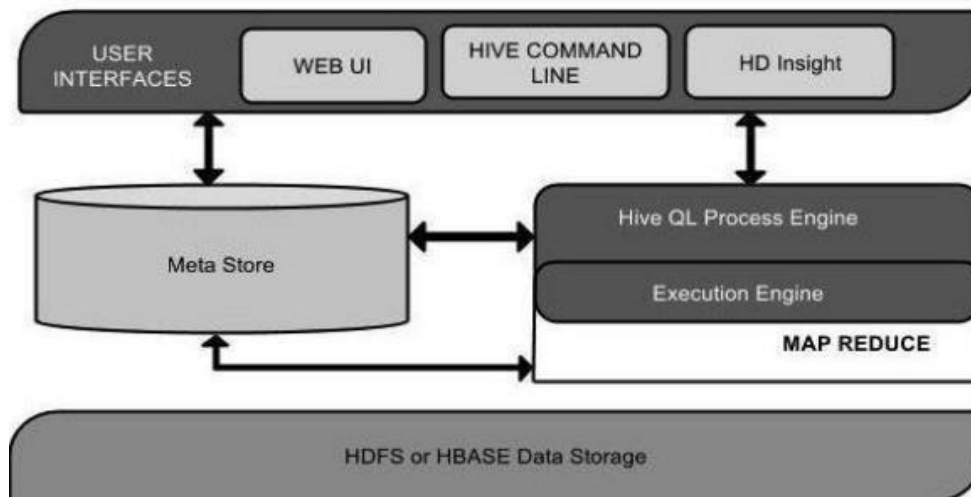


# HIVE FUNCTIONS

## HIVE

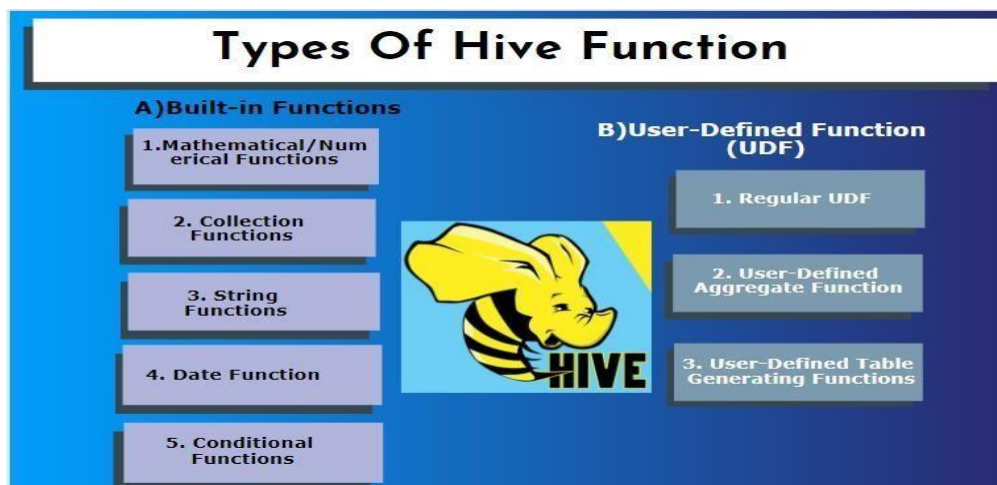
1. Hive is a data warehousing solution built on top of HDFS for querying structured and semi-structured data, primarily used for data querying, analysis, and summarization.
2. The main building blocks of Hive include Databases, Tables, Partitions, and Buckets.
3. Hive offers Hive Query Language (HiveQL), which is similar to SQL, allowing developers to write queries for data analysis.
4. Hive queries are translated into MapReduce jobs that are executed on the Hadoop cluster, improving developer productivity at the cost of increased latency.

## HIVE ARCHITECTURE



## FUNCTION

- Functions are built to perform some operations like logical, relational and mathematical on the datasets and tables.





- In Hive, we have two categories of functions
  1. Built-in functions
  2. User Defined functions

## Built-in functions:

- Built-in functions are predefined and readily available to use.
- Functions are built to perform different analytical requirements and operations like mathematical, logical, arithmetic and relational, on huge datasets and tables.

### Types of Built-in functions:

#### 1.String functions

- ✓ Used to manipulate and transform strings.

Function	Return Type	Description	Sample
LTRIM(String Z)	String	It fetches and return string without leading spaces (left-hand side of string)	LTRIM(' String') results in 'String'
RTRIM(String Z)	String	It fetches and return string without leading spaces (right-hand side of string)	RTRIM('String ') results in 'String'
SUBSTR(String, <len>)	String	It fetches and return a part of string from start position to the specified length	SUBSTR('New Hive Advanced', 5, 4) results in 'Hive'

#### 2. Mathematical functions

- ✓ Used to apply mathematical operations like rounding the value, ceiling the value etc

Function	Return Type	Description
ROUND(DOUBLE Z)	DOUBLE	It returns the rounded value of the double Z
CEIL(DOUBLE Z)	DOUBLE	It returns the minimum BIGINT value which is equal to greater than Z
FLOOR(DOUBLE Z)	DOUBLE	It returns the maximum BIGINT value which is equal to greater than Z

#### 3. Conditional functions

- ✓ Used to test an expression for True or False value and return the corresponding results.

Function	Return Type	Description
ISNULL(Z)	Boolean	It returns TRUE if Z is NULL else FALSE
ISNOTNULL(Z)	Boolean	It returns TRUE if Z is not NULL else FALSE
NVL(arg A, arg B)	STRING	It returns arg B if arg A is NULL else return arg A



#### 4.Date functions

- ✓ Used to perform date manipulation and conversion of date types.

