When the client finishes writing the file, it calls close() on the stream.
7. This flushes all the remaining packets to the DataNode pipeline and waits for relevant acknowledgements before communicating with the NameNode to inform the client that the creation of file is complete.

Anatomy of File Read



Anatomy of File Write cont'd

1. The client opens the file that it wishes to read from by calling open() on the DistributedFileSystem
2. DistributedFileSystem communicates with the NameNode to get the location of the data blocks. NameNode returns the address of the DataNodes that the data blocks are stored on. Subsequent to this, DistributedFileSystem returns DFSInputStream (i.e. a class) to the client to read from the file.
3. Client then calls read() on the stream DFSInputStream, which has address of the DataNodes for the first few blocks of the file, connects to the closet DataNode for the first block in the file.
4. Client calls read() repeatedly to stream the data from the DataNode.
5. When the end of the block is reached, DFSInputStream closes the connection with the DataNode. It repeats the steps to find the best DataNode for the next block and subsequent blocks.
6. When the client completes the reading of the file, it calls close() on FSDataInputStream to close the connection.

HDFS Commands

- To get the list of directories and files at the root of HDFS.
  hadoop fs –ls /
- To create a directory (say, sample) in HDFS
  hadoop fs –mkdir /sample
- To copy a file from local file system to HDFS
  hadoop fs –put /root/sample/test.txt /sample/test.txt
- To copy a file from HDFS to local file system
  hadoop fs –get /sample/test.txt /root/sample/test.txt
- To display the contents of an HDFS file on console
  hadoop fs –cat /sample/test.txt
- To copy a file from one directory to another on HDFS
  hadoop fs –cp /sample/test.txt /sample1
- To remove a directory from HDFS
  hadoop fs –rm –r /sample1

HDFS Example

Let's assume that this sample.txt file contains few lines as text. The content of the file is as follows:

```
Hello I am expert in Big Data
How can I help you
How can I assist you
Are you an engineer
Are you looking for coding
Are you looking for interview questions
what are you doing these days
what are your strengths
```

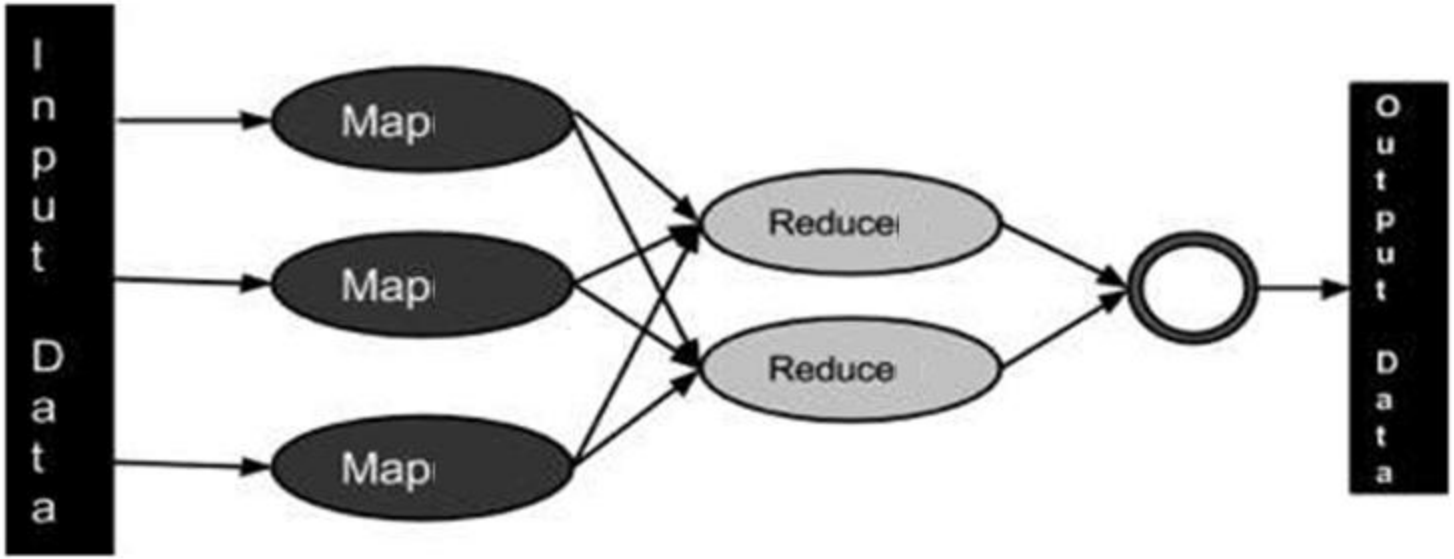
Hence, the above 8 lines are the content of the file. Let's assume that while storing this file in Hadoop, HDFS broke this file into four parts and named each part as first.txt, second.txt, third.txt, and fourth.txt. So, you can easily see that the above file will be divided into four equal parts and each part will contain 2 lines. First two lines will be in the file first.txt, next two lines in second.txt, next two in third.txt and the last two lines will be stored in fourth.txt. All these files will be stored in DataNodes and the Name Node will contain the metadata about them. All this is the task of HDFS.

Data Processing with Hadoop

Data Processing with Hadoop

1. MapReduce is a processing technique and a program model for distributed computing based on java. It is built on divide and conquer algorithm.
2. In MapReduce Programming, the input dataset in split into independent chunks.
3. It contains two important tasks, namely Map and Reduce.
4. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The processing primitive is called mapper. The processing is done in parallel manner. The output produced by the map tasks serves as intermediate data and is stored on the local disk of that server.
5. Reduce task takes the output from a map as an input and combines those data tuples into a smaller set of tuples. The processing primitive is called reducer. The input and output are stored in a file system.
6. Reduce task is always performed after the map job.
7. The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes and takes care of other tasks such as scheduling, monitoring, re-executing failed tasks etc.

Data Processing with Hadoop cont'd



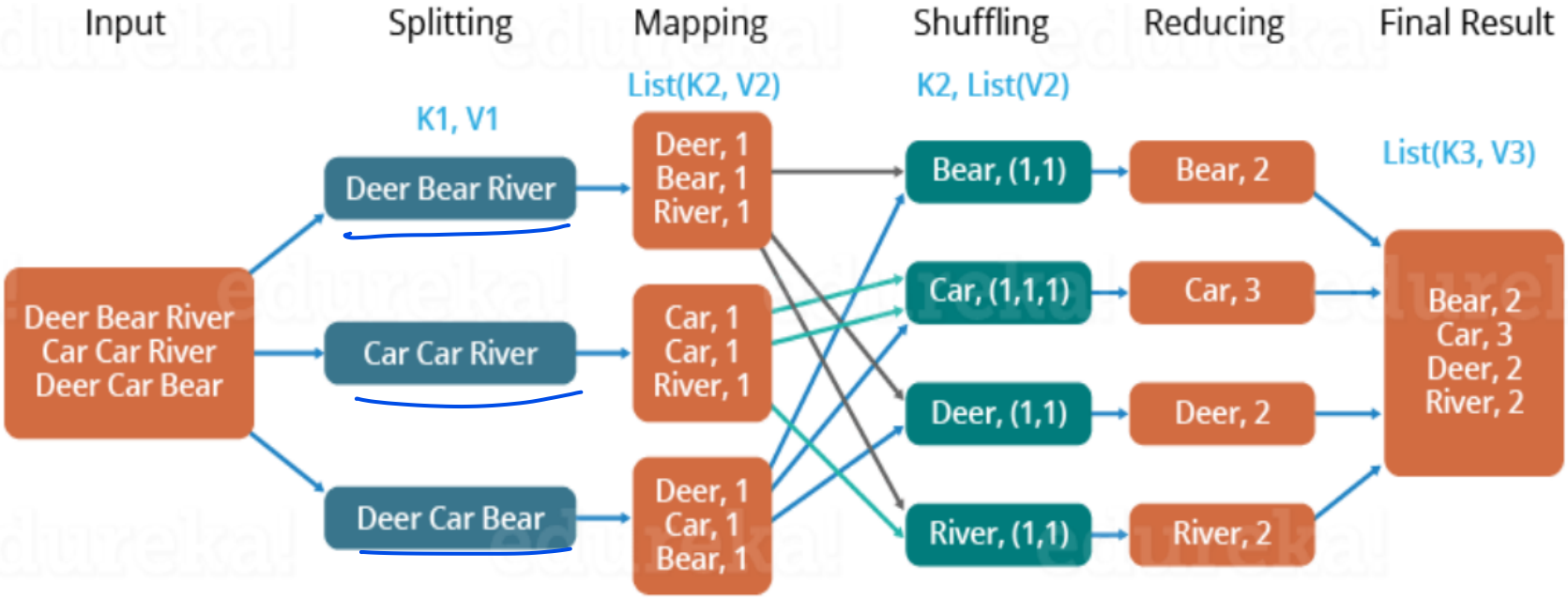
Data Processing with Hadoop cont'd

- The main advantages is that we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster with a configuration change.
- MapReduce program executes in three stages: map stage, shuffle & sorting stage, and reduce stage.
- **Map Stage:** The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
- **Shuffle & Sorting Stage:** Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce. Sort phase in MapReduce covers the merging and sorting of map outputs.
- **Reducer Stage:** The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

How MapReduce Work?

At the crux of MapReduce are two functions: Map and Reduce. They are sequenced one after the other.

- The Map function takes input from the disk as <key,value> pairs, processes them, and produces another set of intermediate <key,value> pairs as output.
- The Reduce function also takes inputs as <key,value> pairs, and produces <key,value> pairs as output.



Working of MapReduce

The types of keys and values differ based on the use case. All inputs and outputs are stored in the HDFS. While the map is a mandatory step to filter and sort the initial data, the reduce function is optional.

<k1, v1> -> Map() -> list(<k2, v2>)
<k2, list(v2)> -> Reduce() -> list(<k3, v3>)

Mappers and Reducers are the Hadoop servers that run the Map and Reduce functions respectively. It doesn't matter if these are the same or different servers.

- **Map**: The input data is first split into smaller blocks. Each block is then assigned to a mapper for processing. For example, if a file has 100 records to be processed, 100 mappers can run together to process one record each. Or maybe 50 mappers can run together to process two records each. The Hadoop framework decides how many mappers to use, based on the size of the data to be processed and the memory block available on each mapper server.

