## 

**UNIT – II Big data Analytics**

SCHOOL OF COMPUTING

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

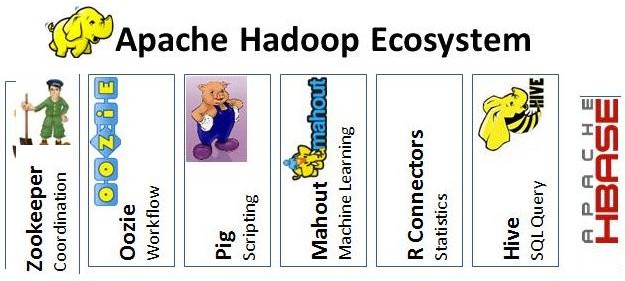
**&**

# DEPARTMENT OF INFORMATION TECHNOLOGY

**Unit II**

BIG DATA TOOLS I 9 Hrs. Big Data Applications using Pig and Hive – Fundamentals of HBase and ZooKeeper – IBM Infosphere Big Insights – Introduction to FLUME – KAFKA.

**Understanding Hadoop and Its Ecosystem**



**Fig:2.1 Apache Hadoop Ecosystem**

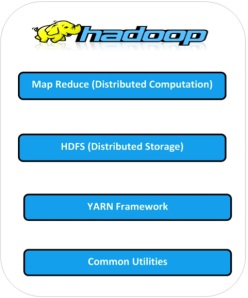
It is quite interesting to envision how we could adopt the Hadoop eco system within the realms of DevOps. I will try to cover it in upcoming series. Hadoop managed by the Apache Foundation is a powerful open-source platform written in java that is capable of processing large amounts of heterogeneous data-sets at scale in a distributive fashion on cluster of computers using simple programming models. It is designed to scale up from single server to thousands of machines, each offering local computation and storage and has become an in-demand technical skill. Hadoop is an Apache top-level project being built and used by a global community of contributors and users.

#### Hadoop Architecture:

The Apache Hadoop framework includes following four modules:

* **Hadoop Common:** Contains Java libraries and utilities needed by other Hadoop modules. These libraries give filesystem and OS level abstraction and comprise of the essential Java files and scripts that are required to start Hadoop.
* **Hadoop Distributed File System (HDFS):** A distributed file-system that provides high-throughput access to application data on the community machines thus providing very high aggregate bandwidth across the cluster.
* **Hadoop YARN:** A resource-management framework responsible for job scheduling and cluster resource management.
* **Hadoop MapReduce:** This is a YARN- based programming model for parallel processing of large data sets.

Below diagram portray four components that are available in Hadoop framework.

F

**Fig:2.2 Apache Hadoop Framework**

All the modules in Hadoop are designed with a fundamental assumption i.e., hardware failure, so should be automatically controlled in software by the framework. Beyond HDFS, YARN and MapReduce, the entire Apache Hadoop “platform” is now commonly

considered to consist of a number of related projects as well: Apache Pig, Apache Hive, Apache HBase, and others.

#### Hadoop Ecosystem:

Hadoop has gained its popularity due to its ability of storing, analyzing and accessing large amount of data, quickly and cost effectively through clusters of commodity hardware. It wont be wrong if we say that Apache Hadoop is actually a collection of several components and not just a single product.

With Hadoop Ecosystem there are several commercial along with an open source products which are broadly used to make Hadoop laymen accessible and more usable.

The following sections provide additional information on the individual components:

#### MapReduce

Hadoop MapReduce is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. In terms of programming, there are **two functions** which are most common in MapReduce.

* **The Map Task:** Master computer or node takes input and convert it into divide it into smaller parts and distribute it on other worker nodes. All worker nodes solve their own small problem and give answer to the master node.
* **The Reduce Task:** Master node combines all answers coming from worker node and forms it in some form of output which is answer of our big distributed problem.

Generally both the input and the output are reserved in a file-system. The framework is responsible for scheduling tasks, monitoring them and even re-executes the failed tasks.

#### Hadoop Distributed File System (HDFS)

HDFS is a distributed file-system that provides high throughput access to data. When data is pushed to HDFS, it automatically splits up into multiple blocks and stores/replicates the data thus ensuring high availability and fault tolerance.

##### Note: A file consists of many blocks (large blocks of 64MB and above).

Here are the **main components of HDFS:**

* **NameNode:** It acts as the master of the system. It maintains the name system i.e., directories and files and manages the blocks which are present on the DataNodes.
* **DataNodes:** They are the slaves which are deployed on each machine and provide the actual storage. They are responsible for serving read and write requests for the clients.
* **Secondary NameNode:** It is responsible for performing periodic checkpoints. In the event of NameNode failure, you can restart the NameNode using the checkpoint.

#### Hive

Hive is part of the Hadoop ecosystem and provides an SQL like interface to Hadoop. It is a data warehouse system for Hadoop that facilitates easy data summarization, ad- hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems.

It provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. Hive also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL.

#### The main building blocks of Hive are –

1. **Metastore** – To store metadata about columns, partition and system catalogue.
2. **Driver** – To manage the lifecycle of a HiveQL statement
3. **Query Compiler** – To compiles HiveQL into a directed acyclic graph.
4. **Execution Engine** – To execute the tasks in proper order which are produced by the compiler.
5. **HiveServer** – To provide a Thrift interface and a JDBC / ODBC server.

#### HBase (Hadoop DataBase)

HBase is a distributed, column oriented database and uses HDFS for the underlying storage. As said earlier, HDFS works on write once and read many times pattern, but this isn’t a case always. We may require real time read/write random access for huge

dataset; this is where HBase comes into the picture. HBase is built on top of HDFS and distributed on column-oriented database.

Here are the **main components of HBase:**

* **HBase Master:** It is responsible for negotiating load balancing across all RegionServers and maintains the state of the cluster. It is not part of the actual data storage or retrieval path.
* **RegionServer:** It is deployed on each machine and hosts data and processes I/O requests.

#### Zookeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization and providing group services which are very useful for a variety of distributed systems. HBase is not operational without ZooKeeper.

#### Mahout

Mahout is a scalable machine learning library that implements various different approaches machine learning. At present Mahout contains four **main groups of algorithms:**

* Recommendations, also known as collective filtering
* Classifications, also known as categorization
* Clustering
* Frequent itemset mining, also known as parallel frequent pattern mining

Algorithms in the Mahout library belong to the subset that can be executed in a distributed fashion and have been written to be executable in MapReduce. Mahout is scalable along three dimensions: It scales to reasonably large data sets by leveraging algorithm properties or implementing versions based on Apache Hadoop.

#### Sqoop (SQL-to-Hadoop)

Sqoop is a tool designed for efficiently transferring structured data from SQL Server and SQL Azure to HDFS and then uses it in MapReduce and Hive jobs. One can even use Sqoop to move data from HDFS to SQL Server.

#### Apache Spark:

Apache Spark is a general compute engine that offers fast data analysis on a large scale. Spark is built on HDFS but bypasses MapReduce and instead uses its own data processing framework. Common uses cases for Apache Spark include real-time queries, event stream processing, iterative algorithms, complex operations and machine learning.

#### Pig

Pig is a platform for analyzing and querying huge data sets that consist of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. Pig’s built-in operations can make sense of semi-structured data, such as log files, and the language is extensible using Java to add support for custom data types and transformations.

Pig has three main **key properties:**

* Extensibility
* Optimization opportunities
* Ease of programming

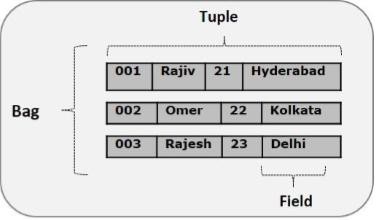
The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets. At the present time, Pig’s infrastructure layer consists of a compiler that produces sequences of MapReduce programs.

|  |  |
| --- | --- |
| **Apache Pig** | **MapReduce** |
| Apache Pig is a data flow language. | MapReduce is a data processing paradigm. |
| It is a high level language. | MapReduce is low level and rigid. |
| Performing a Join operation in Apache Pig is pretty simple. | It is quite difficult in MapReduce to perform a Join operation between datasets. |
| Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig. | Exposure to Java is must to work with MapReduce. |

|  |  |
| --- | --- |
| Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent. | MapReduce will require almost 20 times more the number of lines to perform the same task. |
| There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job. | MapReduce jobs have a long compilation process. |

**Pig Data Model.**

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**.



**Fig:2.3 Pig Data Model**

#### Types

Pig’s data types can be divided into two categories: *scalar* types, which contain a single value, and *complex* types, which contain other types.

#### Scalar Types

Pig’s scalar types are simple types that appear in most programming languages. With the exception of bytearray, they are all represented in Pig interfaces byjava.lang classes, making them easy to work with in UDFs:

*Int,Long ,Float Double, CharArray and byteArray.*

#### Complex Types

Pig has three complex data types: maps, tuples, and bags. All of these types can contain data of any type, including other complex types.

Map

A *map* in Pig is a chararray to data element mapping, where that element can be any Pig type, including a complex type. The chararray is called a key and is used as an index to find the element, referred to as the value.

Map constants are formed using brackets to delimit the map, a hash between keys and values, and a comma between key-value pairs. For example,['name'#'bob', 'age'#55] will create a map with two keys, “name” and“age”. The first value is a chararray, and the second is an integer.

Tuple

A *tuple* is a fixed-length, ordered collection of Pig data elements. Tuples are divided into *fields*, with each field containing one data element. These elements can be of any type—they do not all need to be the same type. A tuple is analogous to a row in SQL, with the fields being SQL columns.

Tuple constants use parentheses to indicate the tuple and commas to delimit fields in the tuple. For example, ('bob', 55) describes a tuple constant with two fields.

Bag

A *bag* is an unordered collection of tuples. Because it has no order, it is not possible to reference tuples in a bag by position. Like tuples, a bag can, but is not required to,

have a schema associated with it. In the case of a bag, the schema describes all tuples within the bag.

Bag constants are constructed using braces, with tuples in the bag separated by commas. For example, {('bob', 55), ('sally', 52), ('john', 25)}constructs a bag with three tuples, each with two fields.

#### Pig Latin.

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.