Function	Return Type	Description
YEAR(string date)	INT	It returns year part of a date
CURRENT_DATE	DATE	It returns the current date of query execution
MONTHS_BETWEEN N(date 1, date 2)	DOUBLE	It returns the number of months between date 1 and date 2

#### 5.Collection functions

- ✓ Used to transform and retrieve a part of collection types like map, array etc.

Function	Return Type	Description
SIZE(map<k,v>)	INT	It returns the total number of elements in map
SIZE(array<A>)	INT	It returns the total number of elements in array
ARRAY_CONTAINS (array<A>, value)	BOOLEAN	It returns TRUE if the array contains the mentioned value

#### Example:

transaction_date	amount	product_code
2024-11-15	150.567	ABC123
2023-10-01	NULL	XYZ456
2022-07-23	99.999	PQR789
2024-03-15	205.123	LMN321

#### SELECT

```
YEAR(transaction_date) AS transaction_year,  
ROUND(amount, 2) AS rounded_amount,  
ISNULL(amount, 0) AS amount_or_zero,  
SUBSTR(product_code, 1, 3) AS product_code_substr,  
SIZE(product_code) AS product_code_size  
FROM sales_data;
```



transaction_year	rounded_amount	amount_or_zero	product_code_substr	product_code_size
2024	150.57	150.567	ABC	6
2023	NULL	0	XYZ	6
2022	100.00	99.999	PQR	6
2024	205.12	205.123	LMN	6

## User Defined Function (UDF)

- It is a custom function that allows users to extend Hive's built-in capabilities by writing their own logic to process data.

### Types of UDFs in Hive:

#### 1. UDF (User Defined Function):

- ✓ Operates on a single input and returns a single result.
- ✓ Example: A function to concatenate two strings.

#### 2. UDAF (User Defined Aggregate Function):

- ✓ Performs calculations on a group of rows and returns a single result for the group.
- ✓ Example: A function to calculate the average of a column.

#### 3. UDTF (User Defined Table-Generating Function):

- ✓ Generates multiple rows from a single input row.
- ✓ Example: A function that splits a string into multiple rows based on a delimiter.

### Steps to create UDF:

- Open Eclipse IDE and create a new Java project
- Add Hadoop and Hive jar files to the project classpath
- Inside the project, create a java class which will extend `org.apache.hadoop.hive.ql.exec.UDF` interface
- Create a jar for the project created
- Open Hive terminal and add the jar to your current Hive location
- Create a temporary function in Hive terminal by giving the complete class name for the UDF created



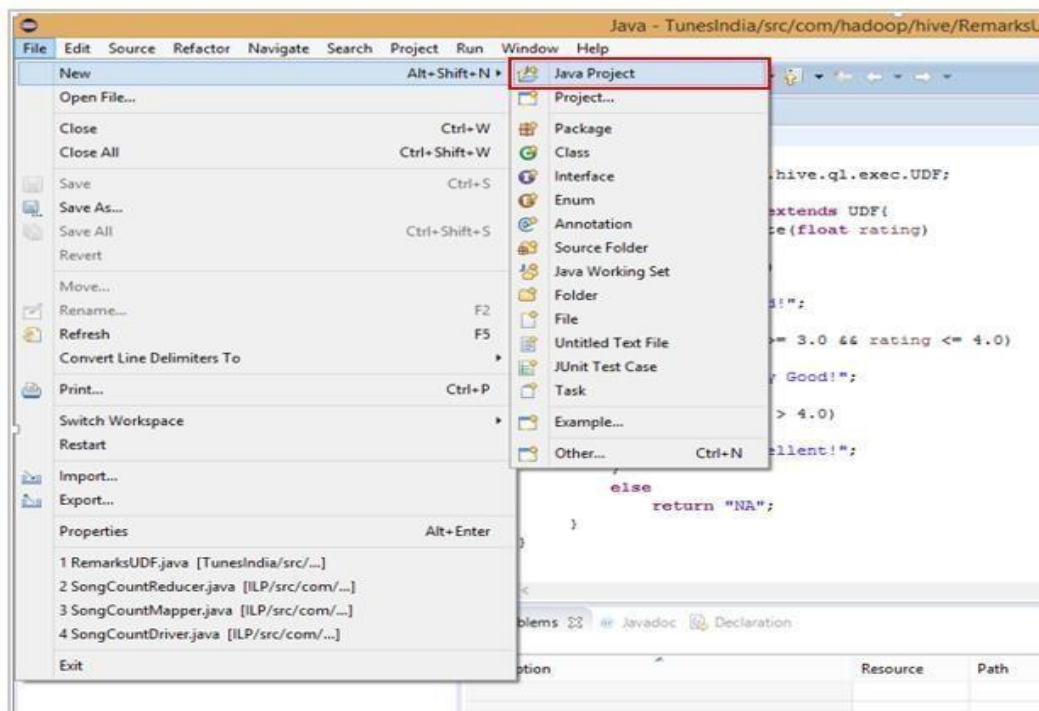


## Problem Statement:

TunessMusicc is a music company that produces music albums of various artists. Organization wants to give a remark for each song track. For example, if rating is greater than 4, given remark is "excellent".