Working of MapReduce cont'd

- **Reduce**: After all the mappers complete processing, the framework shuffles and sorts the results before passing them on to the reducers. A reducer cannot start while a mapper is still in progress. All the map output values that have the same key are assigned to a single reducer, which then aggregates the values for that key.

Class Exercise 1

Draw the MapReduce process to count the number of words for the input:

Dog Cat Rat
Car Car Rat
Dog car Rat
Rat Rat Rat

Class Exercise 2

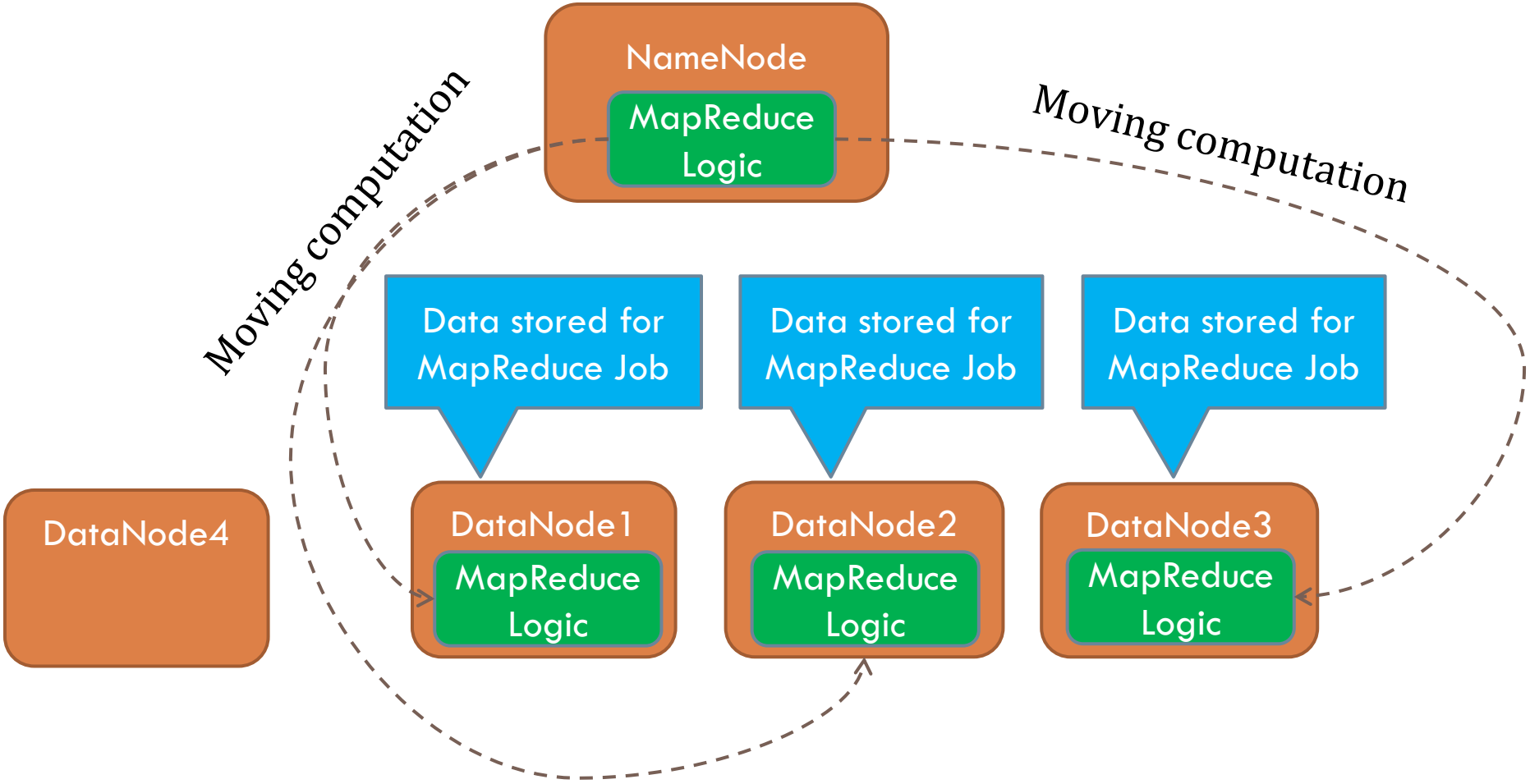
Draw the MapReduce process to find the maximum electrical consumption for each year:

Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Avg
1979	23	23	2	43	24	25	26	26	26	26	25	26	25
1980	26	27	28	28	28	30	31	31	31	30	30	30	29
1981	31	32	32	32	33	34	35	36	36	34	34	34	34
1984	39	38	39	39	39	41	42	43	40	39	38	38	40
1985	38	39	39	39	39	41	41	41	00	40	39	39	45

Data Locality

When a dataset is stored in HDFS, it is divided in to blocks and stored across the DataNodes in the Hadoop cluster. When a MapReduce job is executed against the dataset the individual Mappers will process the blocks (input splits). When the data is not available for the Mapper in the same node where it is being executed, the data needs to be copied over the network from the DataNode which has the data to the DataNode which is executing the Mapper task. Imagine a MapReduce job with over 100 Mappers and each Mapper is trying to copy the data from another DataNode in the cluster at the same time, this would result in serious network congestion as all the Mappers would try to copy the data at the same time and it is not ideal. So it is always effective and cheap to move the computation closer to the data than to move the data closer to the computation. When the data is located on the same node as the Mapper working on the data, it is referred to as Data Local. In this case the proximity of the data is closer to the computation. The ApplicationMaster (MRv2) prefers the node which has the data that is needed by the Mapper to execute the Mapper.

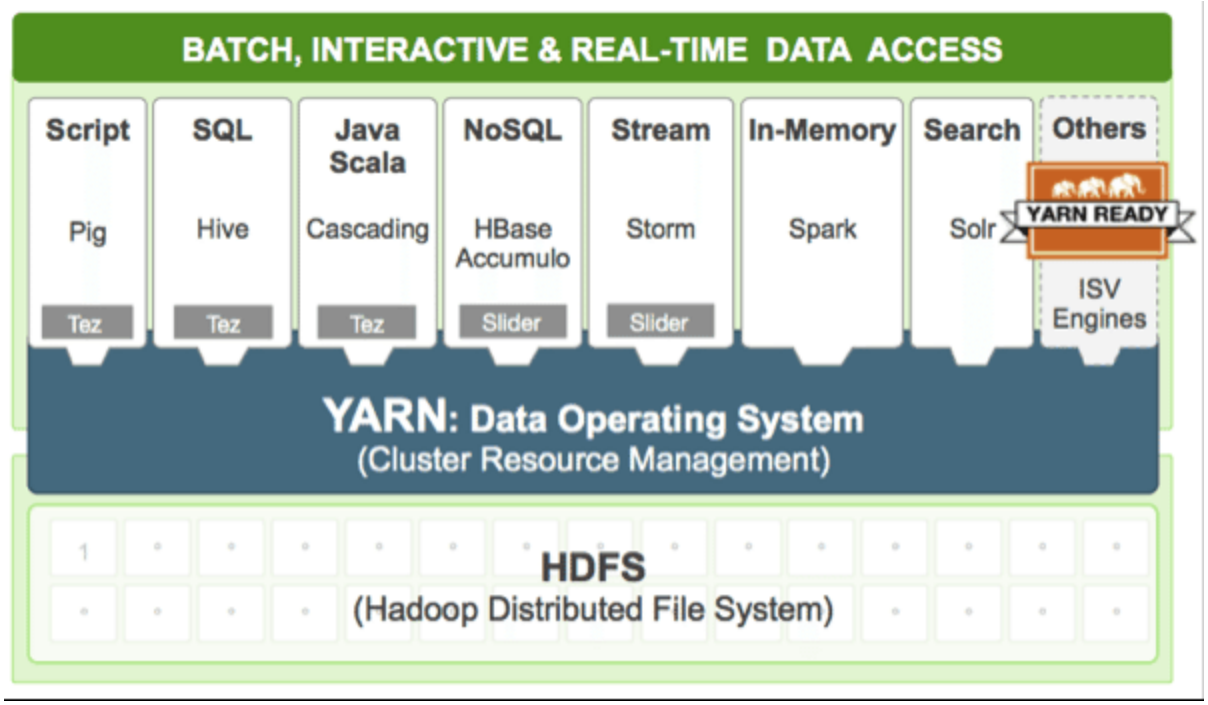
Data Locality cont'd



Apache Hadoop YARN

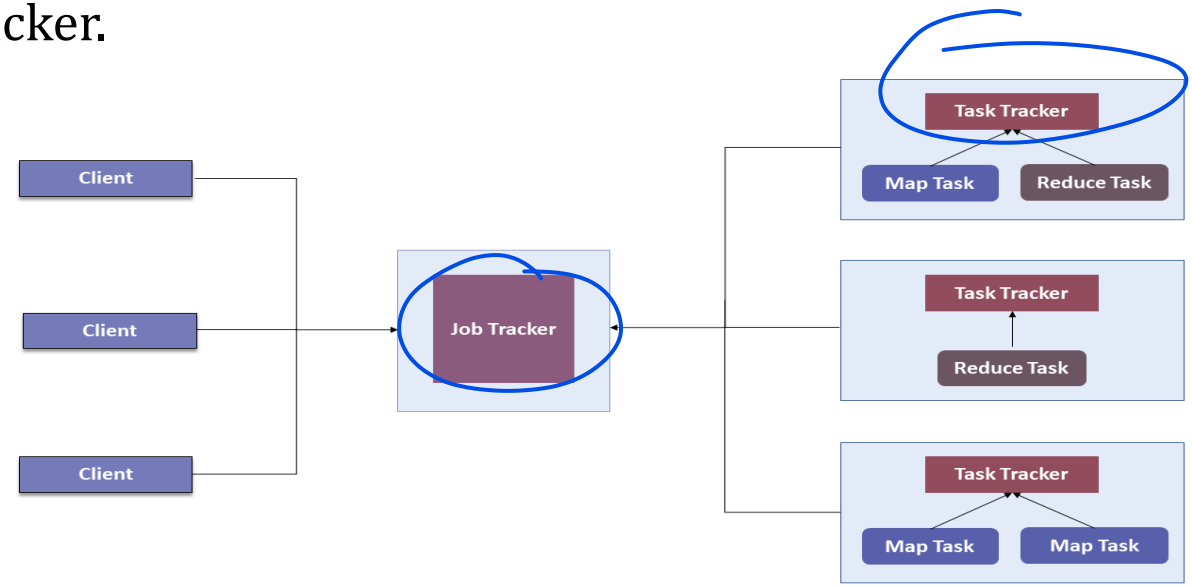
YARN

YARN stands for "Yet Another Resource Negotiator" and is the architectural center of Hadoop 2.0 that **allows multiple data processing engines** such as **interactive SQL**, **real-time streaming**, **data science** and **batch processing** to handle data stored in a single platform, unlocking an entirely new approach to analytics.