#### Pig Latin – Type Construction Operators-

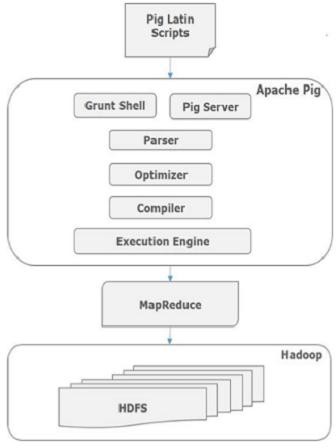
|  |  |
| --- | --- |
| () | **Tuple constructor operator** − This operator is used to construct a tuple. |
| {} | **Bag constructor operator** − This operator is used to construct a bag. |
| [] | **Map constructor operator** − This operator is used to construct a tuple. |

**Pig Latin – Relational 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. |
|  |  |
| **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. |
| CROSS | To create the cross product of two or more relations. |
| **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. |
| **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. |
| **Diagnostic Operators** | |
| DUMP | To print the contents of a relation on the console. |
| DESCRIBE | To describe the schema of a relation. |
| EXPLAIN | To view the logical, physical, or MapReduce execution plans to compute a relation. |
| ILLUSTRATE | To view the step-by-step execution of a series of statements. |



**Fig:2.4 Architecture of PIG**

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

Parser

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

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

Optimizer

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

Compiler

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

Execution engine

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

#### Developing and Testing Pig Latin Script.

Pig provides several tools and diagnostic operators to help you develop your applications. In this section we will explore these and also look at some tools others have written to make it easier to develop Pig with standard editors and integrated development environments (IDEs).

#### Syntax Highlighting and Checking

Syntax highlighting often helps users write code correctly, at least syntactically, the first time around. Syntax highlighting packages exist for several popular editors.

*Pig Latin syntax highlighting packages*

|  |  |
| --- | --- |
| **Tool** | **URL** |
| Eclipse | <http://code.google.com/p/pig-eclipse> |
| Emacs | [http://github.com/cloudera/piglatin-mode,](http://github.com/cloudera/piglatin-mode)<http://sf.net/projects/pig-mode> |
| TextMate | <http://www.github.com/kevinweil/pig.tmbundle> |
| Vim | [http://www.vim.org/scripts/script.php?script\_id=218](http://www.vim.org/scripts/script.php?script_id=2186) |

#### describe

describe shows you the schema of a relation in your script. This can be very helpful as you are developing your scripts. It is especially useful as you are learning Pig Latin and understanding how various operators change the data. describe can be applied to any relation in your script, and you can have multiple describes in a script:

-describe.pig

divs

= load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,

date:chararray, dividends:float);

trimmed = foreach divs generate symbol, dividends;

grpd

= group trimmed by symbol;

avgdiv = foreach grpd generate group, AVG(trimmed.dividends);

describe trimmed;

describe grpd; describe avgdiv;

*trimmed: {symbol: chararray,dividends: float}*

*grpd: {group: chararray,trimmed: {(symbol: chararray,dividends: float)}} avgdiv: {group: chararray,double}*

#### explain

One of Pig’s goals is to allow you to think in terms of data flow instead of MapReduce. But sometimes you need to peek into the barn and see how Pig is compiling your script into MapReduce jobs. Pig provides explain for this. explain is particularly helpful when you are trying to optimize your scripts or debug errors.

There are two ways to use explain. You can explain any alias in your Pig Latin script, which will show the execution plan Pig would use if you stored that relation. You can also take an existing Pig Latin script and apply explain to the whole script in Grunt.

#### illustrate

Often one of the best ways to debug your Pig Latin script is to run your data through it. But if you are using Pig, the odds are that you have a large data set. If it takes several hours to process your data, this makes for a very long debugging cycle.

For example, if you have a join, you have to be careful to sample records from each input such that at least some have the same key. Otherwise, your join will return no results.

To address this issue, the scientists in Yahoo! Research built illustrate into Pig. illustrate takes a sample of your data and runs it through your script, but as it encounters operators that remove data (such as filter, join, etc.), it makes sure that some records pass through the operator and some do not.

#### Pig Statistics

Beginning in version 0.8, Pig produces a summary set of statistics at the end of every run:

The Input, Output, and Counters sections are self-explanatory. The statistics on spills record how many times Pig spilled records to local disk to avoid running out of memory. In local mode theCounters section will be missing because Hadoop does not report counters in local mode.

The Job DAG section at the end describes how data flowed between MapReduce jobs. In this case, the flow was linear.

#### MapReduce Job Status

When you are running your Pig Latin scripts on your Hadoop cluster, finding the status and logs of your job can be challenging. Logs generated by Pig while it plans and manages your query are stored in the current working directory.

#### Debugging Tips

Beyond the tools covered previously, there are a few things I have found useful in debugging Pig Latin scripts. First, if illustrate does not do what you need, use local mode to test your script before running it on your Hadoop cluster.

#### Testing Your Scripts with PigUnit

As part of your development, you will want to test your Pig Latin scripts. Even once they are finished, regular testing helps assure that changes to your UDFs, to your scripts, or in the versions of Pig and Hadoop that you are using do not break your code. *PigUnit* provides a unit- testing framework that plugs into JUnit to help you write unit tests that can be run on a regular basis.PigUnit was added in Pig 0.8.

#### Writing Evaluation

Pig and Hadoop are implemented in Java, and so it is natural to implement UDFs in Java. This allows UDFs access to the Hadoop APIs and to many of Pig’s facilities.

Evaluation Function Basics

All evaluation functions extend the Java class org.apache.pig.EvalFunc. This class uses Java generics. It is parameterized by the return type of your UDF. The core method in this class

is exec. It takes one record and returns one result, which will be invoked for every record that passes through your execution pipeline.

UDFs can also be handed the entire record by passing \* to the UDF. You might expect that in this case the input Tuple argument passed to the UDF would contain all the fields passed into the operator the UDF is in. But it does not.

Interacting with Pig values

Evaluation functions and other UDFs are exposed to the internals of how Pig represents data types. This means that when you read a field and expect it to be an integer, you need to know that it will be an instance of java.lang.Integer. For a complete list of Pig types and how they are represented in Java, see [“Types”.](http://chimera.labs.oreilly.com/books/1234000001811/ch04.html#types)

#### Memory Issues in Eval Funcs

Some operations you will perform in your UDFs will require more memory than is available. As an example, you might want to build a UDF that calculates the cumulative sum of a set of inputs. This will return a bag of values because, for each input, it needs to return the intermediate sum at that input.

#### Filter

The filter statement allows you to select which records will be retained in your data pipeline. A filter contains a predicate. If that predicate evaluates to true for a given record, that record will be passed down the pipeline.

Predicates can contain the equality operators you expect, including == to test equality,

and !=, >, >=,<, and <=. These comparators can be used on any scalar data type. == and != can be applied to maps and tuples.

-- filter\_matches.pig

divs

= load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,

date:chararray, dividends:float);

startswithcm = filter divs by symbol matches 'CM.\*';

#### Load and Store Functions

Pig’s load function is built on top of a Hadoop InputFormat, the class that Hadoop uses to read data.InputFormat serves two purposes: it determines how input will be split between map tasks, and it provides a RecordReader that produces key-value pairs as input to those map tasks. The load function takes these key-value pairs and returns a Pig Tuple.

The base class for the load function is LoadFunc. This is an abstract class, which allows it to provide helper functions and default implementations. Many load functions will only need to extendLoadFunc.

#### Frontend Planning Functions

For all load functions, Pig must do three things as part of frontend planning: 1) it needs to know the input format it should use to read the data; 2) it needs to be sure that the load function

understands where its data is located; and 3) it needs to know how to cast bytearrays returned from the load function.

##### Determining InputFormat

Pig needs to know which InputFormat to use for reading your input. It calls getInputFormat to get an instance of the input format. It gets an instance rather than the class itself so that your load function can control the instantiation: any generic parameters, constructor arguments, etc. For our example load function, this method is very simple. It uses TextInputFormat, an input format that reads text data from HDFS files:

// JsonLoader.java

public InputFormat getInputFormat() throws IOException { return new TextInputFormat();

}

##### Determining the location

Pig communicates the location string provided by the user to the load function via setLocation. So, if the load operator in Pig Latin is A = load 'input';, “input” is the location string. This method is called on both the frontend and backend, possibly multiple times. Thus you need to take care that this method does not do anything that will cause problems if done more than one time. Your load function should communicate the location to its input format. For example, JsonLoader passes the filename via a helper method on FileInputFormat (a superclass of TextInputFormat):

// JsonLoader.java

public void setLocation(String location, Job job) throws IOException { FileInputFormat.setInputPaths(job, location);

}

The Hadoop Job is passed along with the location because that is where input formats usually store their configuration information.

##### Getting the casting functions

Some Pig functions, such as PigStorage and HBaseStorage, load data by default without understanding its type information, and place the data unchanged in DataByteArray objects. At a later time, when Pig needs to cast that data to another type, it does not know how to because it does not understand how the data is represented in the bytearray.

#### Passing Information from the Frontend to the Backend

As with evaluation functions, load functions can make use of UDFContext to pass information from frontend invocations to backend invocations. For details on UDFContext, see [“UDFContext”](http://chimera.labs.oreilly.com/books/1234000001811/ch10.html#udf_context). One significant difference between using UDFContext in evaluation and load functions is determining the instance-specific signature of the function.

In evaluation functions, constructor arguments were suggested as a way to do this. For load functions, the input location usually will be the differentiating factor. However, LoadFunc does not guarantee that it will call setLocation before other methods where you might want to use UDFContext.

#### Additional Load Function Interfaces

Your load function can provide more complex features by implementing additional interfaces. (Implementation of these interfaces is optional.)

##### Loading metadata

Many data storage mechanisms can record the schema along with the data. Pig does not assume the ability to store schemas, but if your storage can hold the schema, it can be very useful. This frees script writers from needing to specify the field names and types as part of the load operator in Pig Latin.

Some types of data storage also partition the data. If Pig understands this partitioning, it can load only those partitions that are needed for a particular script. Both of these functions are enabled by implementing the LoadMetadata interface.

getSchema in the LoadMetadata interface gives your load function a chance to provide a schema. It is passed the location string the user provides as well as the Hadoop Job object, in case it needs information in this object to open the schema.

##### Using partitions

Some types of storage partition their data, allowing you to read only the relevant sections for a given job. The LoadMetadata interface also provides methods for working with partitions in your data. In order for Pig to request the relevant partitions, it must know how the data is partitioned. Pig determines this by calling getPartitionKeys.

#### Store Functions

Pig’s store function is, in many ways, a mirror image of the load function. It is built on top of Hadoop’s OutputFormat. It takes Pig Tuples and creates key-value pairs that its associated output format writes to storage.

StoreFunc is an abstract class, which allows it to provide default implementations for some methods. However, some functions implement both load and store functionality; PigStorage is one example.

#### Store Function Frontend Planning

Store functions have three tasks to fulfill on the frontend:

* Instantiate the OutputFormat they will use to store data.
* Check the schema of the data being stored.
* Record the location where the data will be stored.

##### Determining OutputFormat

Pig calls getOutputFormat to get an instance of the output format that your store function will use to store records. This method returns an instance rather than the classname or the class itself. This allows your store function to control how the class is instantiated.

JsonStorage.java

public OutputFormat getOutputFormat() throws IOException { return new TextOutputFormat<LongWritable, Text>();

}

##### Setting the output location

Pig calls setStoreLocation to communicate the location string the user provides to your store function. Given the Pig Latin store Z into 'output';, “output” is the location string. This method, called on both the frontend and the backend, could be called multiple times

The Hadoop Job is passed to this function as well. Most output formats store the location information in the job.

