# Basics

# What is Hive?

Data warehouse software project built on top of Apache Hadoop.

Using Hive, we can query and analyze the data stored in HDFS.

Hive is used to analyze structured data.

HQL queries implicitly translated to one or more Hadoop Map reduce job(s) for execution.

## What is the hive version used for D Project?

1.3.0

Hive—version

## Features of hive? What hive gives us?

It provides SQL type language for querying called HiveQL or HQL.

It is designed for OLAP (OnLine Analytical Processing).

It stores schema in a database and processed data in HDFS

It is familiar, fast, scalable, and extensible.

1. **Hive is not?**

Hive is not a relational database

Hive is not designed for OnLine Transaction Processing (OLTP)

Hive is not a language for realtime queries and row-level updates

1. **Hive modules/Components?**

The main components of Hive are:

1. **Metastore**:

This component is used to store Schema, metadata of tables, databases and HDFS mapping.

Meta store component can be folder like database called as DERBY or it can be MYSQL.

**Which is the metastore used in D Project?**

**postgresql**

1. **Driver**:

Hive Driver consists of Compiler, Optimizer and Executor.

**Compiler:** Upon receiving a HiveQL statement driver invokes compiler.

The compiler then translates this statement into a plan which consists of a DAG (directed acyclic graph) of map-reduce jobs.

**Optimizer:** Enhances HiveQL for faster execution

**Executor:**The driver submits the individual map-reduce jobs from the DAG to the execution engine in a topological order.

**Which is the execution engine of hive?**

The execution engine used by Hive currently is Hadoop.

Hive manages just life cycle of HiveQL statement during compilation, optimization and execution. The execution engine is Hadoop

On receiving the HiveQL statement, from the Thrift server or some other interfaces, it creates a session handle using session handle we can keep track of statistics like execution time, number of output rows, etc.

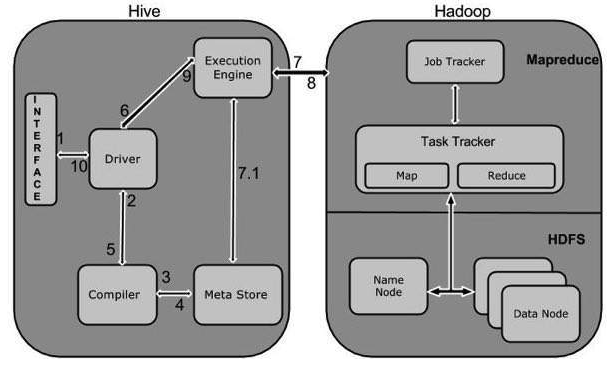
1. **Interfaces to use Hive**

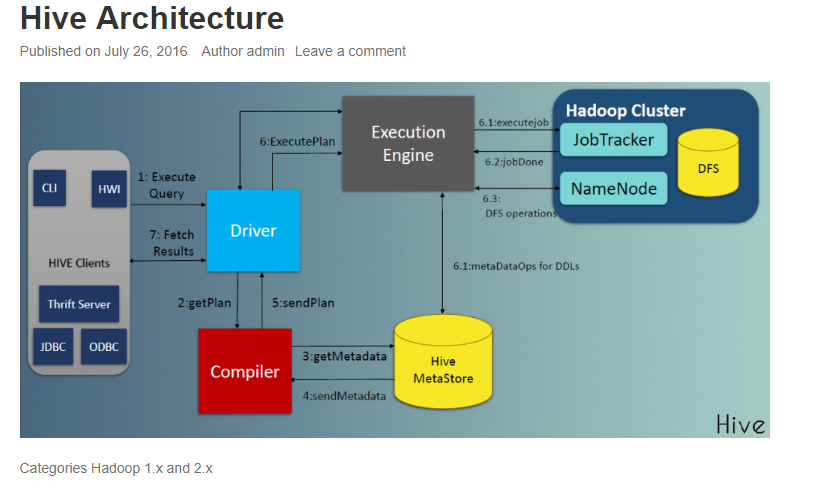
CLI: **C**ommand **L**ine **I**nterface(beeline) like MySQL shell connected to database. **HWIH**ive **W**eb **I**nterface

JDBC/ODBC client for programmatic access.