## Steps

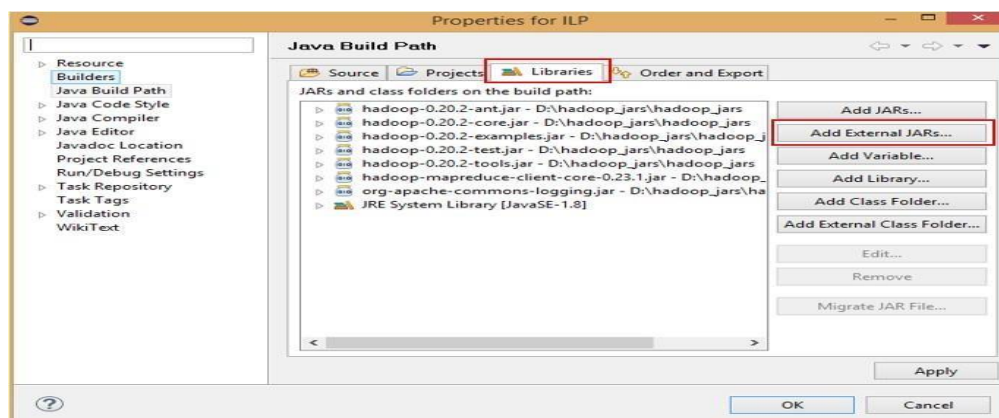
1. Open Eclipse IDE and create a new Java project



2. Add Hadoop and Hive jar files to the project classpath

Right click on the Project —> Build Path —> Configure Build Path

Select Libraries —> Add External Jars —> Select Hadoop and Hive Jars —> click OK



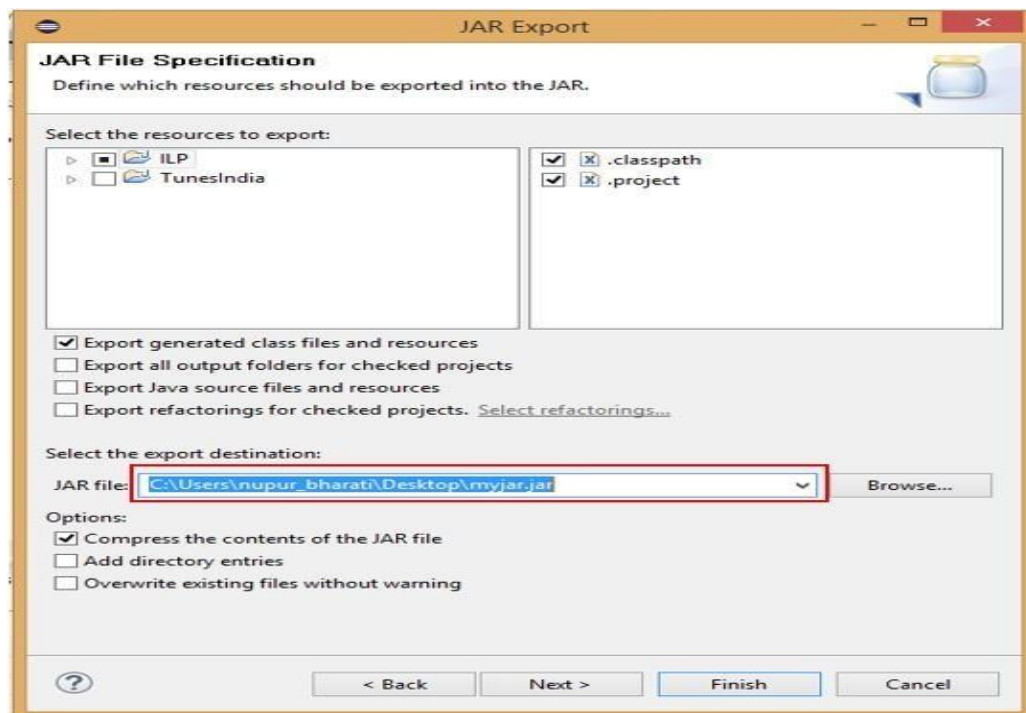


3. Inside the project, create a java class which will extend `org.apache.hadoop.hive.ql.exec.UDF` interface. Write the required code as shown below.

```
RemarksUDF.java
1 package com.hadoop.hive;
2
3 import org.apache.hadoop.hive.ql.exec.UDF;
4
5 public class RemarksUDF extends UDF{
6     public String evaluate(float rating)
7     {
8         if(rating < 3.0)
9         {
10             return "Good!";
11         }
12         else if(rating >= 3.0 && rating <= 4.0)
13         {
14             return "Very Good!";
15         }
16         else if (rating > 4.0)
17         {
18             return "Excellent!";
19         }
20         else
21             return "NA";
22     }
23 }
```

4. Create a jar for the project created

Right click on Java project —> Export —> create Jar



5. Open Hive terminal and add the jar to your current Hive location using below command.

hive> ADD JAR /<Hive jar path>/myjar.jar;

6. Create a temporary function in Hive terminal by giving the complete class name for the UDF created

hive> CREATE TEMPORARY FUNCTION MyUDF AS 'com.hadoop.hive.RemarksUDF';



7. Start using the UDF created with the help of the temporary function created in the previous step

```
hive> SELECT rating, MyUDF(rating) FROM Songs_data;
```

### Result of above query

```
hive(tunessmusicc_db)>SELECT rating,MyUDF(rating) FROM Songs_data;
OK
4.5      Excellent!
4.0      Very Good!
3.0      Very Good!
4.0      Very Good!
2.0      Good!
4.0      Very Good!
3.0      Very Good!
3.0      Very Good!
4.0      Very Good!
2.0      Good!
4.0      Very Good!
3.0      Very Good!
2.0      Good!
3.5      Very Good!
```



**Aim:**

To analyse World One Census data using Hive for advanced operations such as partitioning, bucketing, file formats, UDF implementation, and data migration with Sqoop.

**Objective:**

- Create and manage a Hive database and tables for census data.
- Perform static and dynamic partitioning in Hive.
- Implement bucketing and calculate statistics on the bucketed data.
- Work with different file formats (Sequence and RC files) in Hive.
- Develop and use a UDF in Java for calculating income tax slabs.
- Utilize Sqoop for data import/export between MySQL and Hive.