Pig calls setStoreLocation on both the frontend and backend because output formats usually store their location in the job, as we see in our example store function. This works for MapReduce jobs, where a single output format is guaranteed.

#### Store Functions and UDFContext

Store functions work with UDFContext exactly as load functions do, but with one exception: the signature for store functions is passed to the store function via setStoreFuncUDFContextSignature. See [“Passing Information from the Frontend to the](http://chimera.labs.oreilly.com/books/1234000001811/ch11.html#load_func_udfcontext) [Backend”](http://chimera.labs.oreilly.com/books/1234000001811/ch11.html#load_func_udfcontext) for a discussion of how load functions work with UDFContext.

#### Writing Data

During backend processing, the store function is first initialized, and then takes Pig tuples and converts them to key-value pairs to be written to storage.

##### Preparing to write

Pig calls your store function’s prepareToWrite method in each map or reduce task before writing any data. This call passes a RecordWriter instance to use when writing data. RecordWriter is a class that OutputFormat uses to write individual records.

##### Writing records

putNext is the core method in the store function class. Pig calls this method for every tuple it needs to store. Your store function needs to take these tuples and produce the key-value pairs that its output format expects.

#### Storing Metadata

If your storage format can store schemas in addition to data, your store function can implement the interface StoreMetadata. This provides a storeSchema method that is called by Pig as part of its frontend operations. Pig passes storeSchema a ResourceSchema, the location string, and the job object so that it can connect to its storage.

# **Hive – Introduction**

The term ‘Big Data’ is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

## Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

* + **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
  + **HDFS:**Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

* + **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
  + **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
  + **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

**Note:** There are various ways to execute MapReduce operations:

* + The traditional approach using Java MapReduce program for structured, semi- structured, and unstructured data.
  + The scripting approach for MapReduce to process structured and semi structured data using Pig.
  + The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

## What is Hive

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

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

### Hive is not

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

## **Features of Hive**

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

## **Architecture of Hive**

The following component diagram depicts the architecture of Hive:

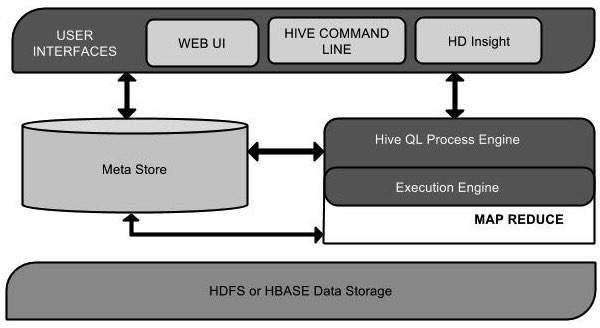


Fig:2.5 Hive Architecture

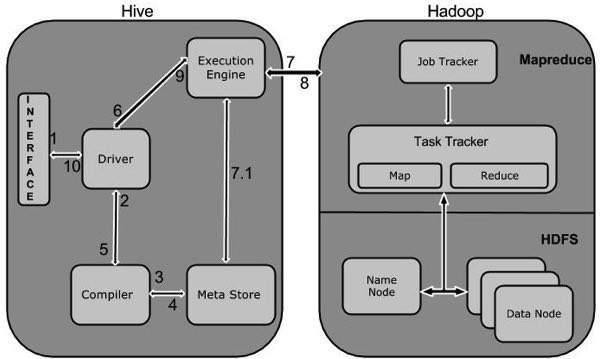
This component diagram contains different units. The following table describes each unit:

|  |  |
| --- | --- |
| **Unit Name** | **Operation** |
| User Interface | Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server). |
| Meta Store | Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a |

|  |  |
| --- | --- |
|  | table, their data types, and HDFS mapping. |
| HiveQL Process Engine | HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it. |
| Execution Engine | The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce. |
| HDFS or HBASE | Hadoop distributed file system or HBASE are the data storage techniques to store data into file system. |

## **Working of Hive**

The following diagram depicts the workflow between Hive and Hadoop.



## **Fig:2.6 Working of Hive**

The following table defines how Hive interacts with Hadoop framework:

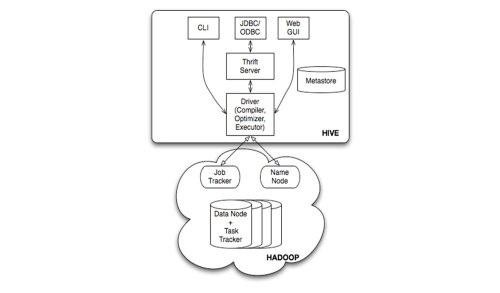
|  |  |
| --- | --- |
| **Step No.** | **Operation** |
| 1 | **Execute Query**  The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute. |
| 2 | **Get Plan**  The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query. |
| 3 | **Get Metadata** |

|  |  |
| --- | --- |
|  | The compiler sends metadata request to Metastore (any database). |
| 4 | **Send Metadata**  Metastore sends metadata as a response to the compiler. |
| 5 | **Send Plan**  The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete. |
| 6 | **Execute Plan**  The driver sends the execute plan to the execution engine. |
| 7 | **Execute Job**  Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job. |
| 7.1 | **Metadata Ops**  Meanwhile in execution, the execution engine can execute metadata operations with Metastore. |
| 8 | **Fetch Result**  The execution engine receives the results from Data nodes. |
| 9 | **Send Results** |

|  |  |
| --- | --- |
|  | The execution engine sends those resultant values to the driver. |
| 10 | **Send Results**  The driver sends the results to Hive Interfaces. |

**Hadoop Hive Architecture**

Hive is one of the most important component of Hadoop,In previous post we discussed about [**Hive Introduction**](http://www.hadooptpoint.com/introduction-hive/)**.**Now we have to know about **Hadoop Hive Architecture.**



Hadoop Hive Architecture

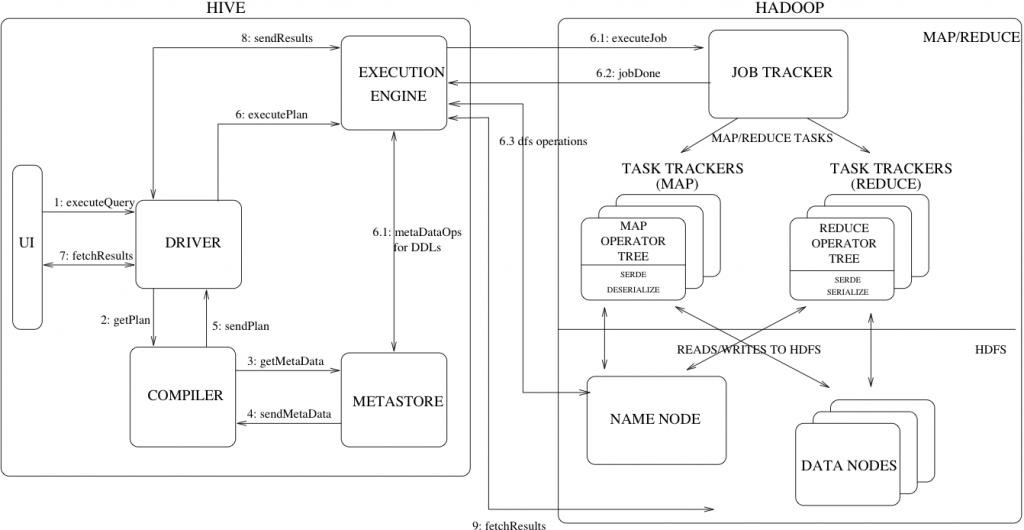
**Fig:2.7 Hadoop Hive Architecture**

The above diagram shows the basic **Hadoop Hive architecture**. Primarily The diagram represents CLI (Command Line Interface),JDBC/ODBC and Web GUI (Web Graphical User Interface ).This represents when user comes with CLI(Hive Terminal) it directly connected to Hive Drivers,When User comes with JDBC/ODBC(JDBC Program) at that time by using API(Thrift Server) it connected to Hive driver and when the user comes with Web GUI(Ambari server) it directly connected to Hive Driver.

The hive driver receives the tasks(Queries) from user and send to Hadoop architecture.The Hadoop architecture uses name node,data node,job tracker and task tracker for receiving and dividing the work what Hive sends to Hadoop (**[Mapreduce Architecture](http://www.hadooptpoint.com/hadoop-mapreduce/)**) .

The below diagram represents clear internal **Hadoop Hive Architecture**

**Fig 2.7 Hadoop Hive Architecture**



Hive\_architecture

The above diagram shows how a typical query flows through the system

**Step 1 :-** The UI calls the execute interface to the Driver

**Step 2 :-** The Driver creates a session handle for the query and sends the query to the compiler to generate an execution plan

**Step 3&4 :-** The compiler needs the metadata so send a request for getMetaData and receives the sendMetaData request from MetaStore.

**Step 5 :-** This metadata is used to typecheck the expressions in the query tree as well as to prune partitions based on query predicates. The plan generated by the compiler is a DAG of stages with each stage being either a map/reduce job, a metadata operation or an operation on HDFS. For map/reduce stages, the plan contains map operator trees (operator trees that are executed on the mappers) and a reduce operator tree (for operations that need reducers).

**Step 6 :-** The execution engine submits these stages to appropriate components (steps 6, 6.1, 6.2 and 6.3). In each task (mapper/reducer) the deserializer associated with the table or intermediate outputs is used to read the rows from HDFS files and these are passed through the associated operator tree.Once the output generate it is written to a temporary HDFS file though the serializer. The temporary files are used to provide the to subsequent map/reduce stages of the plan.For DML operations the final temporary file is moved to the table’s location

**Step 7&8&9 :-** For queries, the contents of the temporary file are read by the execution engine directly from HDFS as part of the fetch call from the Driver

**Major Components of Hive**

**UI :-** UI means User Interface, The user interface for users to submit queries and other operations to the system.

**Driver :-** The Driver is used for receives the quires from UI .This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.

**Compiler :-** The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.

**MetaStore :-** The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.

**Execution Engine :-** The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

This is the main theme of **hadoop hive architecture**

**HIVE VS TRADITIONAL DATABASE**

* Hive resembles a traditional database by supporting SQL interface but it is not a full database. Hive can be better called as data warehouse instead of database.
* Hive enforces schema on read time whereas RDBMS enforces schema on write time.

In RDBMS, a table’s schema is enforced at data load time, If the data being

loaded doesn’t conform to the schema, then it is rejected. This design is called schema on write.

But Hive doesn’t verify the data when it is loaded, but rather when a it is retrieved. This is called schema on read.

Schema on read makes for a very fast initial load, since the data does not have to be read, parsed, and serialized to disk in the database’s internal format. The load operation is just a file copy or move.