1. **Thrift server:** Interface between hive driver and other applications.This server exposes a very simple client API to execute HiveQL statements
2. **Working of Hive?**

**Select count(\*) from TABLE\_NAME;**





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.

1. **Fetch Result:** The execution engine receives the results from Data nodes.
2. **Send Results:**The execution engine sends those resultant values to the driver.
3. **Send Results:** The driver sends the results to Hive Interfaces.

## Sample script to create internal table in hive?

CREATE TABLE TABLE\_NAME (COLUMN\_NAME, INT, date STRING, precip INT)

PARTITIONED BY (COLUMN\_NAME DATA\_TYPE)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

LOCATION ‘/hive/data/weather’;

ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.orc.OrcSerde'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'

TBLPROPERTIES ( 'orc.compress'='SNAPPY', 'orc.create.index'='true');

## Sample script to create External table?

CREATE **EXTERNAL** TABLE weatherext ( wban INT, date STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

LOCATION ‘ /hive/data/weatherext’;

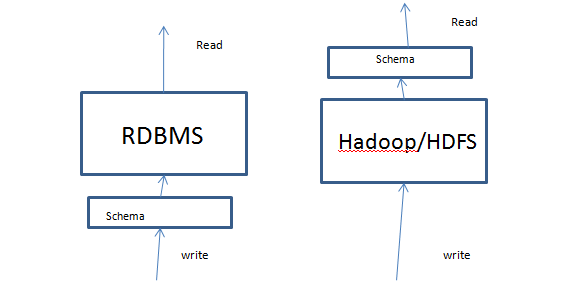
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.orc.OrcSerde'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'

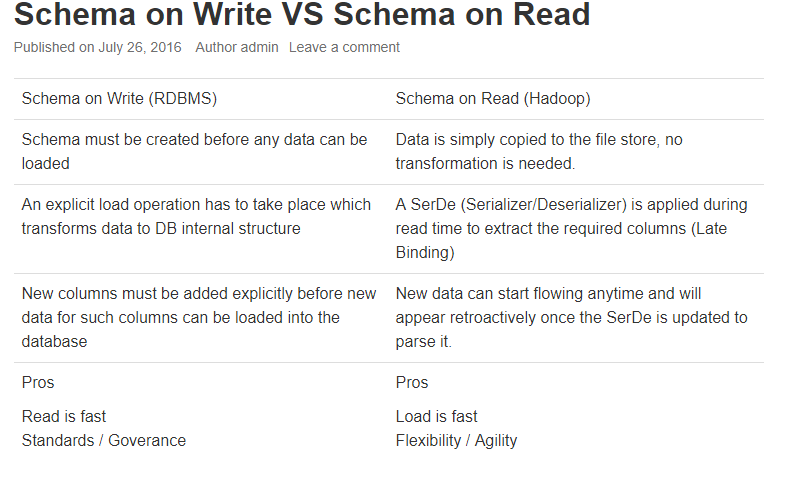
TBLPROPERTIES ( 'orc.compress'='SNAPPY', 'orc.create.index'='true');

1. **Schema on Write and schema on read**

****

**In case of RDBMS we write data once and read regularly. Write frequency is less and read frequency is more. So we need good prformace for read so RDBMS uses schema on write.**

**In case of Hadoop as it is for bigdata. Huge data flow will be there daily and we need to save it so we need more performance during writing. Where as reading frequency is less say for example we read one day’s data once for analysis. So hive uses schema on read.**

****

1. **What happens when there is data type mismatch in case of hive?**

As hive follows schema on read null values will be displayed for the fields when data type mismatch happens q

**Data Units of Hive:**

**Databases:**

In hive databases are created to avoid naming conflicts for tables, views, partitions, columns, and so on.

Also creating database provides security so that we can provide access to one user or group of user.

# Tables:

Homogeneous units of data which have the same schema

## What is the difference between internal table and external table in Hive? IMP

<https://data-flair.training/blogs/hive-internal-tables-vs-external-tables/>

## In what case External table should be used and in what case managed table should be used? IMP

Use **EXTERNAL** tables when the data is also used outside of Hive. For example, the data files are read and processed by an existing program that doesn't lock the files. The data is permanent i.e. used when needed.