Why YARN?

In Hadoop 1.0 which is also referred to as MRV1(MapReduce Version 1), MapReduce performed both processing and resource management functions. It consisted of a Job Tracker which was the single master. The Job Tracker allocated the resources, performed scheduling and monitored the processing jobs. It assigned map and reduce tasks on a number of subordinate processes called the Task Trackers. The Task Trackers periodically reported their progress to the Job Tracker.



Why YARN cont'd

This design resulted in **scalability bottleneck** due to a single Job Tracker. IBM mentioned in its article that according to Yahoo!, the practical limits of such a design are reached with a cluster of 5000 nodes and 40,000 tasks running concurrently. Apart from this limitation, the utilization of computational resources is inefficient in MRV1. Also, the Hadoop framework became limited only to MapReduce processing paradigm.

To overcome all these issues, YARN was introduced in Hadoop version 2.0 in the year 2012 by Yahoo and Hortonworks. The basic idea behind YARN is to relieve MapReduce by taking over the responsibility of Resource Management and Job Scheduling. YARN started to give Hadoop the ability to run non-MapReduce jobs within the Hadoop framework.

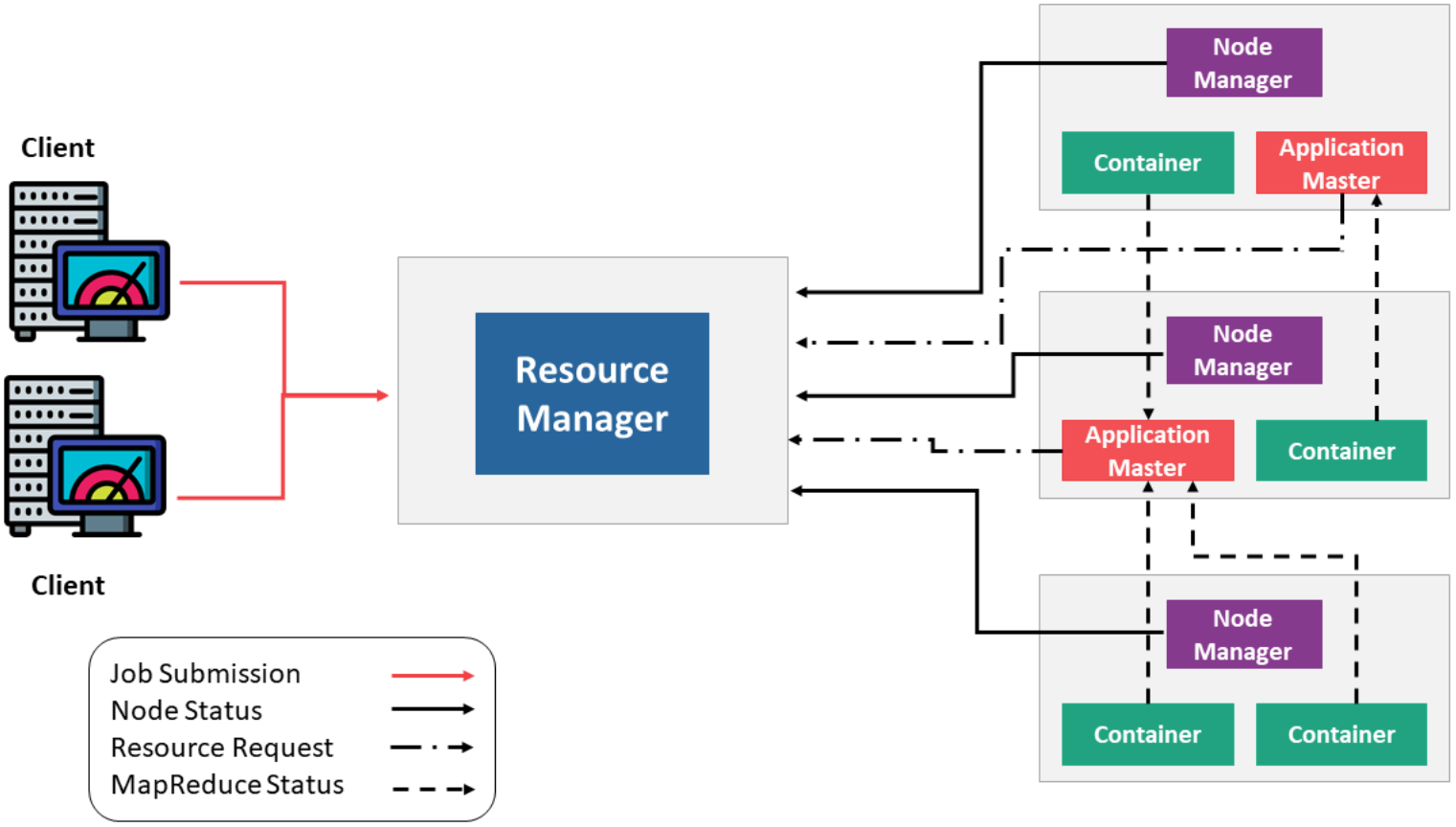
With the introduction of YARN, the Hadoop ecosystem was completely revolutionalized. It became much more flexible, efficient and scalable. When Yahoo went live with YARN in the first quarter of 2013, it aided the company to shrink the size of its Hadoop cluster from 40,000 nodes to 32,000 nodes.

Introduction to YARN

YARN allows different data processing methods like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS. Therefore YARN opens up Hadoop to other types of distributed applications beyond MapReduce. YARN enabled the users to perform operations as per requirement by using a variety of tools like Spark for real-time processing, Hive for SQL, HBase for NoSQL and others. Apart from Resource Management, YARN also performs Job Scheduling. YARN performs all your processing activities by allocating resources and scheduling tasks. Apache Hadoop YARN architecture consists of the following main components :

- **Resource Manager:** Runs on a master daemon and manages the resource allocation in the cluster.
- **Node Manager:** They run on the slave daemons and are responsible for the execution of a task on every single Data Node.
- **Application Master:** Manages the user job lifecycle and resource needs of individual applications. It works along with the Node Manager and monitors the execution of tasks.
- **Container:** Package of resources including RAM, CPU, Network, HDD etc on a single node.

Components of YARN



Resource Manager

- It is the ultimate authority in resource allocation.
- On receiving the processing requests, it passes parts of requests to corresponding node managers accordingly, where the actual processing takes place.
- It is the arbitrator of the cluster resources and decides the allocation of the available resources for competing applications.
- Optimizes the cluster utilization like keeping all resources in use all the time against various constraints such as capacity guarantees, fairness, and SLAs.
- It has two major components: a) Scheduler b) Application Manager
  - **Scheduler**: It is responsible for allocating resources to the various running applications subject to constraints of capacities, queues etc and does not perform any monitoring or tracking of status for the applications. If there is an application failure or hardware failure, the Scheduler does not guarantee to restart the failed tasks.
  - **Application Manager:** It is responsible for accepting job submissions. Negotiates the first container from the Resource Manager for executing the application specific Application Master. Manages running the Application Masters in a cluster and provides service for restarting the Application Master container on failure.

Node Manager

- It takes care of individual nodes in a Hadoop cluster and manages user jobs and workflow on the given node.
- It registers with the Resource Manager and sends heartbeats with the health status of the node.
- Its primary goal is to manage application containers assigned to it by the resource manager.
- It keeps up-to-date with the Resource Manager.
- Application Master requests the assigned container from the Node Manager by sending it a Container Launch Context(CLC) which includes everything the application needs in order to run. The Node Manager creates the requested container process and starts it.
- Monitors resource usage (memory, CPU) of individual containers.
- Performs Log management.
- It also kills the container as directed by the Resource Manager.

Application Master

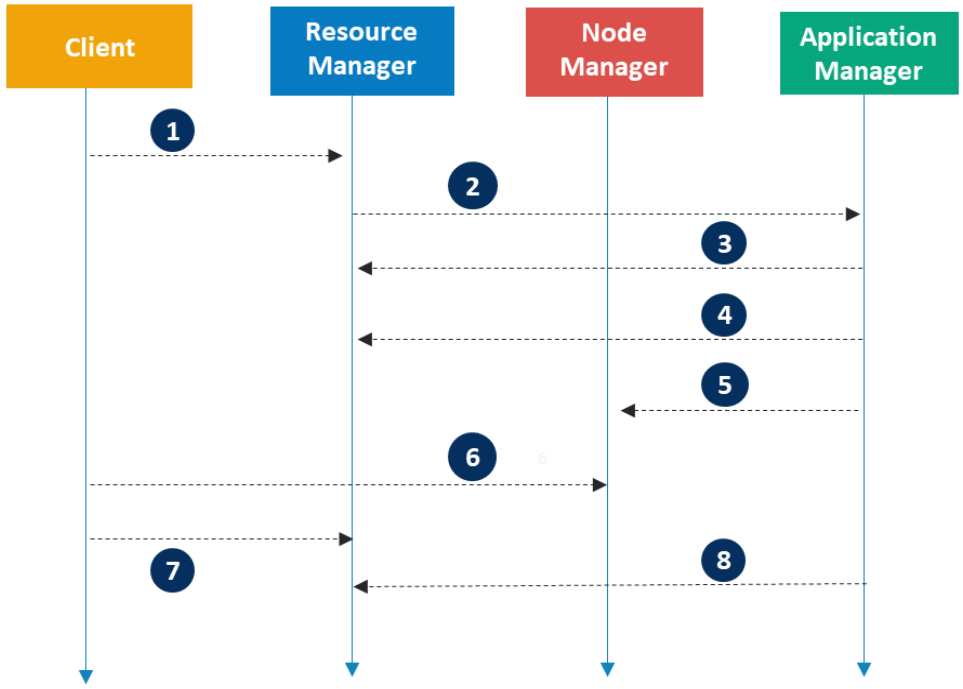
- An application is a single job submitted to the framework. Each such application has a unique Application Master associated with it which is a framework specific entity.
- It is the process that coordinates an application's execution in the cluster and also manages faults.
- Its task is to negotiate resources from the Resource Manager and work with the Node Manager to execute and monitor the component tasks.
- It is responsible for negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring progress.
- Once started, it periodically sends heartbeats to the Resource Manager to affirm its health and to update the record of its resource demands.