Schema on write makes query time performance faster, since the database can index columns and perform compression on the data but it takes longer to load data into the database.

* Hive is based on the notion of Write once, Read many times but RDBMS is designed for Read and Write many times.
* In RDBMS, record level updates, insertions and deletes, transactions and indexes are possible. Whereas these are not allowed in Hive because Hive was built to operate over HDFS data using MapReduce, where full-table scans are the norm and a table update is achieved by transforming the data into a new table.
* In RDBMS, maximum data size allowed will be in 10’s of Terabytes but whereas Hive can 100’s Petabytes very easily.
* As Hadoop is a batch-oriented system, Hive doesn’t support OLTP (Online Transaction Processing) but it is closer to OLAP (Online Analytical Processing) but not ideal since there is significant latency between issuing a query and receiving a reply, due to the overhead of Mapreduce jobs and due to the size of the data sets Hadoop was designed to serve.
* RDBMS is best suited for dynamic data analysis and where fast responses are expected

but Hive is suited for data warehouse applications, where relatively static data is analyzed, fast response times are not required, and when the data is not changing rapidly.

* To overcome the limitations of Hive, HBase is being integrated with Hive to support record level operations and OLAP.
* Hive is very easily scalable at low cost but RDBMS is not that much scalable that too it is very costly scale up.

# HIVEQL Data Types

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

* + Column Types
  + Literals
  + Null Values
  + Complex Types

## Column Types

Column type are used as column data types of Hive. They are as follows:

### Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

|  |  |  |
| --- | --- | --- |
| **Type** | **Postfix** | **Example** |
| TINYINT | Y | 10Y |
| SMALLINT | S | 10S |
| INT | - | 10 |
| BIGINT | L | 10L |

### String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

|  |  |
| --- | --- |
| **Data Type** | **Length** |
| VARCHAR | 1 to 65355 |
| CHAR | 255 |

### Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” and format “yyyy-mm-dd hh:mm:ss.ffffffffff”.

### Dates

DATE values are described in year/month/day format in the form {{YYYY- MM-DD}}.

### Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

DECIMAL(precision, scale)

decimal(10,0)

### Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>

{0:1}

{1:2.0}

{2:["three","four"]}

{3:{"a":5,"b":"five"}}

{2:["six","seven"]}

{3:{"a":8,"b":"eight"}}

{0:9}

{1:10.0}

## Literals

The following literals are used in Hive:

### Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

### Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10-308 to 10308.

## Null Value

Missing values are represented by the special value NULL.

## Complex Types

The Hive complex data types are as follows:

### Arrays

Arrays in Hive are used the same way they are used in Java.

Syntax: ARRAY<data\_type>

### Maps

Maps in Hive are similar to Java Maps.

Syntax: MAP<primitive\_type, data\_type>

### Structs

Structs in Hive is similar to using complex data with comment.

Syntax: STRUCT<col\_name : data\_type [COMMENT col\_comment], ...>

# Hive - Built-in Operators

This chapter explains the built-in operators of Hive. There are four types of operators in Hive:

* + Relational Operators
  + Arithmetic Operators
  + Logical Operators
  + Complex Operators

## Relational Operators

These operators are used to compare two operands. The following table describes the relational operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
| A = B | all primitive types | TRUE if expression A is equivalent to expression B otherwise FALSE. |
| A != B | all primitive types | TRUE if expression A is not equivalent to expression B otherwise FALSE. |
| A < B | all primitive types | TRUE if expression A is less than expression B otherwise FALSE. |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| A > B | all primitive types | TRUE if expression A is greater than expression B otherwise FALSE. |
| A >= B | all primitive types | TRUE if expression A is greater than or equal to expression B otherwise FALSE. |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE. |
| A IS NOT NULL | all types | FALSE if expression A evaluates to NULL otherwise TRUE. |
| A LIKE B | Strings | TRUE if string pattern A matches to B otherwise FALSE. |
| A RLIKE B | Strings | NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B , otherwise FALSE. |
| A REGEXP B | Strings | Same as RLIKE. |

### Example

Let us assume the **employee** table is composed of fields named Id, Name, Salary, Designation, and Dept as shown below. Generate a query to retrieve the employee details whose Id is 1205.

| Id | Name

| Salary | Designation

| Dept |

+ + +

|1201 | Gopal | 45000

|1202 | Manisha | 45000

|1203 | Masthanvali | 40000

|1204 | Krian | 40000

|1205 | Kranthi | 30000

| Technical manager

| Proofreader

| Technical writer

| Hr Admin

| Op Admin

+

| TP

| PR

| TP

| HR

+

|

|

|

|

| Admin|

+ + + + + +

The following query is executed to retrieve the employee details using the above table:

hive> SELECT \* FROM employee WHERE Id=1205;

On successful execution of query, you get to see the following response:

+ + + + +

| ID | Name | Salary | Designation | Dept |

+ + + + +

|1205 | Kranthi | 30000 | Op Admin | Admin |

+ + + + +

The following query is executed to retrieve the employee details whose salary is more than or equal to Rs 40000.

hive> SELECT \* FROM employee WHERE Salary>=40000;

On successful execution of query, you get to see the following response:

+ + + + + +

| ID | Name | Salary | Designation | Dept |

+ + + + + +

|1201 | Gopal | 45000

|1202 | Manisha | 45000

|1203 | Masthanvali| 40000

|1204 | Krian | 40000

| Technical manager

| Proofreader

| Technical writer

| Hr Admin

| TP |

| PR |

| TP |

| HR |

## Arithmetic Operators

These operators support various common arithmetic operations on the operands. All of them return number types. The following table describes the arithmetic operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operators** | **Operand** | **Description** |
| A + B | all number types | Gives the result of adding A and B. |
| A - B | all number types | Gives the result of subtracting B from A. |
| A \* B | all number types | Gives the result of multiplying A and B. |
| A / B | all number types | Gives the result of dividing B from A. |
| A % B | all number types | Gives the reminder resulting from dividing A by B. |
| A & B | all number types | Gives the result of bitwise AND of A and B. |
| A | B | all number types | Gives the result of bitwise OR of A and B. |
| A ^ B | all number types | Gives the result of bitwise XOR of A and B. |
| ~A | all number types | Gives the result of bitwise NOT of A. |

### Example

The following query adds two numbers, 20 and 30.

hive> SELECT 20+30 ADD FROM temp;

On successful execution of the query, you get to see the following response:

+ +

| ADD |

+

| 50

+

+

|

+

## **Logical Operators**

The operators are logical expressions. All of them return either TRUE or FALSE.

|  |  |  |
| --- | --- | --- |
| **Operators** | **Operands** | **Description** |
| A AND B | boolean | TRUE if both A and B are TRUE, otherwise FALSE. |
| A && B | boolean | Same as A AND B. |
| A OR B | boolean | TRUE if either A or B or both are TRUE, otherwise FALSE. |
| A || B | boolean | Same as A OR B. |
| NOT A | boolean | TRUE if A is FALSE, otherwise FALSE. |

|  |  |  |
| --- | --- | --- |
| !A | boolean | Same as NOT A. |

### Example

The following query is used to retrieve employee details whose Department is TP and Salary is more than Rs 40000.

hive> SELECT \* FROM employee WHERE Salary>40000 && Dept=TP;

On successful execution of the query, you get to see the following response:

+

| ID

+

| Name

+ + + +

| Salary

| Designation

| Dept

|

+ + + + + +

|1201 | Gopal | 45000 | Technical manager | TP |

+ + + + + +

## Complex Operators

These operators provide an expression to access the elements of Complex Types.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
| A[n] | A is an Array and n is an int | It returns the nth element in the array A. The first element has index 0. |
| M[key] | M is a Map<K, V> and key has type K | It returns the value corresponding to the key in the map. |
| S.x | S is a struct | It returns the x field of S. |

# Hive - Built-in Functions

This chapter explains the built-in functions available in Hive. The functions look quite similar to SQL functions, except for their usage.

## Built-In Functions

Hive supports the following built-in functions:

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Signature** | **Description** |
| BIGINT | round(double a) | It returns the rounded BIGINT value of the double. |
| BIGINT | floor(double a) | It returns the maximum BIGINT value that is equal or less than the double. |
| BIGINT | ceil(double a) | It returns the minimum BIGINT value that is equal or greater than the double. |
| double | rand(), rand(int seed) | It returns a random number that changes from row to row. |
| string | concat(string A, string B,...) | It returns the string resulting from concatenating B after A. |
| string | substr(string A, int start) | It returns the substring of A starting from start position till the end of string A. |

|  |  |  |
| --- | --- | --- |
| string | substr(string A, int start, int length) | It returns the substring of A starting from start position with the given length. |
| string | upper(string A) | It returns the string resulting from converting all characters of A to upper case. |
| string | ucase(string A) | Same as above. |
| string | lower(string A) | It returns the string resulting from converting all characters of B to lower case. |
| string | lcase(string A) | Same as above. |
| string | trim(string A) | It returns the string resulting from trimming spaces from both ends of A. |
| string | ltrim(string A) | It returns the string resulting from trimming spaces from the beginning (left hand side) of A. |
| string | rtrim(string A) | rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A. |
| string | regexp\_replace(string A, string B, string C) | It returns the string resulting from replacing all substrings in B that match the Java regular expression |

|  |  |  |
| --- | --- | --- |
|  |  | syntax with C. |
| int | size(Map<K.V>) | It returns the number of elements in the map type. |
| int | size(Array<T>) | It returns the number of elements in the array type. |
| value of <type> | cast(<expr> as <type>) | It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from Unix epoch (1970-01-01 00:00:00  UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | It returns the date part of a timestamp string: to\_date("1970- 01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | It returns the year part of a date or a timestamp string: year("1970-01- 01 00:00:00") = 1970, year("1970-  01-01") = 1970 |

|  |  |  |
| --- | --- | --- |
| int | month(string date) | It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") =  11, month("1970-11-01") = 11 |
| int | day(string date) | It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01")  = 1 |
| string | get\_json\_object(string json\_string, string path) | It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid. |

### Example

The following queries demonstrate some built-in functions:

### round() function

hive> SELECT round(2.6) from temp;

On successful execution of query, you get to see the following response:

2.0

### floor() function

hive> SELECT floor(2.6) from temp;

On successful execution of the query, you get to see the following response:

2.0

### floor() function

hive> SELECT ceil(2.6) from temp;

On successful execution of the query, you get to see the following response:

3.0

## Aggregate Functions

Hive supports the following built-in **aggregate functions**. The usage of these functions is as same as the SQL aggregate functions.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Signature** | **Description** |
| BIGINT | count(\*), count(expr), | count(\*) - Returns the total number of retrieved rows. |
| DOUBLE | sum(col), sum(DISTINCT col) | It returns the sum of the elements in the group or the sum of the distinct values of the column in the group. |
| DOUBLE | avg(col), avg(DISTINCT col) | It returns the average of the elements in the group or the average of the distinct values of the column in the group. |
| DOUBLE | min(col) | It returns the minimum value of the column in the group. |
| DOUBLE | max(col) | It returns the maximum value of the column in the group. |

This chapter explains how to create a table and how to insert data into it. The conventions of creating a table in HIVE is quite similar to creating a table using SQL.