Suppose you have a data set, and you have to perform some analytics/problem statements on it. Because of the nature of problem statements, few of them can be done by HiveQL, few of them need Pig Latin and few of them need Map Reduce etc., to get the job done. In this situation External Table comes into picture- the same data set can be used to solve entire analytics instead of having different different copies of same data set for the different different tools. Here Hive don't need authority on the data set because several tools are going to use it.

Use **INTERNAL** tables when: The data is temporary. You want Hive to completely manage the lifecycle of the table and data.

here can be a scenario, where entire analytics/problem statements can be solved by only HiveQL. In such situation Internal Table comes into picture- Means you can put the entire data set into Hive's Warehouse and Hive is going to have complete authority on the data set.

**Partitions:**

Each Table can have one or more partition Keys which determines how the data is stored. Partitions—apart from being storage units—also allow the user to efficiently identify the rows that satisfy a specified criteria; for example, a date\_partition of type STRING and country\_partition of type STRING. Each unique value of the partition keys defines a partition of the Table. For example, all “US” data from “2009-12-23” is a partition of the page\_views table. Therefore, if you run analysis on only the “US” data for 2009-12-23, you can run that query only on the relevant partition of the table, thereby speeding up the analysis significantly. Note however, that just because a partition is named 2009-12-23 does not mean that it contains all or only data from that date; partitions are named after dates for convenience; it is the user’s job to guarantee the relationship between partition name and data content! Partition columns are virtual columns, they are not part of the data itself but are derived on load.

**Buckets (or Clusters):**

Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page\_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page\_view table. These can be used to efficiently sample the data.

## Loading Data into Hive table:

* **Language Manual**
  + **Data manipulate statement**
    - **Data retrival queries**
      * **UDF**

# String functions:

|  |  |  |
| --- | --- | --- |
| **Return type** | **Name(signature)** | **Description** |
| int | **instr**(string str, string substr) | This is like indexOf() of java.  Returns the position of the first occurrence of substr in str. Returns null if either of the arguments are null and returns 0 if substr could not be found in str. Be aware that this is not zero based. The first character in str has index 1 |
| String | **get\_json\_object**(string json\_string, string path) | Extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It will return null if the input json string is invalid. NOTE: The json path can only have the characters [0-9a-z\_], i.e., no upper-case or special characters. Also, the keys **cannot start with numbers.** This is due to restrictions on Hive column names. |
|  |  |  |

**Built in table generating functions: (UDTF)**

1. **How do you choose columns given 100 columns?**

* We know that Partitioning helps to speed up queries with predicates (where condition). So we should choose column which we use most in the where condition
* Need to make sure that cardinality is not too high, otherwise, your query performance would be degraded.
* Partition column selected should contain values that are static over time. If value of partitioned column is changed then there will be row movement between partitions. This operation will be costly.
* Value of the partitioned column should be known. We should create partition on the column which will get some random values on regular basis
* We should partition column such that each partition should have similar amount of data.

# Partitions

## What is Hive Partitioning?

Dividing a table into related parts based on the values of particular columns like date, city, and department.

## Why do we need partitions?

Partitioning the table improves performance significantly because we work on part of data instead working on entire data

## How to create partition?

To create data partitioning in Hive following command is used-

CREATE TABLE table\_name (column1 data\_type, column2 data\_type) PARTITIONED BY (partition1 data\_type, partition2 data\_type,….);

## How partition improves performance?

<https://data-flair.training/blogs/apache-hive-partitions/>

Hive Data Partitioning Example

## What are the types of hive partitioning?

There are two types of Partitioning in Apache Hive.

1. Static Partitioning
2. Dynamic Partitioning

## Explain Hive Static partitioning?

https://analyticshut.com/static-vs-dynamic-partitioning-in-hive/

* We must manually decide how many partitions tables will have and value for those partitions. Consider we have employed table and we want to partition it based on department name. There are a limited number of departments, hence a limited number of partitions.
* One of the major drawbacks of static partitioning is, when we are loading data to some partition, we must make sure we are putting the right data in the right partition.
* One of the major **drawbacks** of static partitioning is, when we are loading data to some partition, we must make sure we are putting the right data in the right partition. In the below example, while loading HR data, it is our responsibility to make sure that the file only has HR data. (Learn more about Load hive command here).