Container

- It is a collection of physical resources such as RAM, CPU cores, and disks on a single node.
- YARN containers are managed by a container launch context which is container life-cycle(CLC). This record contains a map of environment variables, dependencies stored in a remotely accessible storage, security tokens, payload for Node Manager services and the command necessary to create the process.
- It grants rights to an application to use a specific amount of resources (memory, CPU etc.) on a specific host.

Application Workflow

1. Client submits an application
2. Resource Manager allocates a container to start Application Manager
3. Application Manager registers with Resource Manager
4. Application Manager asks containers from Resource Manager
5. Application Manager notifies Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application's status
8. Application Manager unregisters with Resource Manager



Apache Pig

Pig

- Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with Hadoop and perform all the data manipulation operations.
- To write data analysis programs, Pig provides a high-level language known as Pig Latin. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.
- To analyze data using Apache Pig, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as Pig Engine that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Need of Apache Pig

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

- Using Pig Latin, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Apache Pig uses multi-query approach, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.
- Pig Latin is SQL-like language and it is easy to learn Apache Pig when you are familiar with SQL.
- Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

Features of Apache Pig

Apache Pig comes with the following features:

- **Rich set of operators** – It provides many operators to perform operations like join, sort, filer, etc.
- **Ease of programming** – Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
- **Optimization opportunities** – The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.
- **Extensibility** – Using the existing operators, users can develop their own functions to read, process, and write data.
- **UDF's** – Pig provides the facility to create User-defined Functions in other programming languages such as Java and invoke or embed them in Pig Scripts.
- **Handles all kinds of data** – Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

Apache Pig vs. MapReduce

Apache Pig	MapReduce
Apache Pig is a data flow language	MapReduce is a data processing paradigm.
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 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.

Application and History of Apache Pig

Application

Apache Pig is generally used by data scientists for performing tasks involving ad-hoc processing and quick prototyping. Apache Pig is used:

- To process huge data sources such as web logs.
- To perform data processing for search platforms.
- To process time sensitive data loads.

History

In 2006, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset. In 2007, Apache Pig was open sourced via Apache incubator. In 2008, the first release of Apache Pig came out. In 2010, Apache Pig graduated as an Apache top-level project.

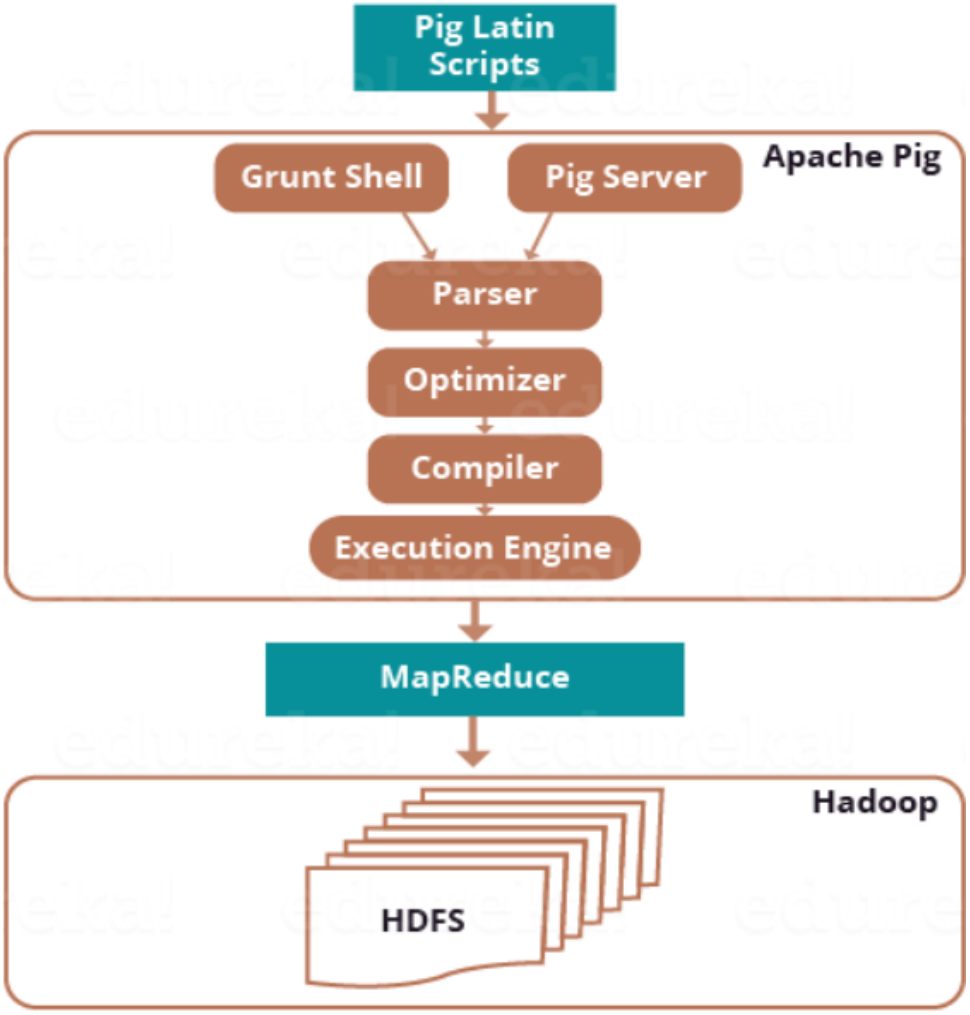
Apache Pig Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a high level 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.

Apache Pig Architecture cont'd



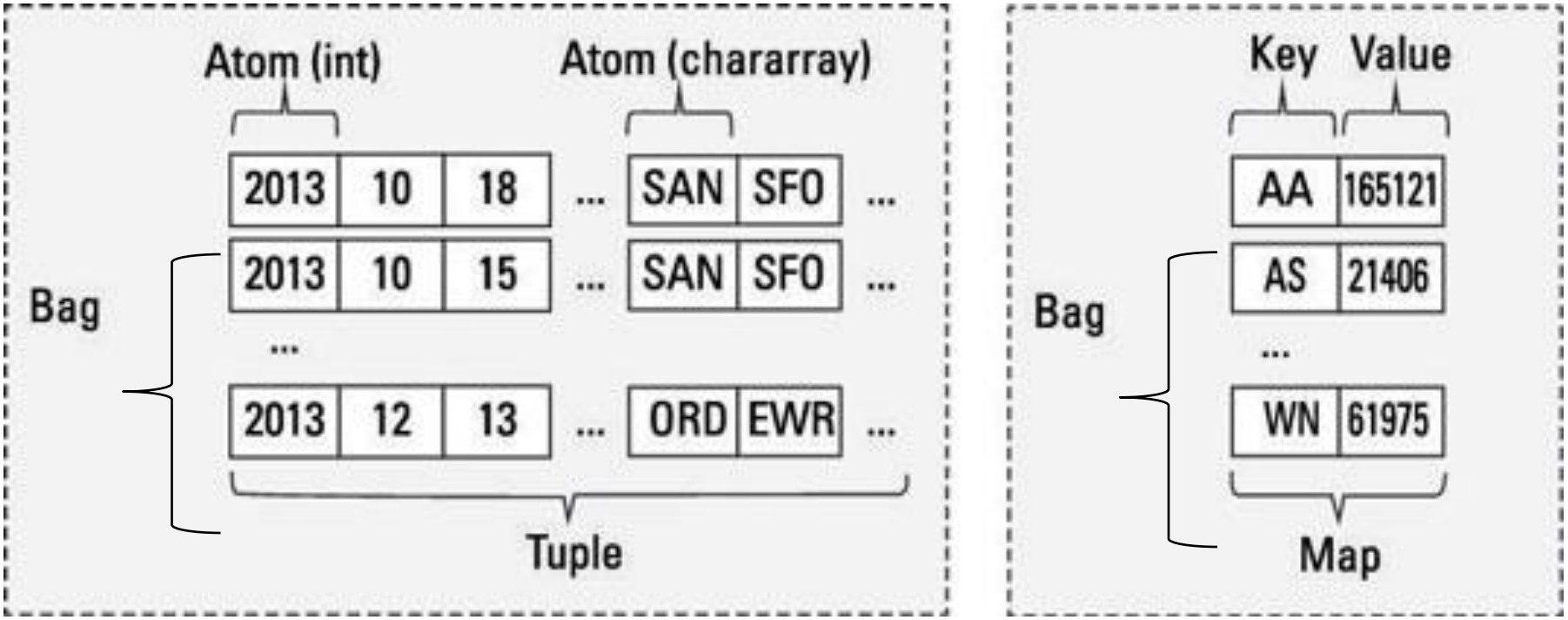
Apache Pig Architecture Components

- **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.

Apache Pig 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. Ex: 'Abhi'
- **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. Ex: ('Abhi', 14)
- **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. Ex: {('Abhi'), ('Manu', (14, 21))}
- **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 '[]'. Ex: ['name'#'Raju', '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 Data Model cont'd



Data Types: Int, Long, Float, Double, charArray, byteArray, tuple, bag, map (collection of tuples)

Apache Pig Latin Execution Modes

Apache Pig can run in two modes, namely, Local Mode and HDFS mode.

- **Local Mode:** In this mode, all the files are installed and run from local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.
- **MapReduce Mode:** MapReduce mode is where the data loaded or processed that exists in the Hadoop File System (HDFS). In this mode, whenever Pig Latin statements are executed to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

Apache Pig Latin Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode (Grunt shell)** – You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (using Dump operator).
- **Batch Mode (Script)** – You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with .pig extension.
- **Embedded Mode (UDF)** – Apache Pig provides the provision of defining our own functions (User Defined Functions) in programming languages such as Java, and using them in our script.

Invoking the Grunt Shell

Grunt shell can be invoked in a desired mode (local/MapReduce) using the –x option as shown below.