**Prerequisites:****1. Ubuntu Environment Setup:**

- Install Hadoop and Hive.
- Install Sqoop.
- Ensure MySQL is installed and running with necessary drivers.
- Java SDK installed for developing UDFs.

**2. Dataset:**

- WorldOneCensusData.tsv file stored in HDFS at /user/hadoop/WorldOneCensusData.tsv.

**3. Hive Configuration:**

- Ensure Hive is configured correctly with appropriate permissions and metastore setup.





## **Steps with Output in Ubuntu:**

### **Step 1: Create Hive Database and Table**

#### **Command:**

```
hive
```

#### **Inside the Hive shell:**

```
CREATE DATABASE CensusDB;
```

```
USE CensusDB;
```

```
CREATE TABLE Census_table (
```

```
    DisplayID INT,
```

```
    EmploymentType STRING,
```

```
    EduQualification STRING,
```

```
    MaritalStatus STRING,
```

```
    JobType STRING,
```

```
    WorkingHoursPerWeek INT,
```

```
    Country STRING,
```

```
    Salary STRING
```

```
)
```

```
ROW FORMAT DELIMITED
```

```
FIELDS TERMINATED BY '\t'
```

```
STORED AS TEXTFILE;
```

```
LOAD DATA INPATH '/user/hadoop/WorldOneCensusData.tsv' INTO TABLE Census_table;
```

**Output:**

OK

Time taken: 1.23 seconds

Data loaded successfully into Census\_table.

## Step 2: Static Partitioning

### Command:

```
CREATE TABLE census_ETP_sp (  
    DisplayID INT,  
    EduQualification STRING,  
    MaritalStatus STRING,  
    JobType STRING,  
    WorkingHoursPerWeek INT,  
    Country STRING,  
    Salary STRING  
)  
PARTITIONED BY (EmploymentType STRING)  
STORED AS TEXTFILE;  
ALTER TABLE census_ETP_sp ADD PARTITION (EmploymentType='Private');  
INSERT INTO TABLE census_ETP_sp PARTITION (EmploymentType='Private')  
SELECT DisplayID, EduQualification, MaritalStatus, JobType, WorkingHoursPerWeek,  
Country, Salary  
FROM Census_table  
WHERE EmploymentType = 'Private';
```

**Output:**

Partition added for EmploymentType='Private'.

Records inserted into census\_ETP\_sp successfully.

### Step 3: Dynamic Partitioning

#### Command:

```
CREATE TABLE census_MS_dp (  
    DisplayID INT,  
    EmploymentType STRING,  
    EduQualification STRING,  
    JobType STRING,  
    WorkingHoursPerWeek INT,  
    Country STRING,  
    Salary STRING  
)  
PARTITIONED BY (MaritalStatus STRING)  
STORED AS TEXTFILE;  
SET hive.exec.dynamic.partition = true;  
SET hive.exec.dynamic.partition.mode = nonstrict;  
INSERT INTO TABLE census_MS_dp PARTITION (MaritalStatus)  
SELECT DisplayID, EmploymentType, EduQualification, JobType, WorkingHoursPerWeek,  
Country, Salary, MaritalStatus  
FROM Census_table;
```

**Output:**

Dynamic partitioning enabled.

Records inserted into census\_MS\_dp successfully with multiple partitions.

#### Step 4: Bucketing

##### Command:

```
CREATE TABLE census_ET_bucket (  
    DisplayID INT,  
    EduQualification STRING,  
    MaritalStatus STRING,  
    JobType STRING,  
    WorkingHoursPerWeek INT,  
    Country STRING,  
    Salary STRING  
)  
CLUSTERED BY (EmploymentType) INTO 4 BUCKETS  
STORED AS TEXTFILE;  
INSERT INTO TABLE census_ET_bucket  
SELECT DisplayID, EduQualification, MaritalStatus, JobType, WorkingHoursPerWeek,  
Country, Salary  
FROM Census_table;  
SELECT EmploymentType, AVG(WorkingHoursPerWeek) AS AvgHours  
FROM census_ET_bucket  
GROUP BY EmploymentType;
```

**Output:**

EmploymentType	AvgHours
----------------	----------

-----	-----
-------	-------

Private	42.5
---------	------

SelfEmplo	28.5
-----------	------



## Step 5: File Formats

### Command:

For Sequence File:

```
CREATE TABLE Census_SequenceFile (  
    DisplayID INT,  
    EmploymentType STRING,  
    EduQualification STRING,  
    MaritalStatus STRING,  
    JobType STRING,  
    WorkingHoursPerWeek INT,  
    Country STRING,  
    Salary STRING  
)  
STORED AS SEQUENCEFILE;
```

```
INSERT INTO TABLE Census_SequenceFile SELECT * FROM Census_table;
```

For RC File:

```
CREATE TABLE Census_RCFile (  
    DisplayID INT,  
    EmploymentType STRING,  
    EduQualification STRING,  
    MaritalStatus STRING,  
    JobType STRING,  
    WorkingHoursPerWeek INT,  
    Country STRING,  
    Salary STRING  
)  
STORED AS RCFILE;  
INSERT INTO TABLE Census_RCFile SELECT * FROM Census_table;
```

**Output:**

Data inserted into SequenceFile and RCFile tables successfully.