*CREATE EXTERNAL TABLE employee\_dept (*

*emp\_id INT,*

*emp\_name STRING*

*) PARTITIONED BY (*

*dept\_name STRING*

*)*

*location '/user/maheshmogal/employee\_dept';*

*LOAD DATA LOCAL INPATH 'hr.txt'*

*INTO TABLE employee\_dept*

*PARTITION (dept\_name='HR');*

Above queries will create table on partition on department name and then load data from our local system to this table in partitioned name “HR”.

Now you should see another drawback with this method. What if you have 1000 different departments or you have some other partitioning column like year and month? You must create those manually and that is no FUN!! This is where dynamic partitions help us.

* With all these drawbacks you might think, static partitions are not that useful.

**Advantage:** But when you have limited and well know a set of values (like departments or state names etc.), you can use static partitioning.

Static partitioning is faster for loading data as all the information is already present in the query. Also, it will stop you from creating unnecessary partitions creating performance issues the very thing which partitions are supposed to help us with.

## Explain Hive dynamic partitioning?

Dynamic partitions provide us with flexibility and create partitions automatically depending on the data that we are inserting into the table.

In last tutorial, we have created orders table. We can insert data in to that table with following query.

hive (maheshmogal)> insert overwrite table order\_partition partition (year,month)

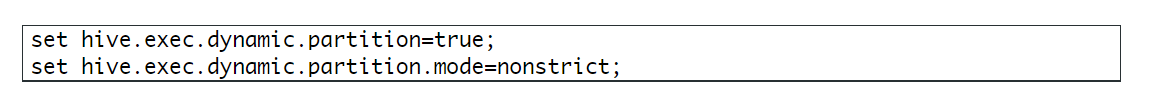
select order\_id, order\_date, order\_status, substr(order\_date,1,4) year, substr(order\_date,5,2) month from orders;

FAILED: SemanticException [Error 10096]: Dynamic partition strict mode requires at least one static partition column. To turn this off set hive.exec.dynamic.partition.mode=nonstrict

Oh we got error? what is this about?

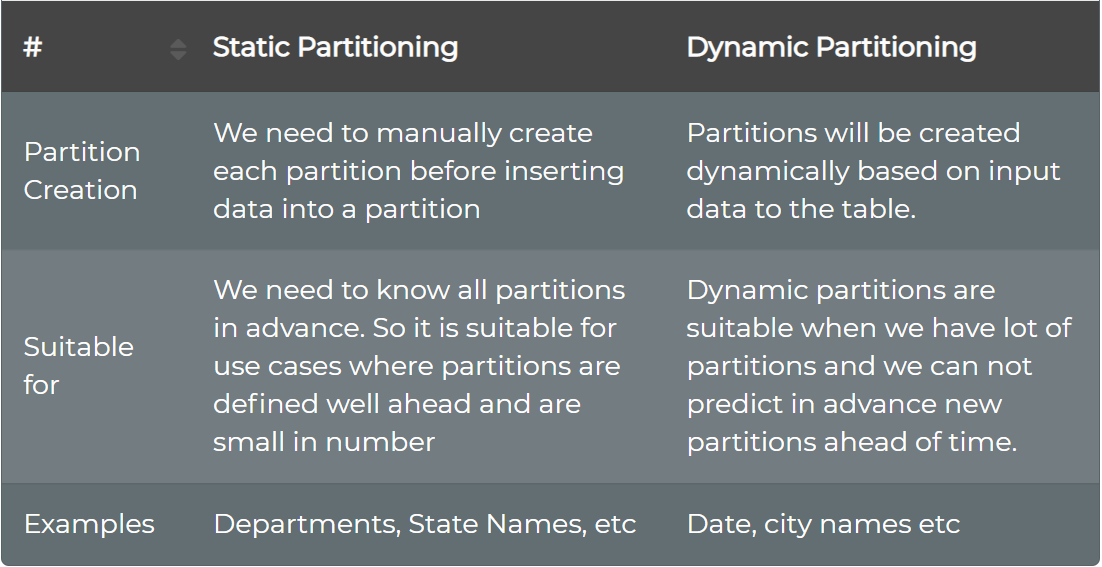
By default, Hive does not enable dynamic partition. This is to protect us, from creating from a huge number of partitions accidentally. In dynamic partition, we are telling hive which column to use for dynamic partition. If we select the wrong column (say order id) we can end up with millions of partitions.