Local mode	MapReduce mode
Command – $ ./pig –x local	**Command –** $ ./pig -x mapreduce
Output – 15/09/28 10:13:03 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443415383991.log 2015-09-28 10:13:04,838 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:/// grunt>	**Output –** 15/09/28 10:28:46 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443416326123.log 2015-09-28 10:28:46,427 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:/// grunt>

Operators

Operator	Description
Loading and Storing	
LOAD	To Load the data from the file system (local/HDFS) into a relation.
STORE	To save a relation to the file system (local/HDFS).
Filtering	
FILTER	To remove unwanted rows from a relation.
DISTINCT	To remove duplicate rows from a relation.
Diagnostic Operators	
DUMP	To print the contents of a relation on the console.
DESCRIBE	To describe the schema of a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more relations.

Operators cont'd

Operator	Description
Grouping and Joining	
JOIN	To join two or more relations.
COGROUP	To group the data in two or more relations.
GROUP	To group the data in a single relation.
Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.

Apache Pig – Reading Data

To analyze data using Apache Pig, the data has to be loaded into Apache Pig. In MapReduce mode, Pig reads (loads) data from HDFS and stores the results back in HDFS. The below dataset contains personal details like id, first name, last name, phone number and city, of six students and stored in student_data.txt with csv format in Hadoop cluster - hdfs://localhost:9000/pig_data/ path.

Student ID	First Name	Last Name	Phone	City
001	Rajiv	Reddy	9848022337	Hyderabad
002	siddarth	Battacharya	9848022338	Kolkata
003	Rajesh	Khanna	9848022339	Delhi
004	Preethi	Agarwal	9848022330	Pune
005	Trupthi	Mohanthy	9848022336	Bhuwaneshwar
006	Archana	Mishra	9848022335	Chennai

Apache Pig – Reading Data cont'd

The data to be loaded into Apache Pig from the file system (HDFS/Local) using LOAD operator of Pig Latin.

Syntax: Relation_name = LOAD 'Input file path' USING function as schema where:

- **relation_name –** the relation in which we want to store the data.
- **Input file path –** HDFS directory where the file is stored (MapReduce mode)
- **function –** function from the set of load functions provided by Apache Pig (BinStorage, JsonLoader, PigStorage, TextLoader).
- **Schema –** define the schema of the data. The required schema can be defined as – (column1 : data type, column2 : data type);

Example:

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt'
       USING PigStorage(',') as ( id:int, firstname:chararray, lastname:chararray,
       phone:chararray,  city:chararray );
```

Apache Pig – Storing Data

The loaded data can be stored in the HDFS file system using the store operator.

Syntax: STORE relation_name INTO ' required_directory_path ' [USING function];

- **relation_name –** the relation in which we want to store the data.
- **Input file path –** HDFS directory where the file is stored (MapReduce mode)
- **function –** function from the set of load functions provided by Apache Pig (BinStorage, JsonLoader, PigStorage, TextLoader).

Example:

```
grunt> STORE student INTO ' hdfs://localhost:9000/pig_Output/' USING
        PigStorage (',');
```

This would store the relation in the HDFS directory "/pig_Output/" wherein student is the relation as explained in LOAD operator.

Apache Pig – Diagnostic Operators

- **Dump Operator:** It is used to run the Pig Latin statements and display the results on the screen. It is generally used for debugging Purpose.
  Syntax: grunt> Dump Relation_Name
  Example: grunt> Dump student → Assume data is read into a relation student using the LOAD operator
- **Describe Operator:** It is used to view the schema of a relation.
  Syntax: grunt> Describe Relation_Name
  Example: grunt> Describe student → Assume data is read into a relation student using the LOAD operator
- **Explain Operator:** It is used to display the logical, physical, and MapReduce execution plans of a relation
  Syntax: grunt> explain Relation_Name
  Example: grunt> explain student → Assume data is read into a relation student using the LOAD operator

Apache Pig – Group Operators

- **Group Operator:** It is used to group the data in one or more relations. It collects the data having the same key.
  Example: grunt> group_data = GROUP student by age; → Assume data is read into a relation student using the LOAD operator
  Example: grunt> group_data = GROUP student by (age, city); → Assume data is read into a relation student using the LOAD operator and it's the illustration of multiple columns grouping
- **Group ALL Operator:** A relation can be grouped by all the columns and is shown below.
  Example: grunt> group_all = GROUP student_details All;
- **Cogroup Operator:** The only difference between the Group and Cogroup operators is that the group operator is normally used with one relation, while the cogroup operator is used in statements involving two or more relations.

Apache Pig – Cogroup Operator Example


```
grunt> student_details = LOAD
'hdfs://localhost:9000/pig_data/student_details.txt' USING PigStorage(',')
as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray,
city:chararray);

grunt> employee_details = LOAD
'hdfs://localhost:9000/pig_data/employee_details.txt' USING PigStorage(',')
 as (id:int, name:chararray, age:int, city:chararray);
```

Now, let us group the records/tuples of the relations student_details and employee_details with the key age, as shown below.

```
grunt> cogroup_data = COGROUP student_details by age, employee_details by
age;
```

The cogroup operator groups the tuples from each relation according to age where each group depicts a particular age value.

Apache Pig – Join Operator

The JOIN operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two particular tuples are matched, else the records are dropped. Joins can be of the following types –

- **Self-join:** It is used to join a table with itself as if the table were two relations, temporarily renaming at least one relation.
- **Inner-join:** returns rows when there is a match in both tables.
- **Outer-join:**
  - Left outer join: returns all rows from the left table, even if there are no matches in the right relation.
  - Right outer join: returns all rows from the right table, even if there are no matches in the left table.
  - Full outer join: returns rows when there is a match in one of the relations.

Apache Pig – Self-Join Operator Example

Self-join: Generally, in Apache Pig, to perform self-join, we will load the same data multiple times, under different aliases (names). Therefore let us load the contents of the file customers.txt as two tables as shown below.

```
grunt> customers1 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt'
USING PigStorage(',') as (id:int, name:chararray, age:int, address:chararray,
salary:int);
```

```
grunt> customers2 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt'
USING PigStorage(',') as (id:int, name:chararray, age:int, address:chararray,
salary:int);
```

Example: Let us perform self-join operation on the relation customers, by joining the two relations customers1 and customers2 as shown below.

```
grunt> customers3 = JOIN customers1 BY id, customers2 BY id;
```

Apache Pig – Inner Join Operator Example

Inner-join: It creates a new relation by combining column values of two relations (say A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, the column values for each matched pair of rows of A and B are combined into a result row.

grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING PigStorage(',') as (id:int, name:chararray, age:int, address:chararray, salary:int);

grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING PigStorage(',') as (oid:int, date:chararray, customer_id:int, amount:int);

Example: Let us perform inner join operation on the two relations customers and orders as shown below.

grunt> coustomer_orders = JOIN customers BY id, orders BY customer_id;

Apache Pig – Outer Join Operator Example

grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING PigStorage(',') as (id:int, name:chararray, age:int, address:chararray, salary:int);

grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING PigStorage(',') as (oid:int, date:chararray, customer_id:int, amount:int);

Example :

grunt> outer_left = JOIN customers BY id **LEFT OUTER**, orders BY customer_id;

grunt> outer_right = JOIN customers BY id **RIGHT OUTER**, orders BY customer_id;

grunt> outer_full = JOIN customers BY id **FULL OUTER**, orders BY customer_id;

Apache Pig – Cross Operator

The CROSS operator computes the cross-product of two or more relations.

Example:

```
grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING PigStorage(',') as (id:int, name:chararray, age:int, address:chararray, salary:int);

grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING PigStorage(',') as (oid:int, date:chararray, customer_id:int, amount:int);

grunt> cross_data = CROSS customers, orders;

grunt> Dump cross_data;
```

Apache Pig – Union Operator

The UNION operator of Pig Latin is used to merge the content of two relations. To perform UNION operation on two relations, their columns and domains must be identical.

Example:

```
grunt> student1 = LOAD 'hdfs://localhost:9000/pig_data/student_data1.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, phone:chararray, city:chararray);

grunt> student2 = LOAD 'hdfs://localhost:9000/pig_data/student_data2.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, phone:chararray, city:chararray);

grunt> student = UNION student1, student2;