## Create Table Statement

Create Table is a statement used to create a table in Hive. The syntax and example are as follows:

### 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 you need to create a table named **employee** using **CREATE TABLE** statement. 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 |

|  |  |  |
| --- | --- | --- |
| 4 | Designation | String |

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, destination String)

COMMENT ‘Employee details’ ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘\t’ LINES TERMINATED BY ‘\n’

STORED AS TEXTFILE;

If you add the option IF NOT EXISTS, Hive ignores the statement in case the table already exists.

On successful creation of table, you get to see the following response:

OK

Time taken: 5.905 seconds hive>

### JDBC Program

The JDBC program to create a table is given example.

import java.sql.SQLException;

import java.sql.Connection; import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveCreateTable {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",

"");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("CREATE TABLE IF NOT EXISTS "

+" employee ( eid int, name String, "

+" salary String, destignation String)"

+" COMMENT ‘Employee details’"

+" ROW FORMAT DELIMITED"

+" FIELDS TERMINATED BY ‘\t’"

+" LINES TERMINATED BY ‘\n’"

+" STORED AS TEXTFILE;");

System.out.println(“ Table employee created.”);

con.close();

}

}

Save the program in a file named HiveCreateDb.java. The following commands are used to compile and execute this program.

$ javac HiveCreateDb.java

$ java HiveCreateDb

### Output

Table employee created.

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.

### Example

We will 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;

On successful download, you get to see the following response:

OK

Time taken: 15.905 seconds hive>

### JDBC Program

Given below is the JDBC program to load given data into the table.

import java.sql.SQLException; import java.sql.Connection; import java.sql.ResultSet; import java.sql.Statement; import java.sql.DriverManager;

public class HiveLoadData {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",

"");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("LOAD DATA LOCAL INPATH '/home/user/sample.txt'" + "OVERWRITE INTO TABLE employee;");

System.out.println("Load Data into employee successful");

con.close();

}

}

Save the program in a file named HiveLoadData.java. Use the following commands to compile and execute this program.

$ javac HiveLoadData.java

$ java HiveLoadData

### Output:

Load Data into employee successful

**Hive - Alter Table**

This chapter explains how to alter the attributes of a table such as changing its table name, changing column names, adding columns, and deleting or replacing columns.

## Alter Table Statement

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 ...])

## Rename To… Statement

The following query renames the table from **employee** to **emp**.

hive> ALTER TABLE employee RENAME TO emp;

### JDBC Program

The JDBC program to rename a table is as follows.

import java.sql.SQLException; import java.sql.Connection; import java.sql.ResultSet; import java.sql.Statement; import java.sql.DriverManager;

public class HiveAlterRenameTo {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",

"");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("ALTER TABLE employee RENAME TO emp;"); System.out.println("Table Renamed Successfully"); con.close();

}

}

Save the program in a file named HiveAlterRenameTo.java. Use the following commands to compile and execute this program.

$ javac HiveAlterRenameTo.java

$ java HiveAlterRenameTo

### Output:

Table renamed successfully.

**HiveQL – Query Data Select-Order By**

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

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

Given below is the syntax of the SELECT query:

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];

## Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, 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.

+

| ID

+

| Name

+ + + +

| Salary

| Designation

| Dept

|

+ + + + + +

The following query retrieves the employee details using the above scenario:

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

On successful execution of the query, you get to see the following response:

+

| ID

+

|1201

|1202

|1203

|1204

+

| Name

+

| Gopal

| Manisha

| Masthanvali

| Krian

+ +

| Salary

| Designation

+

+

+

| Dept

+

| 45000

| 45000

| 40000

| 40000

| Technical manager | TP

| Proofreader

| Technical writer

| Hr Admin

| PR

| TP

| HR

+

|

+

|

|

|

|

+ + + + + +

### JDBC Program

The JDBC program to apply where clause for the given example is as follows.

import java.sql.SQLException; import java.sql.Connection; import java.sql.ResultSet; import java.sql.Statement; import java.sql.DriverManager;

public class HiveQLWhere {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",

"");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery("SELECT \* FROM employee WHERE salary>30000;");

System.out.println("Result:");

System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {

System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " + res.getString(4) + " " + res.getString(5));

}

con.close();

}

}

Save the program in a file named HiveQLWhere.java. Use the following commands to compile and execute this program.

$ javac HiveQLWhere.java

$ java HiveQLWhere

### Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Salary | Designation | Dept |
| 1201 | Gopal | 45000 | Technical manager | TP |
| 1202 | Manisha | 45000 | Proofreader | PR |
| 1203 | Masthanvali | 40000 | Technical writer | TP |
| 1204 | Krian | 40000 | Hr Admin | HR |

**HiveQL - Select-Group By**

This chapter explains the details of GROUP BY clause in a SELECT statement. The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

## Syntax

The syntax of GROUP BY clause is as follows:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference [WHERE where\_condition] [GROUP BY col\_list] [HAVING having\_condition] [ORDER BY col\_list]]

[LIMIT number];

## Example

Let us take an example of SELECT…GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

+

| ID

+

|1201

+

| Name

+

| Gopal

+ + + +

| Salary

| Designation

| Dept

|

+

+

+

+

| 45000

| Technical manager | TP

|

+

+

+ + + +

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |1202 | | Manisha | | | 45000 | | | Proofreader | | | PR | | |
| |1203 | | Masthanvali | | | 40000 | | | Technical writer | | | TP | | |
| |1204 | | Krian | | | 45000 | | | Proofreader | | | PR | | |
| |1205 | | Kranthi | | | 30000 | | | Op Admin | | | Admin | | |

The following query retrieves the employee details using the above scenario.

hive> SELECT Dept,count(\*) FROM employee GROUP BY DEPT;

On successful execution of the query, you get to see the following response:

+

+

| Dept | Count(\*)

+ +

|Admin |

|PR

|TP

+

1

| 2

| 3

+

+

|

+

|

|

|

+

**HiveQL - Select-Joins**

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOIN.

## Syntax

join\_table:

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference join\_condition

| table\_reference LEFT SEMI JOIN table\_reference join\_condition

| table\_reference CROSS JOIN table\_reference [join\_condition]

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

+ + +

+ +

| ID | NAME | AGE | ADDRESS | SALARY

+ + + + +

+

|

+

+ + +

+ +

+

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | 1 | | | Ramesh | | | 32 | | | Ahmedabad | | | 2000.00 | | |
| | 2 | | | Khilan | | | 25 | | | Delhi | | | 1500.00 | | |
| | 3 | | | kaushik | | | 23 | | | Kota | | | 2000.00 | | |
| | 4 | | | Chaitali | | | 25 | | | Mumbai | | | 6500.00 | | |
| | 5 | | | Hardik | | | 27 | | | Bhopal | | | 8500.00 | | |
| | 6 | | | Komal | | | 22 | | | MP | | | 4500.00 | | |
| | 7 | | | Muffy | | | 24 | | | Indore | | | 10000.00 | | |

Consider another table ORDERS as follows:

+ + + + +

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+ + + + +

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+ + + + +

There are different types of joins given as follows:

* + JOIN
  + LEFT OUTER JOIN
  + RIGHT OUTER JOIN
  + FULL OUTR JOIN

## JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT FROM CUSTOMERS c JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| | 3 | | kaushik | | 23 | | | 3000 | | |
| | 3 | | kaushik | | 23 | | | 1500 | | |
| | 2 | | Khilan | | 25 | | | 1560 | | |
| | 4 | | Chaitali | | 25 | | | 2060 | | |

## LEFT OUTER JOIN

+ + + + +

| ID | NAME | AGE | AMOUNT |

+ + + + +

+ + + + +

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c

LEFT OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+ + + + +

| ID | NAME | AMOUNT | DATE |

+

| 1

| 2

| 3

| 3

| 4

| 5

| 6

| 7

+ +

+ +

| Ramesh

| Khilan

| kaushik

| kaushik

| NULL

| 1560

| 3000

| 1500

| NULL

|

| Chaitali | 2060

| 2009-11-20 00:00:00 |

| 2009-10-08 00:00:00 |

| 2009-10-08 00:00:00 |

| 2008-05-20 00:00:00 |

| Hardik

| Komal

| Muffy

| NULL

| NULL

| NULL

| NULL

| NULL

| NULL

|

|

|

+ + + + +

## RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+

| ID

+

| 3

| 3

| 2

| 4

+

+

| NAME

+ +

| AMOUNT | DATE

+

| kaushik

| kaushik

| Khilan

+

+

+

|

+

| 3000

| 1500

| 1560

| Chaitali | 2060

| 2009-10-08 00:00:00 |

| 2009-10-08 00:00:00 |

| 2009-11-20 00:00:00 |

| 2008-05-20 00:00:00 |

+

+

+

+

## FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c

FULL OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+ + + +

| ID | NAME | AMOUNT | DATE

+

|

+ + + + +

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 1 | | | Ramesh | | | NULL | | | NULL | | |
| | | 2 | | | Khilan | | | 1560 | | | 2009-11-20 00:00:00 | | |
| | | 3 | | | kaushik | | | 3000 | | | 2009-10-08 00:00:00 | | |
| | | 3 | | | kaushik | | | 1500 | | | 2009-10-08 00:00:00 | | |
| | | 4 | | | Chaitali | | | 2060 | | | 2008-05-20 00:00:00 | | |
| | | 5 | | | Hardik | | | NULL | | | NULL | | |
| | | 6 | | | Komal | | | NULL | | | NULL | | |
| | | 7 | | | Muffy | | | NULL | | | NULL | | |
| | | 3 | | | kaushik | | | 3000 | | | 2009-10-08 00:00:00 | | |
| | | 3 | | | kaushik | | | 1500 | | | 2009-10-08 00:00:00 | | |
| | | 2 | | | Khilan | | | 1560 | | | 2009-11-20 00:00:00 | | |
| | | 4 | | | Chaitali | | | 2060 | | | 2008-05-20 00:00:00 | | |

+ + + + +

**NoSQL Databases: An Overview**

Over the last few years we have seen the rise of a new type of databases, known as NoSQL databases, that are challenging the dominance of relational databases. Relational databases have dominated the software industry for a long time providing mechanisms to store data persistently, concurrency control, transactions, mostly standard interfaces and mechanisms to integrate application data, reporting. The dominance of relational databases, however, is cracking.