An easier way is, we can set Hive’s dynamic property mode to nonstrict using the following command. This will allow us to create dynamic partitions in the table without any static partition.



Now if you run the insert query, it will create all required dynamic partitions and insert correct data into each partition. This all happens in one single query and we do not have to manually check data and correct partition.

## What is the difference between static partitioning and dynamic partitioning?



# Bucketing

## What is bucketing in hive?

For decomposing table data sets into more manageable parts, it uses Hive Bucketing concept.

## Why we need Bucketing when we have partitioning?

Even though partitioning helps to decompose the data and improve performance it will give effective results only in few scenarios. Like

– When there is the limited number of partitions.

– Or, while partitions are of comparatively equal size.

Although, it is not possible in all scenarios. For example when are partitioning our tables based geographic locations like country. Hence, some bigger countries will have large data in partitions. While small countries data will create small data in partitions. Hence, at that time Partitioning will not be ideal.

So, to solve that problem of over partitioning, Hive offers Bucketing concept. It is another effective technique for decomposing table data sets into more manageable parts.

## How Bucketing happens in Hive?

Basically, this concept is based on hashing function on the bucketed column. Along with mod by the total number of buckets.

Where the hash\_function depends on the type of the bucketing column.

## Which keyword is used to divide tables into buckets?

CLUSTERED BY clause

***CREATE TABLE bucketed\_user(***

***firstname VARCHAR(64),***

***lastname VARCHAR(64),***

***address STRING,***

***city VARCHAR(64),***

***state VARCHAR(64),***

***post STRING,***

***phone1 VARCHAR(64),***

***phone2 STRING,***

***email STRING,***

***web STRING***

***)***

***COMMENT 'A bucketed sorted user table'***

***PARTITIONED BY (country VARCHAR(64))***

***CLUSTERED BY (state) SORTED BY (city) INTO 32 BUCKETS***

***STORED AS SEQUENCEFILE;***

## How data is stored in folder when you divide table into buckets?

Generally, in the table directory, each bucket is just a file, and Bucket numbering is 1-based.

## Can we do bucket with partition?

Yes, bucketing can be done with and without partitioning.

## When will you use bucketing in Hive? **IMP**

Partitioning is most effective for low volume data, as it carries the possibility of too many small partition creations and too many directories.

since bucketing results in equal volumes of data in each partition, joins at Map side will be quicker.

# Performance

## What are the performance aspects we can do on hive table?

<https://www.qubole.com/blog/hive-best-practices/>

* **Partitioning the table**

Hive partitioning is an effective method to improve the query performance on larger tables. Partitioning allows you to store data in separate sub-directories under table location. It dramatically helps the queries which are queried upon the partition key(s). Although the selection of partition key is always a prudent decision, it should always be a low cardinal attribute. For example, if your data is associated with the time dimension, then the date could be a good partition key. Similarly, if data is associated with location, like a country or state, it’s a good idea to have hierarchical partitions like country/state.

* **Input Format Selection**

Input formats play a critical role in Hive performance. For example, JSON, the text type of input format, is not the right choice for an extensive production system where data volume is high. These types of readable formats take a lot of space and have some parsing overhead (e.g. JSON parsing). To address these problems, Hive comes with columnar input formats like RCFile, ORC, etc. Columnar formats allow you to reduce the read operations in analytics queries by allowing each column to be accessed individually. There are some other binary formats like Avro, sequence files, Thrift, and ProtoBuf, which can help in various use cases.

* **Compress map/reduce output:**

Compression techniques significantly reduce the intermediate data volume, which internally minimizes the amount of data transfers between mappers and reducers. All this generally occurs over the network. Compression can be applied to the mapper and reducer output individually. Keep in mind that gzip-compressed files are not splittable. That means this should be applied with caution. Compressed file size should not be larger than a few hundred megabytes. Otherwise, it can potentially lead to an imbalanced job. Other options of compression codec could be snappy, lzo, bzip, etc.

* **De-normalizing data**

Normalization is a standard process used to model your data tables with certain rules to deal with a redundancy of data and anomalies. In simpler words, if you normalize your data sets, you end up creating multiple relational tables which can be joined at the run time to produce the results. Joins are expensive and complicated operations to perform and are common reasons for performance issues. Because of that, it’s a good idea to avoid highly normalized table structures because they require to join queries to derive the desired metrics.