grunt> Dump student;
```

Apache Pig – Split Operator

The SPLIT operator is used to split a relation into two or more relations.

Example:

grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);

Let us now split the relation into two, one listing the students of age less than 23, and the other listing the students having the age between 22 and 25.

grunt> SPLIT student into student1 if age<23, student2 if (22<age and age>25);

grunt> Dump student1;

grunt> Dump student2;

Apache Pig – Distinct Operator

The DISTINCT operator is used to remove redundant (duplicate) tuples from a relation.

Syntax:

grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);

Assume there are some duplicate tuples in the relation

grunt> distinct_data = DISTINCT student;

grunt> Dump distinct_data ;

Apache Pig – Foreach Operator

The FOREACH operator is used to generate specified data transformations based on the column data.

Example:

grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);

Assume there are 8 tuples in the relation and get the id, age, and city values of each student from the relation student_details and store it into another relation named foreach_data

grunt> foreach_data = FOREACH student_details GENERATE id,age,city;

grunt> Dump foreach_data ;

Apache Pig – Order By Operator

The ORDER BY operator is used to display the contents of a relation in a sorted order based on one or more fields.

Example:

grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);

Sort the relation in a descending order based on the age of the student and store it into another relation named order_by_data

grunt> order_by_data = ORDER student_details BY age DESC;

grunt> Dump order_by_data ;

Apache Pig – LIMIT Operator

The LIMIT operator is used to get a limited number of tuples from a relation.

Example:

grunt> student_details = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);

Sort the relation in a descending order based on the age of the student and store it into another relation named order_by_data and then limit the data to 4 and then store it into another relation named limit_data

grunt> order_by_data = ORDER student_details BY age DESC;

grunt> limit_data = LIMIT order_by_data 4;

grunt> Dump limit_data ;

Apache Hive

Hive

Apache Hive is a data warehouse system built on top of Hadoop and is used for analyzing structured and semi-structured data. Hive abstracts the complexity of Hadoop MapReduce. Basically, it provides a mechanism to project structure onto the data and perform queries written in HQL (Hive Query Language) that are similar to SQL statements. Internally, these queries or HQL gets converted to map reduce jobs by the Hive compiler. Therefore, one don't need to worry about writing complex MapReduce programs to process data using Hadoop. It is targeted towards users who are comfortable with SQL. Apache Hive supports Data Definition Language (DDL), Data Manipulation Language (DML) and Pluggable Functions.

DDL: create table, create index, create views.

DML: Select, Where, group by, Join, Order By

Pluggable Functions:

UDF: User Defined Function

UDAF: User Defined Aggregate Function

UDTF: User Defined Table Function

Hive Data Types

Numeric Types

- TINYINT (1-byte signed integer, from -128 to 127)
- SMALLINT (2-byte signed integer, from -32,768 to 32,767)
- INT (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
- BIGINT (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- FLOAT (4-byte single precision floating point number)
- DOUBLE (8-byte double precision floating point number)
- DECIMAL (Hive 0.13.0 introduced user definable precision and scale)

Date/Time Types

- TIMESTAMP
- DATE

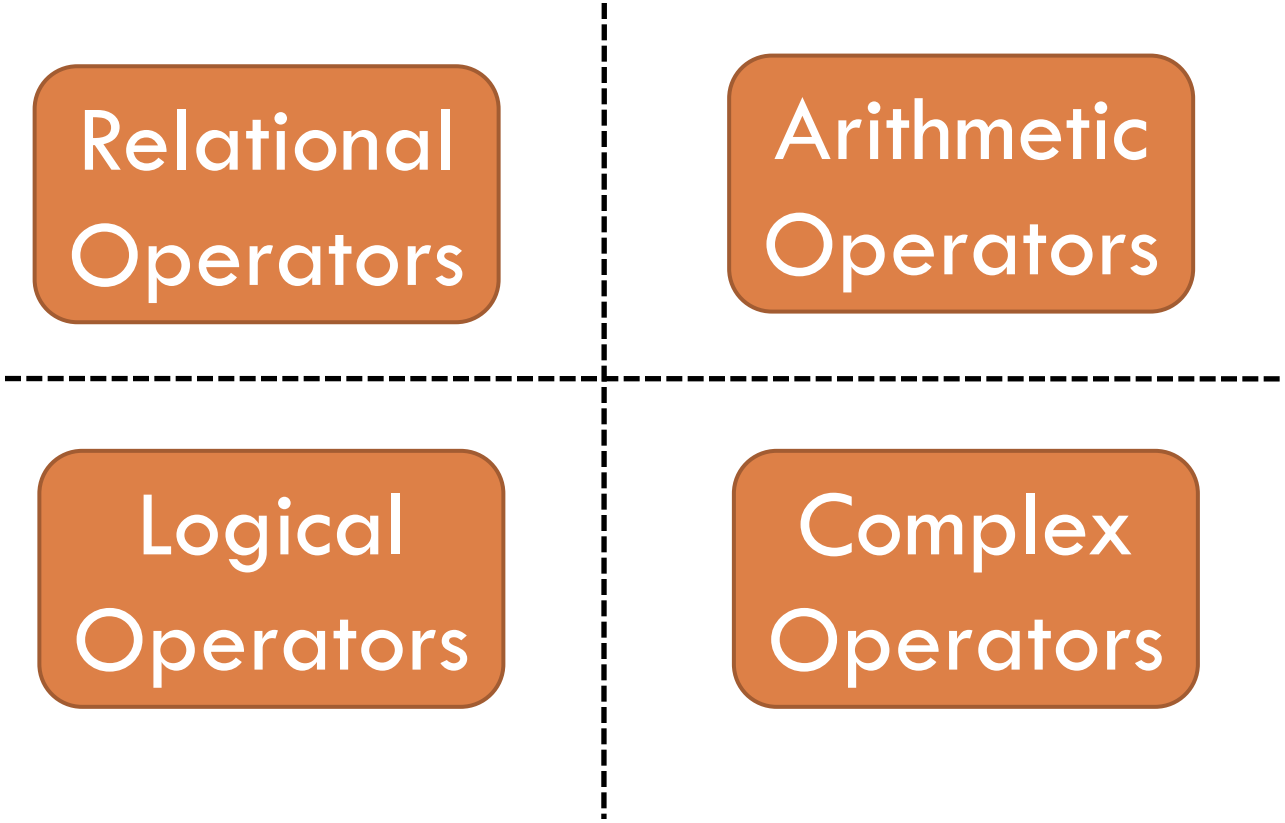
String Types

- STRING
- CHAR
- VARCHAR

Misc Types

- BOOLEAN
- BINARY

Self Study – Built-in Operators



Hive DDL Commands

Create Database Statement: A database in Hive is a namespace or a collection of tables. The syntax for this statement is: CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>. IF NOT EXISTS is an optional clause, which notifies the user that a database with the same name already exists. We can use SCHEMA in place of DATABASE in this command.

- hive> CREATE SCHEMA userdb; → Create the database by name userdb
- hive> SHOW DATABASES; → verify a databases list

Drop Database Statement: Drop Database is a statement that drops all the tables and deletes the database. Its syntax is: DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];

- hive> DROP DATABASE IF EXISTS userdb; → drop a database by name userdb
- hive> DROP DATABASE IF EXISTS userdb; → drops the database using CASCADE i.e. dropping respective tables before dropping the database.

Hive DDL Commands cont'd

Create Table Statement: Create Table is a statement used to create a table in Hive. Syntax:

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[ROW FORMAT row_format]
[STORED AS file_format]

Example: Let us assume to create a table named employee. The following table lists the fields and their data types in employee table:

Sr. No	Field Name	Data Type
1	EID	int
2	Name	String
3	Salary	Float

Create Table Statement

The following data is a Comment, Row formatted fields such as Field terminator, Lines terminator, and Stored File type.

```
COMMENT 'Employee details'
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED IN TEXT FILE
```

The following query creates a table named employee using the above data.

```
hive> CREATE TABLE IF NOT EXISTS employee ( eid int, name String,
salary String)
COMMENT 'Employee details'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Load Data Statement

Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement. While inserting data into Hive, it is better to use LOAD DATA to store bulk records. There are two ways to load data: one is from local file system and second is from Hadoop file system.

Syntax: The syntax for load data is as follows:

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

- LOCAL is identifier to specify the local path. It is optional.
- OVERWRITE is optional to overwrite the data in the table.
- PARTITION is optional.

Load Data Statement Example

Let's insert the following data into the table. It is a text file named sample.txt in /home/user directory.

1201	Gopal	45000	Technical manager
1202	Manisha	45000	Proof reader
1203	Masthanvali	40000	Technical writer
1204	Kiran	40000	Hr Admin
1205	Kranthi	30000	Op Admin

The following query loads the given text into the table.

```
hive> LOAD DATA LOCAL INPATH '/home/user/sample.txt'
OVERWRITE INTO TABLE employee;
```

Alter Table

It is used to alter a table in Hive.

Syntax: The statement takes any of the following syntaxes based on what attributes we wish to modify in a table.

ALTER TABLE name RENAME TO new_name
ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name new_name new_type
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

Example:

- hive> ALTER TABLE employee RENAME TO emp; → Rename the table from employee to emp
- ALTER TABLE employee CHANGE name ename String; → Rename the column from name to ename
- hive> ALTER TABLE employee ADD COLUMNS (dept STRING COMMENT 'Department name'); → add a column named dept to the employee table

Hive Partitioning

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into buckets, to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named Tab1 contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time.

Hive Partitioning Example

The following file contains employeedata table.

/tab1/employeedata/file1

id, name, dept, yoj
1, gopal, TP, 2012
2, kiran, HR, 2012
3, kaleel,SC, 2013
4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

1, gopal, TP, 2012
2, kiran, HR, 2012

/tab1/employeedata/2013/file3

3, kaleel,SC, 2013
4, Prasanth, SC, 2013

Hive QL

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore.

Select-Where Statement

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Select-Where Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table, with fields named Id, Name, Salary, Designation, and Dept.

Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

```
hive> SELECT * FROM employee WHERE salary>30000;
```

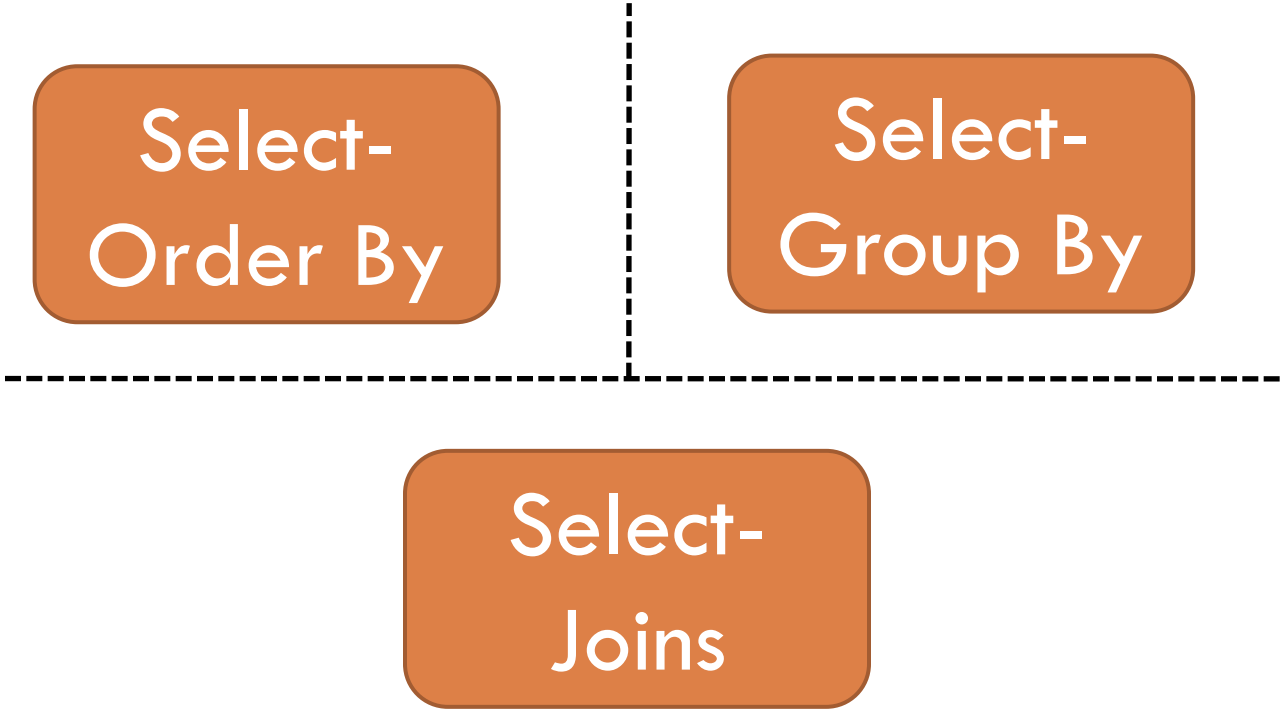
Generate a query to retrieve the employee details who belongs to IT dept.

```
hive> SELECT * FROM employee WHERE dept='IT';
```