**NoSQL what does it mean**

What does NoSQL mean and how do you categorize these databases? NoSQL means Not Only SQL, implying that when designing a software solution or product, there are more than one storage mechanism that could be used based on the needs. NoSQL was a hashtag (#nosql) choosen for a meetup to discuss these new databases. The most important result of the rise of NoSQL is Polyglot Persistence. NoSQL does not have a prescriptive definition but we can make a set of common observations, such as:

* Not using the relational model
* Running well on clusters
* Mostly open-source
* Built for the 21st century web estates
* Schema-less

HBase – Introduction

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

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

## Limitations of Hadoop

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

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

## Hadoop Random Access Databases

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

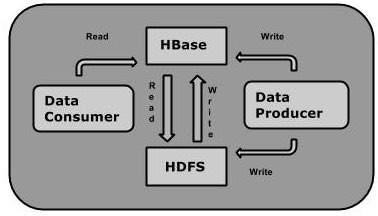
## What is HBase?

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

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

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

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



## Fig 2.7 HBase Read and Write operation

## HBase and HDFS

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

Stoage Mechanism in HBase

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

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

Given below is an example schema of table in HBase.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Rowid** | **Column Family** | | | **Column Family** | | | **Column Family** | | | **Column Family** | | |
| **col1** | **col2** | **col3** | **col1** | **col2** | **col3** | **col1** | **col2** | **col3** | **col1** | **col2** | **col3** |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |

## Column Oriented and Row Oriented

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

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

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

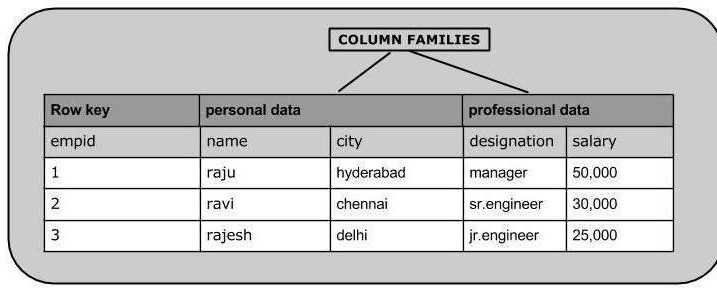


Fig :2.8 Coloumn Families

## **HBase and RDBMS**

|  |  |
| --- | --- |
| **HBase** | **RDBMS** |
| HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families. | An RDBMS is governed by its schema, which describes the whole structure of tables. |

|  |  |
| --- | --- |
| It is built for wide tables. HBase is horizontally scalable. | It is thin and built for small tables. Hard to scale. |
| No transactions are there in HBase. | RDBMS is transactional. |
| It has de-normalized data. | It will have normalized data. |
| It is good for semi-structured as well as structured data. | It is good for structured data. |

Features of HBase

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

## Where to Use HBase

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

## Applications of HBase

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

## **HBase History**

|  |  |
| --- | --- |
| **Year** | **Event** |
| Nov 2006 | Google released the paper on BigTable. |
| Feb 2007 | Initial HBase prototype was created as a Hadoop contribution. |
| Oct 2007 | The first usable HBase along with Hadoop 0.15.0 was released. |
| Jan 2008 | HBase became the sub project of Hadoop. |
| Oct 2008 | HBase 0.18.1 was released. |
| Jan 2009 | HBase 0.19.0 was released. |
| Sept 2009 | HBase 0.20.0 was released. |
| May 2010 | HBase became Apache top-level project. |

In HBase, tables are split into regions and are served by the region servers. Regions are vertically divided by column families into “Stores”. Stores are saved as files in HDFS. Shown below is the architecture of HBase.

**Note:** The term ‘store’ is used for regions to explain the storage structure.

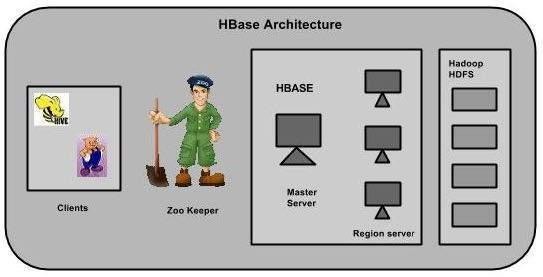


Fig 2.9 HBase Architecture

HBase has three major components: the client library, a master server, and region servers. Region servers can be added or removed as per requirement.

## **MasterServer**

The master server -

* Assigns regions to the region servers and takes the help of Apache ZooKeeper for this task.
* Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.
* Maintains the state of the cluster by negotiating the load balancing.
* Is responsible for schema changes and other metadata operations such as creation of tables and column families.

## **Regions**

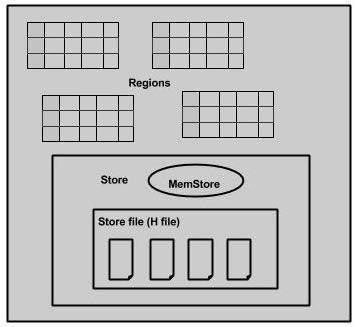
Regions are nothing but tables that are split up and spread across the region servers.

### **Region server**

The region servers have regions that -

* Communicate with the client and handle data-related operations.
* Handle read and write requests for all the regions under it.
* Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:



F

Fig : 2.10 Region Diagram

The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.

## **Zookeeper**

* Zookeeper is an open-source project that provides services like maintaining configuration information, naming, providing distributed synchronization, etc.
* Zookeeper has ephemeral nodes representing different region servers. Master servers use these nodes to discover available servers.
* In addition to availability, the nodes are also used to track server failures or network partitions.
* Clients communicate with region servers via zookeeper.
* In pseudo and standalone modes, HBase itself will take care of zookeeper.

# **Loading Data in HBase & Querying Data in Hbase.**

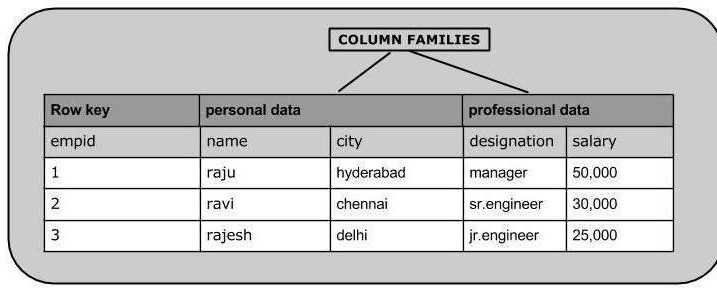
**HBase - Create Data:**

## **Inserting Data using HBase Shell**

This chapter demonstrates how to create data in an HBase table. To create data in an HBase table, the following commands and methods are used:

* **put** command,
* **add()** method of **Put** class, and
* **put()** method of **HTable** class.

As an example, we are going to create the following table in HBase.

Using **put** command, you can insert rows into a table. Its syntax is as follows:

put ’<table name>’,’row1’,’<colfamily:colname>’,’<value>’

### Inserting the First Row

Let us insert the first row values into the emp table as shown below.

hbase(main):005:0> put 'emp','1','personal data:name','raju'

0 row(s) in 0.6600 seconds

hbase(main):006:0> put 'emp','1','personal data:city','hyderabad'

0 row(s) in 0.0410 seconds

hbase(main):007:0> put 'emp','1','professional data:designation','manager'

0 row(s) in 0.0240 seconds

hbase(main):007:0> put 'emp','1','professional data:salary','50000'

0 row(s) in 0.0240 seconds

Insert the remaining rows using the put command in the same way. If you insert the whole table, you will get the following output.

hbase(main):022:0> scan 'emp'

ROW

COLUMN+CELL

1 column=personal data:city, timestamp=1417524216501, value=hyderabad

1 column=personal data:name, timestamp=1417524185058, value=ramu

1 column=professional data:designation, timestamp=1417524232601,

value=manager

1 column=professional data:salary, timestamp=1417524244109, value=50000

2 column=personal data:city, timestamp=1417524574905, value=chennai

2 column=personal data:name, timestamp=1417524556125, value=ravi

2 column=professional data:designation, timestamp=1417524592204,

value=sr:engg

2 column=professional data:salary, timestamp=1417524604221, value=30000

3 column=personal data:city, timestamp=1417524681780, value=delhi

3 column=personal data:name, timestamp=1417524672067, value=rajesh

3 column=professional data:designation, timestamp=1417524693187,

value=jr:engg

3 column=professional data:salary, timestamp=1417524702514,

value=25000

## Inserting Data Using Java API

You can insert data into Hbase using the **add()** method of the **Put** class. You can save it using the **put()** method of the **HTable** class. These classes belong to the **org.apache.hadoop.hbase.client** package. Below given are the steps to create data in a Table of HBase.

### Step 1:Instantiate the Configuration Class

The **Configuration** class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the**HbaseConfiguration** class as shown below.

Configuration conf = HbaseConfiguration.create();

### Step 2:Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts configuration object and table name as parameters. You can instantiate HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

### Step 3: Instantiate the PutClass

To insert data into an HBase table, the **add()** method and its variants are used. This method belongs to **Put**, therefore instantiate the put class. This class requires the row name you want to insert the data into, in string format. You can instantiate the **Put** class as shown below.

Put p = new Put(Bytes.toBytes("row1"));

### Step 4: Insert Data

The **add()** method of **Put** class is used to insert data. It requires 3 byte arrays representing column family, column qualifier (column name), and

the value to be inserted, respectively. Insert data into the HBase table using the add() method as shown below.

p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column name"),Bytes.toBytes("value"));

### Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the **put()** method of HTable class as shown below.

hTable.put(p);

### Step 6: Close the HTable Instance

After creating data in the HBase Table, close the **HTable** instance using the**close()** method as shown below.

hTable.close();

Given below is the complete program to create data in HBase Table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration; import org.apache.hadoop.hbase.client.HTable; import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes;

public class InsertData{

public static void main(String[] args) throws IOException {

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable hTable = new HTable(config, "emp");

// Instantiating Put class

// accepts a row name.

Put p = new Put(Bytes.toBytes("row1"));

// adding values using add() method

// accepts column family name, qualifier/row name ,value

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("name"),Bytes.toBytes("raju"));

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("hyderabad"));

p.add(Bytes.toBytes("professional"),Bytes.toBytes("designation"),

Bytes.toBytes("manager"));

p.add(Bytes.toBytes("professional"),Bytes.toBytes("salary"),

Bytes.toBytes("50000"));

// Saving the put Instance to the HTable.

hTable.put(p);

System.out.println("data inserted");

// closing HTable

hTable.close();

}

}

Compile and execute the above program as shown below.

$javac InsertData.java

$java InsertData

The following should be the output:

data inserted

# HBase - Update Data

## Updating Data using HBase Shell

You can update an existing cell value using the **put** command. To do so, just follow the same syntax and mention your new value as shown below.

put ‘table name’,’row ’,'Column family:column name',’new value’

The newly given value replaces the existing value, updating the row.