* **Map join**

Map joins are efficient if a table on the other side of a join is small enough to fit in the memory. Hive supports a parameter, hive.auto.convert.join, which suggests that Hive tries to map join automatically when it’s set to “true.” When using this parameter, be sure the auto-convert is enabled in the Hive environment.

Additionally, it’s essential to ensure the bucketing flag is set (SET hive.enforce.bucketing=true;) every time before writing data to the bucketed table. To leverage the bucketing in the join operation, we should SET hive.optimize.bucketmapjoin=true. This setting hints to Hive to do bucket level join during the map stage join. It also reduces the scan cycles to find a particular key because bucketing ensures that the key is present in a specific bucket.

* **Parallel execution**

Hadoop can execute MapReduce jobs in parallel, and several queries executed on Hive automatically use this parallelism. However, single, complex Hive queries commonly are translated to several MapReduce jobs that are executed by default sequencing. Some of a query’s MapReduce stages are often not interdependent and could be executed in parallel. They then can take advantage of spare capacity on a cluster and improve cluster utilization while at the same time reducing the overall query execution time. Hive’s configuration to change this behavior is merely switching a single flag SET hive.exec.parallel=true.

* **Vectorization**:

Vectorization allows Hive to process a batch of rows together instead of processing one row at a time. Each batch consists of a column vector which is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. To enable vectorization, set this configuration parameter SET hive.vectorized.execution.enabled=true.

* **Unit Testing**

Merely speaking, unit testing determines whether the smallest testable piece of your code works exactly as you expect. Unit testing gives a couple of benefits, i.e., detecting problems early, making it easier to change and refactor code, being a form of documentation that explains how code works, to name a few.

In Hive, you can unit test UDFs, SerDes, streaming scripts, Hive queries, and more. To a large extent, it is possible to verify your whole HiveQL query’s correctness by running quick local unit tests without even touching a Hadoop cluster. Because executing HiveQL query in the local mode takes literally seconds, compared to minutes, hours or days if it runs in the Hadoop mode, it certainly saves enormous amounts of development time.

There are several tools available that help you to test Hive queries. Some of them that you might want to look at HiveRunner, Hive\_test, and Beetest.

* **Sampling**

Sampling allows users to take a subset of datasets and analyze it without analyzing the entire data set. If a representative sample is used, then a query can return meaningful results and finish quicker and consume fewer compute resources. Hive offers a built-in TABLESAMPLE clause that allows you to sample your tables. TABLESAMPLE can sample at various granularity levels – it can return only subsets of buckets (bucket sampling), or HDFS blocks (block sampling), or only first N records from each input split. Alternatively, you can implement your own UDF that filters out records according to your sampling algorithm

# File Storage formats

## Text/CSV Files

Text and CSV files are quite common and frequently Hadoop developers and data scientists received text and CSV files to work upon.

However, CSV files do not support block compression, thus compressing a CSV file in Hadoop often comes at a significant read performance cost.

## JSON

JSON records contain JSON files where each line is its own JSON datum.

In the case of JSON files, metadata is stored, and the file is also splitable but again it also doesn’t support block compression.

## Sequence Files

Sequence file stores data in binary format

It also doesn’t store metadata

It supports block compression.

Due to complexity, sequence files are mainly used in flight data as an intermediate storage.

## RC Files (Record columnar)

RC file was the first columnar file in Hadoop.

Significant compression and query performance benefits.

Disadvantage: RC does not support schema evolution. If you want to add anything to RC file you will have to rewrite the file. Also, it is a slower process.

## ORC Files (Optimized Record Columnar).

ORC is the compressed version of RC file and supports all the benefits of RC file with some enhancements like ORC files compress better than RC files, enabling faster queries.

Some benchmarks indicate that ORC files compress to be the smallest of all file formats in Hadoop.

Disadvantage:

But it doesn’t support schema evolution.

## Parquet Files

Parquet file is another columnar file

Parquet also enjoys the features like compression and query performance benefits but is generally slower to write than non-columnar file formats.

In Parquet format, new columns can be added at the end of the structure

Note: In Hive Parquet column names should be lowercase. If it is of mixed cases, then hive will not read it and will give you null value.