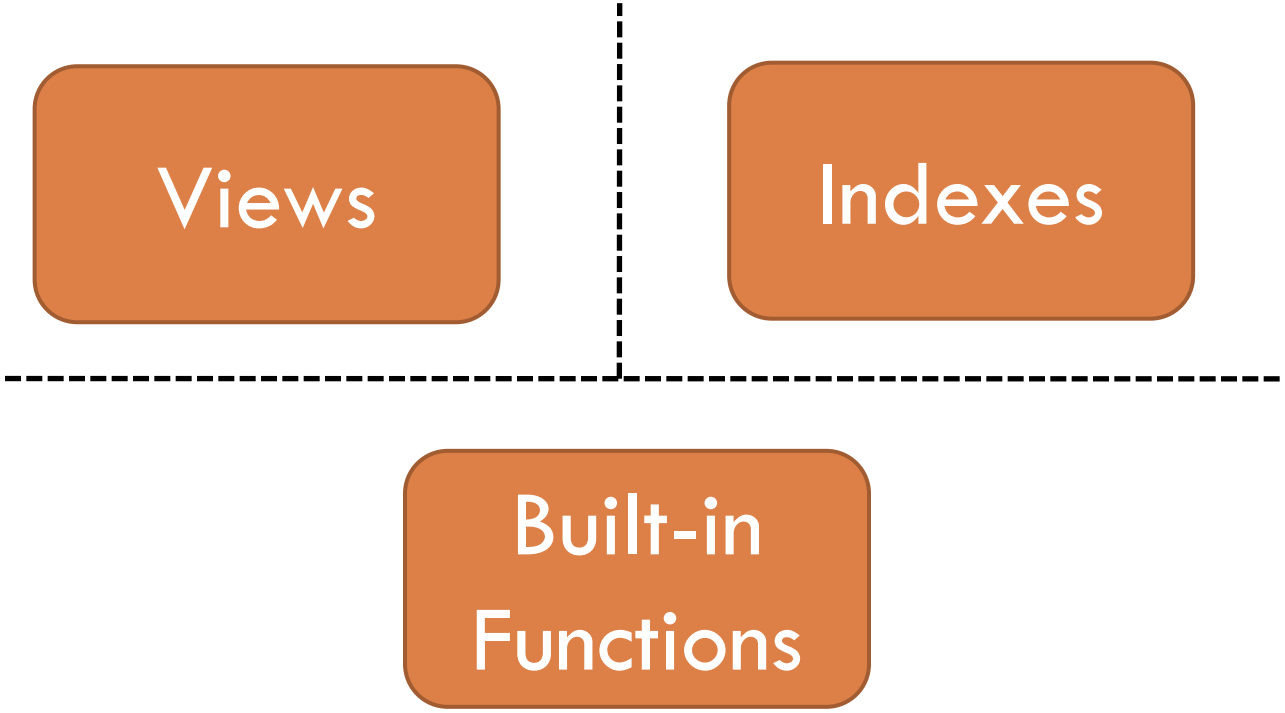
Generate a query to retrieve the employee details who belongs to IT dept and drawing more than Rs 30000..

```
hive> SELECT * FROM employee WHERE dept='IT' AND salary>30000;
```

Self Study – Hive QL



Self Study



Apache Scoop

Scoop

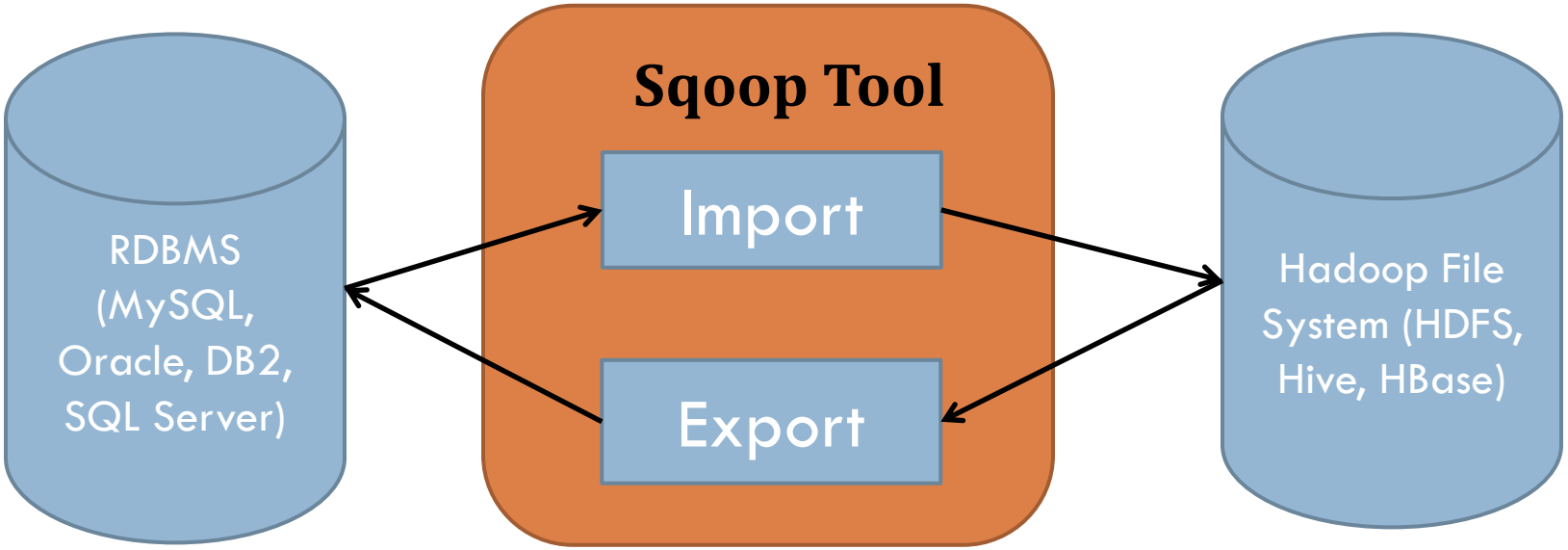
Sqoop is a tool designed to transfer data between Hadoop and relational database servers. It is used to import data from relational databases such as MySQL, Oracle to Hadoop HDFS, and export from Hadoop file system to relational databases.

The traditional application management system, that is, the interaction of applications with relational database using RDBMS, is one of the sources that generate Big Data. Such Big Data, generated by RDBMS, is stored in Relational Database Servers in the relational database structure.

When Big Data storages and analyzers such as MapReduce, Hive, Pig, etc. of the Hadoop ecosystem came into picture, they required a tool to interact with the relational database servers for importing and exporting the Big Data residing in them. Here, Sqoop occupies a place in the Hadoop ecosystem to provide feasible interaction between relational database server and Hadoop's HDFS.

Sqoop – "SQL to Hadoop and Hadoop to SQL"

How Sqoop Works?



Sqoop Import: The import tool imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS. All records are stored as text data in text files or as binary data in Avro and Sequence files.

Sqoop Export: The export tool exports a set of files from HDFS back to an RDBMS. The files given as input to Sqoop contain records, which are called as rows in table. Those are read and parsed into a set of records and delimited with user-specified delimiter.

Sqoop - Import

The 'Import tool' imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS. All records are stored as text data in the text files or as binary data in Avro and Sequence files.

Syntax: $ sqoop import (generic-args) (import-args)
$ sqoop-import (generic args) (import args) → alternative approach

The Hadoop specific generic arguments must precede any import arguments, and the import arguments can be of any order.

Importing a Table into HDFS

$ sqoop import --connect --table --username --password --target-dir

--connect Takes JDBC url and connects to database
--table Source table name to be imported
--username Username to connect to database
--password Password of the connecting user
--target-dir Imports data to the specified directory

Sqoop – Import cont'd

Importing Selected Data from Table

$ sqoop import --connect --table --username --password --columns --where

--columns Selects subset of columns
--where Retrieves the data which satisfies the condition

Importing Data from Query

$ sqoop import --connect --table --username --password --query

--query Executes the SQL query provided and imports the results

Importing Data from Query

$ sqoop import --connect --table --username --password --incremental <mode> --check-column <column name> --last-value <last check column value>

Sqoop import supports two types of incremental imports mode namely Append, and Lastmodified.

Sqoop – Import Example

Let us take an example of three tables named as emp (ID, Name, Des, Dalary, Dept), emp_address (ID, House_No, Street, City, Pincode), and emp_contact (ID, Phone_No, Email), which are in a database called userdb in a MySQL database server.

Importing a Table into HDFS:

```
$ sqoop import -- connect jdbc:mysql://localhost/userdb \
--username root  --table emp
```

Importing a Table into Target Directory:

```
$ sqoop import -- connect jdbc:mysql://localhost/userdb \
--username root  --table emp_address --target-dir /emp_addr
```

Import Subset of Table Data:

```
$ sqoop import -- connect jdbc:mysql://localhost/userdb \
--username root  --table emp_contact --where “Phone_No =’953467890’”
```

Sqoop – Import Example cont'd

Incremental Import :

$ sqoop import --connect jdbc:mysql://localhost/userdb --username root \
--table emp --incremental append --check-column ID --last value 1205

Import All Tables

Syntax:

$ sqoop import-all-tables (generic-args) (import-args)
$ sqoop-import-all-tables (generic-args) (import-args) → alternative approach

Note: If import-all-tables is used, it is mandatory that every table in that database must have a primary key field.

Example:

$ sqoop import-all-tables --connect jdbc:mysql://localhost/userdb \
--username root

Sqoop - Export

The 'Export tool' export data back from the HDFS to the RDBMS database. The target table must exist in the target database. The files which are given as input to the Sqoop contain records, which are called rows in table. Those are read and parsed into a set of records and delimited with user-specified delimiter.

The default operation is to insert all the record from the input files to the database table using the INSERT statement. In update mode, Sqoop generates the UPDATE statement that replaces the existing record into the database.