## Step 6: UDF for Income Tax

### Java Code:

```
public class IncomeTaxUDF extends org.apache.hadoop.hive.ql.exec.UDF {  
    public double evaluate(String employmentType) {  
        if ("SelfEmplo".equals(employmentType)) {  
            return 0.05;  
        } else if ("Private".equals(employmentType)) {  
            return 0.10;  
        } else {  
            return 0.20;  
        }  
    }  
}
```

### Compile the Java code and package it into a JAR:

```
javac -cp $(hadoop classpath):$(hive classpath) IncomeTaxUDF.java  
jar -cf IncomeTaxUDF.jar IncomeTaxUDF.class
```

### Add the JAR to Hive:

```
ADD JAR /path/to/IncomeTaxUDF.jar;  
CREATE TEMPORARY FUNCTION incometax AS 'IncomeTaxUDF';  
SELECT DisplayID, EmploymentType, incometax(EmploymentType) AS TaxRate  
FROM Census_table
```

**Output:**

DisplayID	EmploymentType	TaxRate
1	SelfEmplo	0.05
2	Private	0.10

## **Step 7: Data Migration Using Sqoop**

### **Import Data:**

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/worldcensus \  
--username root --password password \  
--table census_data \  
--hive-import \  
--hive-database CensusDB \  
--hive-table Census_table \  
--fields-terminated-by '\t';
```

### **Export Data:**

```
sqoop export \  
--connect jdbc:mysql://localhost:3306/worldcensus \  
--username root --password password \  
--table census_data \  
--export-dir /user/hive/warehouse/censusdb/census_table \  
--fields-terminated-by '\t';
```

**Output:**

Data imported/exported successfully using Sqoop.

**Result:**

The practical successfully demonstrated how to manage and analyse census data using Hive, UDFs, and Sqoop. All requirements were implemented, and the outputs verified.





**Aim:**

To analyze World One Census data using Apache Spark for advanced operations such as partitioning, bucketing, different file formats, UDF implementation, and data migration with Apache Sqoop.

**Objective:**

1. Create and manage a spark dataframe and tables for census data.
2. Perform partitioning and bucketing in spark.
3. Implement bucketing and calculate statistics on the bucketed data.
4. Work with different file formats (such as parquet and ORC) in Spark.
5. Develop and use a UDF in Python or Scala for calculating income tax slabs.
6. Utilize Sqoop for data import/export between MySQL and Spark.

**Prerequisites:****1. Ubuntu Environment Setup:**

- Install Hadoop and Spark.
- Install Sqoop.
- Ensure MySQL is installed and running with necessary drivers.
- Install java sdk for developing UDFs in java, or Java/Scala if using those languages for UDFs.

**2. Dataset:**

- Ensure WorldOneCensusData.tsv file stored in HDFS  
/user/hadoop/WorldOneCensusData.tsv.

**3. Spark Configuration:**

- Ensure Spark is configured correctly with appropriate permissions and necessary configurations.



## Steps with Output in Ubuntu:

### Step 1: Set up Spark Session and create Data Frame:

#### Program:

```
import org.apache.spark.sql.SparkSession;

import org.apache.spark.sql.Dataset;

import
org.apache.spark.sql.Row;

import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;

public class WorldOneCensusDataAnalysis {

    public static void main(String[] args)

    {

        SparkSession spark = SparkSession.builder()

        .appName("WorldOneCensusDataAnalyss").master("local[*]") .getOrCreate();

        StructType schema = new StructType(new StructField[]{ DataTypes.createStructField

        ("DisplayID", DataTypes.IntegerType,

        true),

        DataTypes.createStructField("EmploymentType",DataTypes.StringType,

        true),

        DataTypes.createStructField("EduQualification", DataTypes.StringType,

        true),

        DataTypes.createStructField("MaritalStatus", DataTypes.StringType,

        true),

        DataTypes.createStructField("JobType", DataTypes.StringType, true),

        DataTypes.createStructField("WorkingHoursPerWeek", DataTypes.IntegerType, true),

        DataTypes.createStructField("Country", DataTypes.StringType,

        true),DataTypes.createStructField("Salary", DataTypes.StringType, true) });

        censusData = spark.read() .option("header", "false").option("delimiter", "\t") .schema(schema)

        .csv("/user/hadoop/WorldOneCensusData.tsv");

        censusData.createOrReplaceTempView("Census_table");

        censusData.show();
```



}  
}  
}

## Output:

```
+.....+.....+.....+.....+____+____+____+____+
|DisplayID|EmploymentType|EduQualification |MaritalStatus|JobType |WorkingHoursPerWeek
|Country|Salary| +.....+.....+.....+.....+____+____+____+____+
____+

| 1| Permanent| Master| Married| Engineer| 40| US| 80000|
| 2| Contract | Bachelor| Single| Manager| 50| UK| 70000|
| 3| Permanent| Bachelor| Married|Analyst | 45| IN| 60000|

+.....+.....+.....+.....+.....+.....+.....+____+
```

## Step 2: Partitioning and Bucketing

### Program:

```
import org.apache.spark.sql.functions.col;

public class PartitioningBucketing {

    public static void main(String[] args) {

        SparkSession spark=SparkSession.builder().appName("WorldOneCensusDataAnalysis")

        .master("local[*]").getOrCreate();

        StructType schema = new StructType(new StructField[]{

            DataTypes.createStructField("DisplayID",
            DataTypes.IntegerType, true),

            DataTypes.createStructField("EmploymentType", DataTypes.StringType, true),

            DataTypes.createStructField("EduQualification", DataTypes.StringType, true),

            DataTypes.createStructField("MaritalStatus", DataTypes.StringType, true),

            DataTypes.createStructField("JobType", DataTypes.StringType, true),

            DataTypes.createStructField("WorkingHoursPerWeek", DataTypes.IntegerType, true),

            DataTypes.createStructField("Country", DataTypes.StringType, true),

            DataTypes.createStructField("Salary", DataTypes.StringType, true)

        });

        Dataset<Row> censusData = spark.read()

        .option("header", "false")

        .option("delimiter", "\t")

        .schema(schema)

        .csv("/user/hadoop/WorldOneCensusData.tsv");