However, Impala can handle mixed cases.

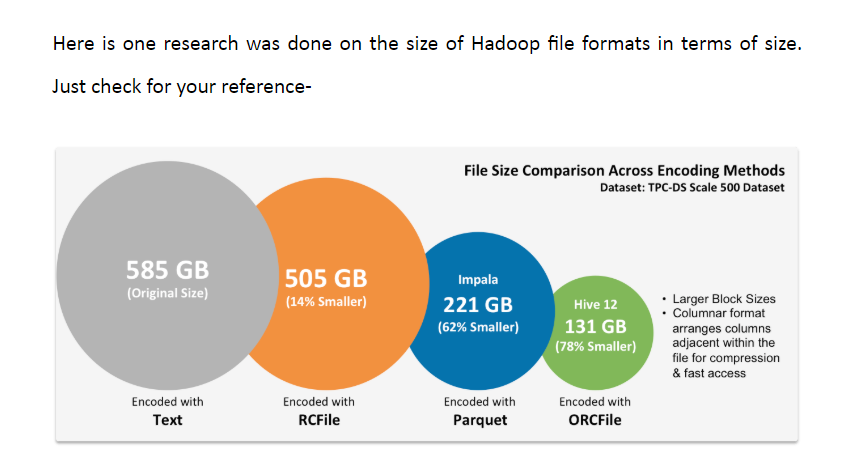
## Avro

Avro stores metadata with the data itself and allows specification of an independent schema for reading the file.

You can rename, add, delete, and change the data types of fields by defining a new independent schema.

Also, Avro files are splitable, support block compression and enjoy broad, relatively mature, tool support within the Hadoop ecosystem.

## Comparing compression of file format



## Difference between ORC parquet and Avro?

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **ORC** | **Parquet** | **Avro** |
| **Data Storage Format** | Column based Format | Column based Format | Row based Format |
| **Block Compression** | **Yes.**  High compression. | **Yes.**  High compression better than Avro but less than ORC | **Yes**  Compression rate is less compared to Parquet. |
| **Schema evolution** | **Yes.**  Not as good as Avro, but better than Parquet.s | **Yes**  Yes But not as good as Avro. When it comes to columnar ORC is better than parquet | **Yes (very good.)**  Avro offers superior schema evolution, Because of innovative use of JSON to describe the data, while using binary format to optimize storage size. |
| **performance** | **Write is Slow.**  **Read is fast** | **Write is Slow.**  **Read is fast** | **Write is fast.**  **Read is slow.** |
| **Compatible platforms** | Hive, Presto.  HortonWorks Data Platform (HDP) | Impala, Arrow drill, Spark.  Cloudera Distrbution of Hadoop (CDH) | Kafka, Druid |
| **Use case** | When you need more read performance  (Horton works and Hive)  We should use column based file format when we are trying to do a query on very few columns from your entire dataset. | When you need more read performance.  (Cloudera Impala)  We should use column based file format when we are trying to do a query on very few columns from your entire dataset. | If schema is going to change over time |
| **Storage Requirements** | **131 GB** | **221 GB** |  |

# What are the formats you can use for hive tables? **IMP**

**Text, sequence file, RC file , AVRO file, ORC file, Parquet file.**

# UDF

How to write an UDF?

<http://www.bmc.com/blogs/how-to-write-a-hive-user-defined-function-udf-in-java/>

1. Need to create a class which extends org.apache.hadoop.hive.ql.exec.UDF

Ex: public class PersonaDataDeserializeUDF extends UDF

1. Need to implement a method with name evaluate refer link for more details.

**public** String evaluate(String inputStr)

{

//Logic here

}

1. Build a this file as a jar files
2. Upload built jar to an hdfs path

*hdfs dfs -put persona-udf.jar /AppData/UDF/*

1. Create Hive UDF function using this jar

For this user should have admin permissions

*set role admin*

Then create udf function

*create function biads.persona\_byte\_to\_json AS ‘fully qualified class name’ USING JAR ‘hdfs://hacluster/AppData/UDF/persona-udf.jar’;*

*create function biads.persona AS ‘fully qualified class name’ USING JAR ‘hdfs://hacluster/AppData/UDF/persona-udf-1.jar’;*

1. Then start using the function as per requirement