### Example

Suppose there is a table in HBase called **emp** with the following data.

hbase(main):003:0> scan 'emp'

ROW COLUMN + CELL

row1 column = personal:name, timestamp = 1418051555, value = raju

row1 column = personal:city, timestamp = 1418275907, value = Hyderabad

row1 column = professional:designation, timestamp = 14180555,value = manager row1 column = professional:salary, timestamp = 1418035791555,value = 50000

1 row(s) in 0.0100 seconds

The following command will update the city value of the employee named ‘Raju’ to Delhi.

hbase(main):002:0> put 'emp','row1','personal:city','Delhi'

0 row(s) in 0.0400 seconds

The updated table looks as follows where you can observe the city of Raju has been changed to ‘Delhi’.

hbase(main):003:0> scan 'emp'

ROW COLUMN + CELL

row1 column = personal:name, timestamp = 1418035791555, value = raju row1 column = personal:city, timestamp = 1418274645907, value = Delhi

row1 column = professional:designation, timestamp = 141857555,value = manager row1 column = professional:salary, timestamp = 1418039555, value = 50000

1 row(s) in 0.0100 seconds

## Updating Data Using Java API

You can update the data in a particular cell using the **put()** method. Follow the steps given below to update an existing cell value of a table.

### Step 1: Instantiate the Configuration Class

**Configuration** class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the**HbaseConfiguration** class as shown below.

Configuration conf = HbaseConfiguration.create();

### Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

### Step 3: Instantiate the Put Class

To insert data into HBase Table, the **add()** method and its variants are used. This method belongs to **Put**, therefore instantiate the **put** class. This class requires the row name you want to insert the data into, in string format. You can instantiate the **Put** class as shown below.

Put p = new Put(Bytes.toBytes("row1"));

### Step 4: Update an Existing Cell

The **add()** method of **Put** class is used to insert data. It requires 3 byte arrays representing column family, column qualifier (column name), and the value to be inserted, respectively. Insert data into HBase table using the**add()** method as shown below.

p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column name"),Bytes.toBytes("value")); p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("Delih"));

### Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the **put()** method of the HTable class as shown below.

hTable.put(p);

### Step 6: Close HTable Instance

After creating data in HBase Table, close the **HTable** instance using the close() method as shown below.

hTable.close();

Given below is the complete program to update data in a particular table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes;

public class UpdateData{

public static void main(String[] args) throws IOException {

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable hTable = new HTable(config, "emp");

// Instantiating Put class

//accepts a row name

Put p = new Put(Bytes.toBytes("row1"));

// Updating a cell value

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("Delih"));

// Saving the put Instance to the HTable.

hTable.put(p);

System.out.println("data Updated");

// closing HTable hTable.close();

}

}

Compile and execute the above program as shown below.

$javac UpdateData.java

$java UpdateData

The following should be the output:

data Updated

# HBase - Read Data

## Reading Data using HBase Shell

The **get** command and the **get()** method of **HTable** class are used to read data from a table in HBase. Using **get** command, you can get a single row of data at a time. Its syntax is as follows:

get ’<table name>’,’row1’

### Example

The following example shows how to use the get command. Let us scan the first row of the **emp** table.

hbase(main):012:0> get 'emp', '1'

COLUMN

CELL

personal : city timestamp = 1417521848375, value = hyderabad

personal : name timestamp = 1417521785385, value = ramu

professional: designation timestamp = 1417521885277, value = manager

professional: salary timestamp = 1417521903862, value = 50000

4 row(s) in 0.0270 seconds

## Reading a Specific Column

Given below is the syntax to read a specific column using the **get** method.

hbase> get 'table name', ‘rowid’, {COLUMN ⇒ ‘column family:column name ’}

### Example

Given below is the example to read a specific column in HBase table.

hbase(main):015:0> get 'emp', 'row1', {COLUMN ⇒ 'personal:name'} COLUMN CELL

personal:name timestamp = 1418035791555, value = raju

1 row(s) in 0.0080 seconds

## Reading Data Using Java API

To read data from an HBase table, use the **get()** method of the HTable class. This method requires an instance of the **Get** class. Follow the steps given below to retrieve data from the HBase table.

### Step 1: Instantiate the Configuration Class

**Configuration** class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the**HbaseConfiguration** class as shown below.

Configuration conf = HbaseConfiguration.create();

### Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

### Step 3: Instantiate the Get Class

You can retrieve data from the HBase table using the **get()** method of the**HTable** class. This method extracts a cell from a given row. It requires a **Get**class object as parameter. Create it as shown below.

Get get = new Get(toBytes("row1"));

### Step 4: Read the Data

While retrieving data, you can get a single row by id, or get a set of rows by a set of row ids, or scan an entire table or a subset of rows.

You can retrieve an HBase table data using the add method variants in **Get**class.

To get a specific column from a specific column family, use the following method.

get.addFamily(personal)

To get all the columns from a specific column family, use the following method.

get.addColumn(personal, name)

### Step 5: Get the Result

Get the result by passing your **Get** class instance to the get method of the**HTable** class. This method returns the **Result** class object, which holds the requested result. Given below is the usage of **get()** method.

Result result = table.get(g);

### Step 6: Reading Values from the Result Instance

The **Result** class provides the **getValue()** method to read the values from its instance. Use it as shown below to read the values from the **Result** instance.

byte [] value = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name")); byte [] value1 = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

Given below is the complete program to read values from an HBase table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration; import org.apache.hadoop.hbase.client.Get;

import org.apache.hadoop.hbase.client.HTable; import org.apache.hadoop.hbase.client.Result; import org.apache.hadoop.hbase.util.Bytes;

public class RetriveData{

public static void main(String[] args) throws IOException, Exception{

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable table = new HTable(config, "emp");

// Instantiating Get class

Get g = new Get(Bytes.toBytes("row1"));

// Reading the data

Result result = table.get(g);

// Reading values from Result class object

byte [] value = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

byte [] value1 = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

// Printing the values

String name = Bytes.toString(value); String city = Bytes.toString(value1);

System.out.println("name: " + name + " city: " + city);

}

}

Compile and execute the above program as shown below.

$javac RetriveData.java

$java RetriveData

The following should be the output:

name: Raju city: Delhi

**Kafka**

Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

Point to Point Messaging System

In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

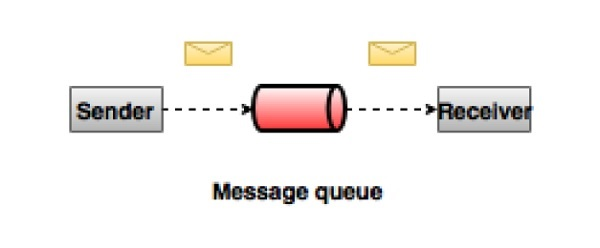


Fig 2.11 Kafka Point to Point Messaging System

In  the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.

Kafka

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Following are a few benefits of Kafka −

* Reliability − Kafka is distributed, partitioned, replicated and fault tolerance.
* Scalability − Kafka messaging system scales easily without down time..
* Durability − Kafka uses Distributed commit log which means messages persists on disk as fast as possible, hence it is durable..
* Performance − Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

Kafka is very fast and guarantees zero downtime and zero data loss.

Need for Kafka

Kafka is a unified platform for handling all the real-time data feeds. Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the ability to handle a large number of diverse consumers. Kafka is very fast, performs 2 million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.

Apache Flume

Apache Flume is a tool/service/data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log files, events (etc...) from various sources to a centralized data store.

Flume is a highly reliable, distributed, and configurable tool. It is principally designed to copy streaming data (log data) from various web servers to HDFS

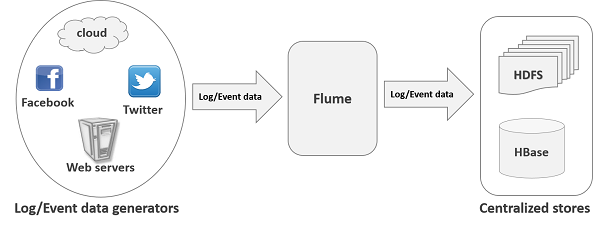


Fig: 2.12 Flume from various web servers to HDFS

Applications of Flume

Assume an e-commerce web application wants to analyze the customer behavior from a particular region. To do so, they would need to move the available log data in to Hadoop for analysis. Here, Apache Flume comes to our rescue.

Flume is used to move the log data generated by application servers into HDFS at a higher speed.

Advantages of Flume

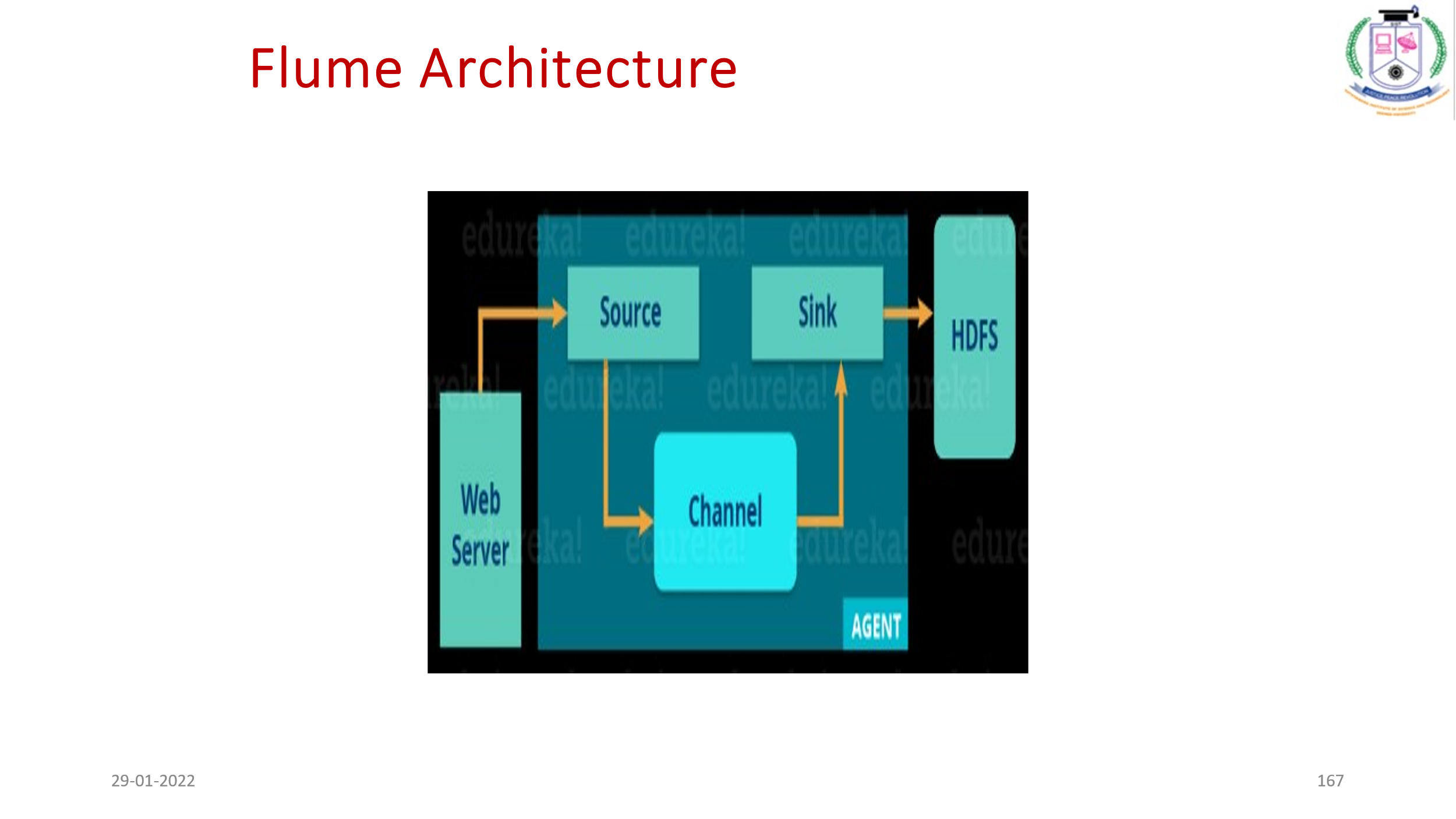
Here are the advantages of using Flume −

* Using Apache Flume we can store the data in to any of the centralized stores (HBase, HDFS).
* When the rate of incoming data exceeds the rate at which data can be written to the destination, Flume acts as a mediator between data producers and the centralized stores and provides a steady flow of data between them.
* Flume provides the feature of contextual routing.
* The transactions in Flume are channel-based where two transactions (one sender and one receiver) are maintained for each message. It guarantees reliable message delivery.
* Flume is reliable, fault tolerant, scalable, manageable, and customizable.