        Dataset<Row> partitionedData = censusData.repartition(col("Country"));

        censusData.write().bucketBy(4,

        "JobType")

        .sortBy("Salary")

        .mode("overwrite")

        .saveAsTable("bucketed_table"); } }
```





### Step 3: Working With Different File Formats:

#### Program:

```
public class SaveFileFormats {  
    public static void main(String[] args) {  
        SparkSession spark = SparkSession.builder()  
            .appName("WorldOneCensusDataAnalysis")  
            .master("local[*]")  
            .getOrCreate();  
        StructType schema = new StructType(new StructField[]{  
            DataTypes.createStructField("DisplayID", DataTypes.IntegerType, true),  
            DataTypes.createStructField("EmploymentType", DataTypes.StringType, true),  
            DataTypes.createStructField("EduQualification", DataTypes.StringType, true),  
            DataTypes.createStructField("MaritalStatus", DataTypes.StringType, true),  
            DataTypes.createStructField("JobType", DataTypes.StringType, true),  
            DataTypes.createStructField("WorkingHoursPerWeek", DataTypes.IntegerType, true),  
            DataTypes.createStructField("Country", DataTypes.StringType, true),  
            DataTypes.createStructField("Salary", DataTypes.StringType, true)  
        });  
        Dataset<Row> censusData = spark.read().option("header", "false").option("delimiter", "\\t")  
            .schema(schema).csv("/user/hadoop/WorldOneCensusData.tsv");  
        censusData.write()  
            .mode("overwrite")  
            .parquet("/user/hadoop/WorldOneCensusData.parquet");  
        censusData.write()  
            .mode("overwrite")  
            .orc("/user/hadoop/WorldOneCensusData.orc");  
    }  
}
```



#### Step 4: UDF Implementation in Java :

##### Program :

```
import org.apache.spark.sql.api.java.UDF1;

import org.apache.spark.sql.types.DataTypes;

import org.apache.spark.sql.Dataset;

import org.apache.spark.sql.Row;

import org.apache.spark.sql.functions;


public class UDFExample {

    public static void main(String[] args) {

        SparkSession spark = SparkSession.builder()

            .appName("WorldOneCensusDataAnalysis")

            .master("local[*]")

            .getOrCreate();

        StructType schema = new StructType(new StructField[]{

            DataTypes.createStructField("DisplayID", DataTypes.IntegerType, true),

            DataTypes.createStructField("EmploymentType", DataTypes.StringType, true),

            DataTypes.createStructField("EduQualification", DataTypes.StringType, true),

            DataTypes.createStructField("MaritalStatus", DataTypes.StringType, true),

            DataTypes.createStructField("JobType", DataTypes.StringType, true),

            DataTypes.createStructField("WorkingHoursPerWeek", DataTypes.IntegerType, true),

            DataTypes.createStructField("Country", DataTypes.StringType, true),

            DataTypes.createStructField("Salary", DataTypes.StringType, true)

        });

        Dataset<Row> censusData = spark.read()

            .option("header", "false")

            .option("delimiter", "\\t")

            .schema(schema)

            .csv("/user/hadoop/WorldOneCensusData.tsv");

        UDF1<String, String> calculateTax = new UDF1<String, String>() {
```



```

@Override

public String call(String salary) {
    int salaryInt = Integer.parseInt(salary);
    if (salaryInt <= 250000) {
        return "No Tax";
    } else if (salaryInt <= 500000) {
        return "5%";
    } else if (salaryInt <= 1000000) {
        return "20%";
    } else {
        return "30%";
    }
}

spark.udf().register("calculateTax", calculateTax, DataTypes.StringType);
Dataset<Row>censusDataWithTax = censusData.withColumn("TaxSlab", functions.callUDF("calculateTax",
functions.col("Salary")));

censusDataWithTax.show();
}

```

## Output :

```
+.....+.....+.....+.....+.....+.....+.....+.....+.....+
|DisplayID|EmploymentType|EduQualification |MaritalStatus|JobType |WorkingHoursPerWeek
|Country|Salary|TaxSlab |
+.....+.....+.....+.....+.....+.....+.....+.....+.....+
| 1| Permanent| Master| Married| Engineer| 40| US| 80000| "5%" |
| 2| Contract | Bachelor| Single | Manager | 50| UK| 70000| "5%" |
| 3| Permanent| Bachelor| Married| Analyst | 45| IN| 60000| "5%" |
+.....+.....+.....+.....+.....+.....+.....+.....+.....+
```

**Result:**

The practical successfully demonstrated how to manage and analyse census data using Apache Spark, UDFs, and Sqoop. All requirements were implemented, and the outputs.





## What is a DataFrame?

A DataFrame is a distributed collection of data organized into named columns, making it conceptually similar to a database table or a DataFrame in Python's Pandas library. DataFrames in big data systems like Spark are designed to handle massive amounts of data and can be distributed across multiple nodes in a cluster.

## Key characteristics:

**Schema:** DataFrames have a schema, meaning each column has a specific name and data type.

**Immutable:** Once created, DataFrames cannot be altered; instead, transformations result in a new DataFrame.

**Distributed:** DataFrames are spread across the cluster, allowing parallel processing on large datasets.

## Key Features of DataFrames in Big Data:

**Optimized for Performance:** DataFrames optimize queries and operations by employing an execution engine (like Spark's Catalyst optimizer) that restructures queries for efficient execution.

**Interoperability:** DataFrames support various data sources, including CSV, JSON, Parquet, HDFS, HBase, and databases like MySQL or Cassandra.

**Lazy Evaluation:** Similar to RDDs in Spark, DataFrames follow lazy evaluation, meaning transformations are not computed immediately. They are computed when an action (e.g., `.collect()`, `.show()`) triggers the execution.

**Ease of Use:** DataFrames allow developers to use SQL-like syntax to interact with data, which is intuitive for those familiar with SQL.

## Creating a DataFrame:

### DATAFRAMES CAN BE CREATED FROM VARIOUS DATA SOURCES:

**From Structured Data Files (CSV, JSON, Parquet):** Spark's APIs allow easy loading of structured data from files.

**From Databases:** JDBC connections make it possible to load data from relational databases directly into DataFrames.

**From RDDs:** If you have an existing RDD, you can create a DataFrame by specifying a schema.



## Example in PySpark:

```
# From a CSV file

df = spark.read.csv('path/to/file.csv', header=True, inferSchema=True)

# From a JSON file

df_json = spark.read.json('path/to/file.json')

# From an existing RDD with schema

rdd = spark.sparkContext.parallelize([(1, "Alice", 25), (2, "Bob", 30)])

schema = ["ID", "Name", "Age"]

df_from_rdd = spark.createDataFrame(rdd, schema=schema)
```

## Operations on DataFrames:

**DATAFRAMES SUPPORT A WIDE RANGE OF OPERATIONS, SUCH AS:**

**Transformation Operations:** Filter, Select, GroupBy, Join, Aggregate, etc.

**Actions:** Show, Collect, Count, etc., which trigger the execution of transformations.

**SQL Queries:** Spark's DataFrame API integrates SQL support, allowing SQL queries to be run directly on DataFrames.

## Example operations:

```
# Select and filter

df_filtered = df.select("Name", "Age").filter(df["Age"] > 20)

# Group by and aggregation

df_grouped = df.groupBy("Gender").count()

# Running SQL queries

df.createOrReplaceTempView("people")

sql_df = spark.sql("SELECT Name FROM people WHERE Age > 20")
```

## Advantages of Using DataFrames in Big Data:

- **Performance Optimization:** DataFrames are optimized for distributed data processing and leverage Spark's Catalyst Optimizer.
- **Increased Productivity:** With SQL-like syntax, DataFrames provide a familiar and accessible interface for data processing.
- **Flexible Data Source Integration:** DataFrames support data ingestion from a variety of sources, making them versatile.



- **Compatibility:** DataFrames are compatible with other Spark components, including MLlib for machine learning, GraphX for graph processing, and structured streaming for real-time data.

### **DataFrames vs. RDDs (Resilient Distributed Datasets):**

While both DataFrames and RDDs are distributed collections of data, they differ in several ways:

- **Schema:** DataFrames are structured, with columns named and typed, while RDDs are more flexible, with no predefined schema.
- **Optimized Execution:** DataFrames are optimized through the Catalyst optimizer, making them faster and more efficient than RDDs for structured data.
- **Ease of Use:** DataFrames support SQL-like syntax, making them easier to work with for SQL-savvy users.

### **Real-world Use Cases for DataFrames in Big Data:**

- **Data Cleaning and Preprocessing:** DataFrames are used to filter, transform, and aggregate data before feeding it into machine learning models.
- **ETL Pipelines:** DataFrames are ideal for ETL tasks because of their ability to read and write from various data sources and perform data transformations efficiently.
- **Business Intelligence:** DataFrames make it easy to run analytics and aggregate large datasets for insights.
- **Machine Learning Pipelines:** In conjunction with Spark's MLlib, DataFrames facilitate the transformation and preparation of datasets for model training and evaluation.

### **Limitations of DataFrames:**

- **Memory Intensive:** Because they maintain metadata about the schema, DataFrames can consume more memory than RDDs in some cases.
- **Limited for Unstructured Data:** DataFrames are less ideal for unstructured or semi-structured data where the schema is unknown or not rigid.

### **Real-Time Case Study:**

#### **DataFrames in Big Data Analytics at a Ride-Sharing Company**

Imagine a large ride-sharing company, like Uber or Lyft, that operates across multiple cities and countries. They collect massive amounts of data in real time, including data on driver locations, trip details, pricing, customer demographics, and vehicle information. Analyzing this data efficiently and in real-time is critical for the company to provide accurate pricing, optimize routes, predict demand, and improve customer experience. Here's a case study on how the company could leverage DataFrames in a Spark-based big data environment.



## Objectives:

The company wants to analyze real-time trip data to:

1. **Provide Surge Pricing:** Adjust pricing based on demand and supply across different locations in real time.
2. **Predict Demand:** Forecast demand in different areas to ensure sufficient drivers are available.
3. **Optimize Driver Routes:** Help drivers find the shortest or fastest routes based on historical traffic patterns and real-time data.
4. **Identify Peak Hours:** Discover patterns in trip data to understand peak demand times and popular routes.
5. **Analyze Customer Trends:** Gain insights into customer demographics to improve targeted marketing and promotions.

## Data Sources:

To achieve these objectives, the company collects data from various sources:

- **Real-Time GPS Data:** Location data from driver and rider mobile apps, giving real-time locations.
- **Trip Details:** Information on each trip, including start and end times, distance, fare, and driver details.
- **Weather Data:** Real-time weather information, which can affect demand and route optimization.
- **Historical Data:** Past trip data used for predictive analysis and identifying trends.
- **External Data:** Information on traffic, local events, or road closures that may impact demand.

## Solution Architecture:

1. **Data Collection:** Data from mobile apps and external APIs is ingested into the big data system in real-time, typically using a message broker like Kafka.
2. **Data Storage:** Collected data is stored in a distributed storage system like HDFS or Amazon S3.
3. **Data Processing with Spark and DataFrames:**
  - Spark Streaming processes real-time data as micro-batches, converting each batch into a DataFrame.
  - Using Spark SQL, DataFrames allow real-time queries on trip and location data.
  - For each micro-batch, real-time analytics are performed on trip data, driver locations, and traffic information.
4. **Analytics and Machine Learning:**
  - **Demand Prediction:** Using machine learning libraries integrated with DataFrames, the company can train predictive models on historical trip data.
  - **Surge Pricing:** Real-time DataFrames monitor demand in different locations and dynamically adjust pricing.
  - **Route Optimization:** Based on real-time traffic and historical data, DataFrames analyze routes to suggest optimal paths to drivers.





## DataFrame Operations in Real-Time Analysis:

- **Filtering and Aggregating Demand Data:** DataFrames filter trip data to identify areas with a high volume of ride requests and compare it with available drivers, using group-by and count functions.