Syntax: $ sqoop export (generic-args) (import-args)
$ sqoop-export (generic args) (import args) → alternative approach

Example:
Let us take an example of the employee data in file, in HDFS. The employee data is available in emp_data file in 'emp/' directory in HDFS.

It is mandatory that the table to be exported is created manually and is present in the database from where it has to be exported.

Sqoop – Export Example

The following query is to create the table 'employee' in mysql command line.

```
$ mysql
mysql> USE db;
mysql> CREATE TABLE employee (id INT NOT NULL PRIMARY KEY,
  name VARCHAR(20), deg VARCHAR(20), salary INT, dept VARCHAR(10));
```

The following command is used to export the table data (which is in emp_data file on HDFS) to the employee table in db database of Mysql database server.

```
$ sqoop export --connect jdbc:mysql://localhost/db  --username root \
--table employee  --export-dir /emp/emp_data
```

The following command is used to verify the table in mysql command line.

```
mysql>select * from employee;
```

Apache HBase

HBase

HBase is a data model designed to provide quick random access to huge amounts of structured data. It is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable. It leverages the fault tolerance provided by the HDFS. It is a part of the Hadoop ecosystem that provides random real-time read/write access to data

Limitations of Hadoop

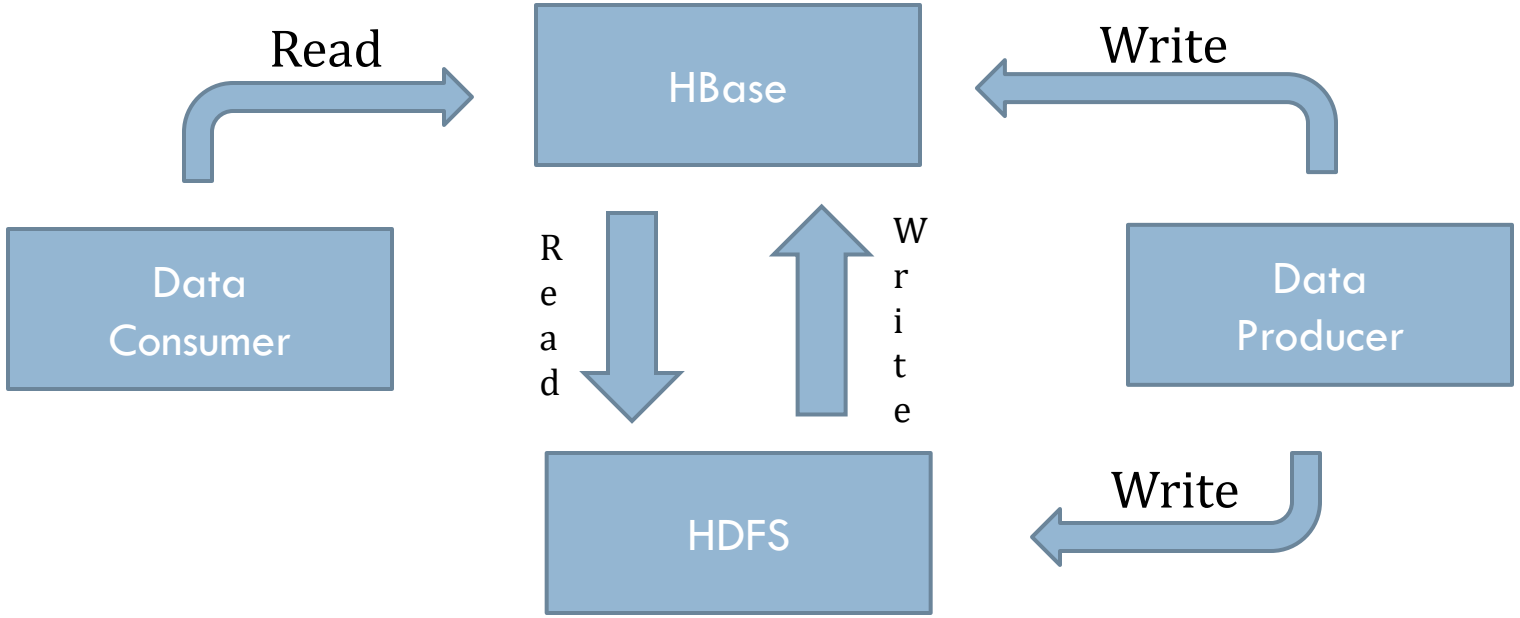
Hadoop accessed data only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs. A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

Hadoop Random Access Databases

Storage 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.

HBase cont'd

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.



HDFS vs. HBase

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.	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

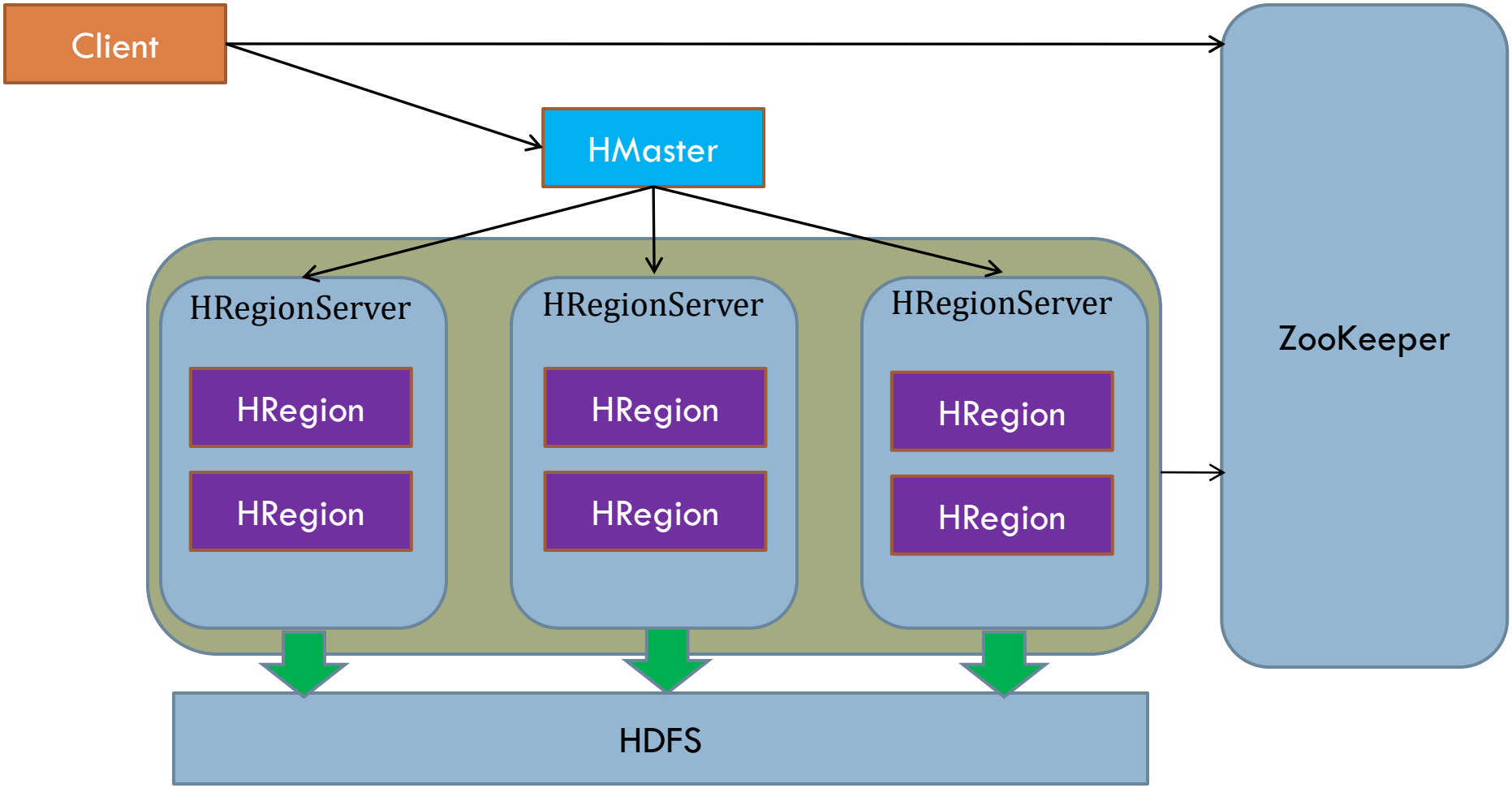
HBase is a column-oriented database and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the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

Example:

Rowid	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

HBase Architecture



HBase Architecture cont'd

HBase architecture consists mainly of four components

- HMaster
- HRegionserver
- HRegions
- Zookeeper
- HDFS

HMaster: HMaster is the implementation of a Master server in HBase architecture. It acts as a monitoring agent to monitor all Region Server instances present in the cluster and acts as an interface for all the metadata changes. In a distributed cluster environment, Master runs on NameNode. Master runs several background threads.

The client communicates in a bi-directional way with both HMaster and ZooKeeper. For read and write operations, it directly contacts with HRegion servers. HMaster assigns regions to region servers and in turn, check the health status of region servers.

HBase Architecture cont'd

HRegions Servers: When Region Server receives writes and read requests from the client, it assigns the request to a specific region, where the actual column family resides. However, the client can directly contact with HRegion servers, there is no need of HMaster mandatory permission to the client regarding communication with HRegion servers. The client requires HMaster help when operations related to metadata and schema changes are required.

HRegionServer is the Region Server implementation. It is responsible for serving and managing regions or data that is present in a distributed cluster. The region servers run on Data Nodes present in the Hadoop cluster.

HRegions: HRegions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families. It contains multiple stores, one for each column family. It consists of mainly two components, which are Memstore and Hfile.

HBase Architecture cont'd

ZooKeeper: In HBase, Zookeeper is a centralized monitoring server which maintains configuration information and provides distributed synchronization. Distributed synchronization is to access the distributed applications running across the cluster with the responsibility of providing coordination services between nodes. If the client wants to communicate with regions, the server's client has to approach ZooKeeper first.

Conclusion: HBase is one of NoSql column-oriented distributed database available in Apache foundation. HBase gives more performance for retrieving fewer records rather than Hadoop or Hive. It's very easy to search for given any input value because it supports indexing, transactions, and updating. Online real-time analytics can be performed using HBase integrated with Hadoop ecosystem. It has an automatic and configurable sharding for datasets or tables and provides restful API's to perform the MapReduce jobs.


THANK
YOU!