Features of Flume

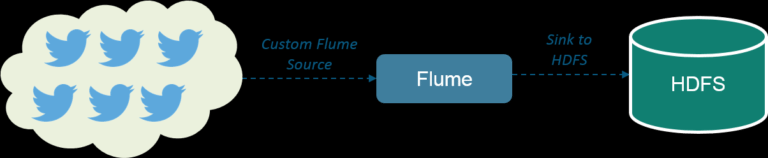
Some of the notable features of Flume are as follows −

* Flume ingests log data from multiple web servers into a centralized store (HDFS, HBase) efficiently.
* Using Flume, we can get the data from multiple servers immediately into Hadoop.
* Along with the log files, Flume is also used to import huge volumes of event data produced by social networking sites like Facebook and Twitter, and e-commerce websites like Amazon and Flipkart.
* Flume supports a large set of sources and destinations types.
* Flume supports multi-hop flows, fan-in fan-out flows, contextual routing, etc.
* Flume can be scaled horizontally.



* The flume agent has 3 components: source, sink and channel.
  + **Source**: It accepts the data from the incoming streamline and stores the data in the channel.
  + **Channel**: In general, the reading speed is faster than the writing speed. Thus, we need some buffer to match the read & write speed difference. Basically, the buffer acts as a intermediary storage that stores the data being transferred temporarily and therefore prevents data loss. Similarly, channel acts as the local storage or  a temporary storage between the source of data and persistent data in the HDFS.
  + **Sink**: Then, our last component i.e. Sink, collects the data from the channel and commits or writes the data in the HDFS permanently.

Streaming Twitter Data:

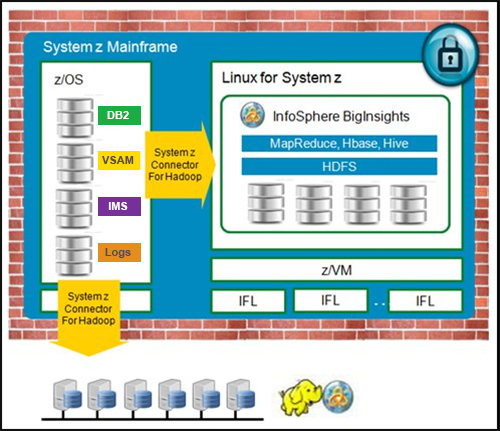


InfoSphere® BigInsight

* InfoSphere® BigInsight is **a software platform for discovering, analyzing, and visualizing data from disparate sources**. You use this software to help process and analyze the volume, variety, and velocity of data that continually enters your organization every day.

**Highlights of InfoSphere BigInsights**

* igInsights allows organizations to cost-effectively analyze a wide variety and large volume of data to gain insights that were not previously possible.
* BigInsights is focused on providing enterprises with the capabilities they need to meet critical business requirements while maintaining compatibility with the Hadoop project.
* BigInsights includes a variety of IBM technologies that enhance and extend the value of open-source Hadoop software to facilitate faster time-to-value, including application accelerators, analytical facilities, development tools, platform improvements and enterprise software integration.
* While BigInsights offers a wide range of capabilities that extend beyond the Hadoop functionality, IBM has taken an optin approach: you can use the IBM extensions to Hadoop based on your needs rather than being forced to use the extensions that come with InfoSphere BigInsights.
* In addition to core capabilities for installation, configuration and management, InfoSphere BigInsights includes advanced analytics and user interfaces for the non-developer business analyst.
* It is flexible to be used for unstructured or semi-structured information; the solution does not require schema definitions or data preprocessing and allows for structure and associations to be added on the fly across information types.
* The platform runs on commonly available, low-cost hardware in parallel, supporting linear scalability; as information grows, we simply add more commodity hardware.
* InfoSphere BigInsights provides a unique set of capabilities that combine the innovation from the Apache Hadoop ecosystem with robust support for traditional skill sets and already installed tools. The ability to leverage existing skills and tools through open-source capabilities helps drive lower total cost of ownership and faster time-to-value.  Thus InfoSphere BigInsights enables new solutions for problems that were previously too large and complex to solve cost-effectively.

 **Extend mainframe analytic capabilities with BigInsights**

* The Integrated Facility for Linux is an IBM mainframe and Power Systems processor dedicated to running the Linux operating system. On IBM Z and IBM LinuxONE machines, IFLs can be used with or without hypervisors such as z/VM and KVM.
* **Advantages of IBM infosphere Big insights**
* Hadoop applications can exist within the System z security perimeter.
* Clients can use mainframe technologies, including IBM HiperSockets™, to securely access production data, and move that data to and from Hadoop for processing.
* Clients can realize the management advantages of running Hadoop on a private cloud infrastructure, providing configuration flexibility and virtualized storage, and avoiding need to deploy and manage discrete cluster nodes and a separate network infrastructure.
* Clients can extend System z governance to hybrid Hadoop implementations.

**Open source utilities in InfoSphere BigInsights:**

* InfoSphere BigInsights is 100% compatible with open source Hadoop.
* open source utilities in InfoSphere BigInsights:
  + PIG
  + Hive / HCatalog
  + Oozie
  + HBASE
  + Zookeeper
  + Flume
  + Avro

Chukwa

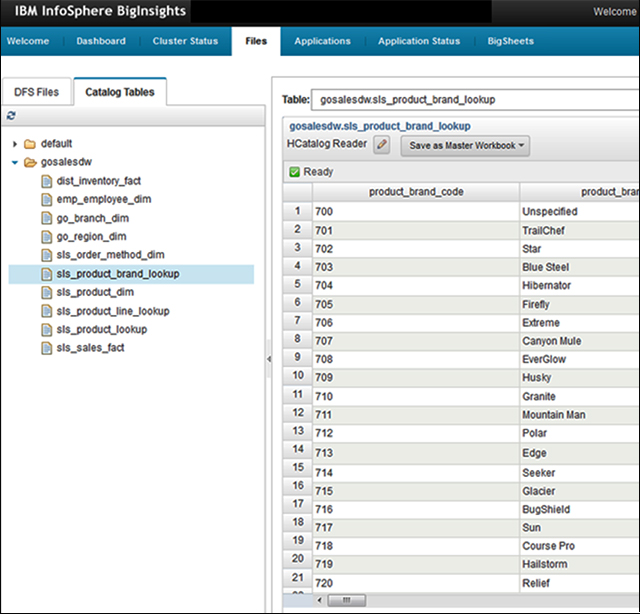
**Advanced software capabilities of IBM Infosphere:**

* Big SQL: Big SQL is a rich, ANSI-compliant SQL implementation.
* SQL language compatibility
* Support for native data sources
* Performance
* Federation
* Security

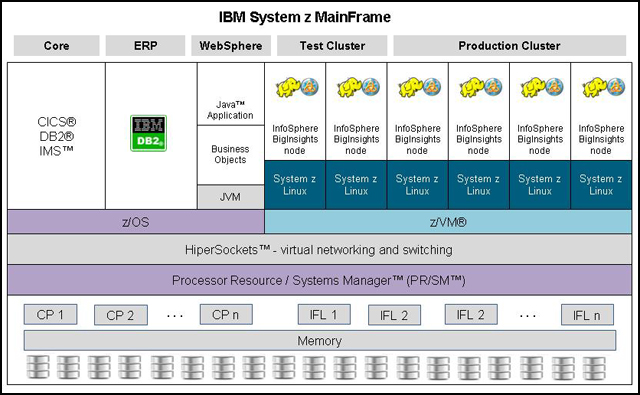
**IBM infosphere interfaces**

* Big R: Big R is a set of libraries that provide end-to-end integration with the popular R programming language that is included in InfoSphere BigInsights. Big R provides a familiar environment for developers and data scientists proficient with the R language.
* Big Sheets: Big Sheets is a spreadsheet style data manipulation and visualization tool that allows business users to access and analyze data in Hadoop without the need to be knowledgeable in Hadoop scripting languages or MapReduce programming. The BigSheets interface is shown in Figure 3. Using built-in line readers, BigSheets can import data in multiple formats. In this example, it is importing data that is stored in Hive.

**InfoSphere BigSheets interface**



* **Application Accelerators**: IBM InfoSphere BigInsights extends the capabilities of open source Hadoop with accelerators that use pre-written capabilities for common big data use cases to build quickly high-quality applications. Here are some of the accelerators that are included in InfoSphere BigInsights:
* **Text Analytics Accelerators:** A set of facilities for developing applications that analyze text across multiple spoken languages
* **Machine Data Accelerators**: Tools that are aimed at developers that make it easy to develop applications that process log files, including web logs, mail logs, and various specialized file formats



**Accelerators:** Tools to easily import and analyze social data at scale from multiple online sources, including tweets, boards, and blogs

**Deploying Big insights on System Z**

**When to use Biginsights on a cloud based service**

* Clients do not want to be bothered with acquiring and maintain a Hadoop on cluster on their own premises.
* Data that is stored and processed on Hadoop comes largely from external sources as opposed to local data sources (such as cloud-based social media aggregators).
* Clients want to retain the flexibility to alter the size of clusters up and down based on changing requirements.
* IBM Infosphere:

<https://namitkabra.wordpress.com/2015/01/08/what-is-infosphere-biginsights/>

http://www.redbooks.ibm.com/abstracts/tips1215.html#contents