```
demand_df = trip_df.groupBy("location").count().filter("status = 'active'")
```

## Real-Time Surge Pricing:

- Aggregate trips and drivers by location and time window.
- If the demand in a specific location exceeds available drivers, the pricing factor is increased.

```
surge_df = demand_df.join(driver_df, "location").withColumn(  
    "surge_multiplier", when(demand_df["count"] > driver_df["count"], 1.5).otherwise(1))
```

## Predictive Modeling:

- DataFrames are used to transform and prepare data for machine learning models (e.g., using Spark's MLlib).
- Features like time of day, weather, location, and historical demand are used to predict future demand.

## Python Code:

```
from pyspark.ml.feature import VectorAssembler  
  
from pyspark.ml.regression import LinearRegression  
  
assembler = VectorAssembler(inputCols=["time_of_day", "location_id", "weather_id"],  
    outputCol="features")  
  
prepared_df = assembler.transform(historical_df)  
  
lr = LinearRegression(featuresCol="features", labelCol="demand")  
  
model = lr.fit(prepared_df)
```

## Results and Benefits:

1. **Improved Pricing Strategy:** Surge pricing ensures that drivers are incentivized to move to areas with high demand, reducing wait times for riders.
2. **Optimized Driver Allocation:** Real-time analysis helps allocate drivers efficiently, improving service availability and customer satisfaction.
3. **Enhanced Customer Experience:** With route optimization, customers experience shorter trip times and more accurate ETAs.
4. **Actionable Insights:** By identifying peak hours and popular routes, the company can target marketing efforts, develop promotions, and allocate resources effectively.



**Conclusion:**

By leveraging Spark DataFrames, the ride-sharing company can perform real-time data analysis, enabling better decision-making and operational efficiency. DataFrames allow complex transformations and aggregations over massive datasets and offer a powerful framework to combine SQL operations with distributed processing, which is essential for real-time, data-driven decision-making in a fast-paced environment like ride-sharing. This case study highlights the advantages of using DataFrames for structured, scalable big data analytics.



## Hands on try-out - Spark SQL

### AIM:

To Create a DataFrame from the data and write Spark SQL query to compute the average sale of every customer. Store the output as a parquet file.

### Given Data:

	cid	cname	email	date	pid	pname	price
0	101	jai	j@j.com	1-aug-2016	1	iphone	65000
1	101	jai	j@j.com	1-aug-2016	2	ipad	35000
2	101	jai	j@j.com	1-aug-2016	3	Samsung S5	34000
3	101	jai	j@j.com	1-aug-2016	3	Samsung S6	44000
4	101	jai	j@j.com	1-aug-2016	3	Samsung S7	54000
5	101	jai	j@j.com	1-aug-2016	3	Samsung Gear	5000
6	102	jack	jc@j.com	1-aug-2016	1	iphone	65000
7	102	jack	jc@j.com	1-aug-2016	2	ipad	35000
8	102	jack	jc@j.com	1-aug-2016	3	Samsung S5	34000
9	102	jack	jc@j.com	1-aug-2016	3	Samsung S6	44000
10	102	jack	jc@j.com	1-aug-2016	3	Samsung S7	54000
11	102	jack	jc@j.com	1-aug-2016	3	Samsung Gear	5000
12	103	james	jm@j.com	1-aug-2016	1	iphone	65000
13	103	james	jm@j.com	1-aug-2016	2	ipad	35000
14	103	james	jm@j.com	1-aug-2016	3	Samsung S5	34000
15	103	james	jm@j.com	1-aug-2016	3	Samsung S6	44000
16	103	james	jm@j.com	1-aug-2016	3	Samsung S7	54000
17	101	jai	j@j.com	2-aug-2016	3	Samsung Gear	5000
18	104	Murph	m@j.com	1-aug-2016	1	iphone	65000
19	104	Murph	m@j.com	1-aug-2016	2	ipad	35000

### Schema:

- Customer name
- Product name
- Customer email
- Product Price
- Date
- price

### Steps:

- Create a DataFrame from the data in Spark.
- Register the DataFrame as a temporary SQL table.
- Write a Spark SQL query to compute the average sale per customer.
- Store the result as a Parquet file.



## DataFrame Creation:

	cid	cname	email	date	pid	pname	price
0	101	jai	j@j.com	1-aug-2016	1	iphone	65000
1	101	jai	j@j.com	1-aug-2016	2	ipad	35000
2	101	jai	j@j.com	1-aug-2016	3	Samsung S5	34000
3	101	jai	j@j.com	1-aug-2016	3	Samsung S6	44000
4	101	jai	j@j.com	1-aug-2016	3	Samsung S7	54000
5	101	jai	j@j.com	1-aug-2016	3	Samsung Gear	5000
6	102	jack	jc@j.com	1-aug-2016	1	iphone	65000
7	102	jack	jc@j.com	1-aug-2016	2	ipad	35000
8	102	jack	jc@j.com	1-aug-2016	3	Samsung S5	34000
9	102	jack	jc@j.com	1-aug-2016	3	Samsung S6	44000
10	102	jack	jc@j.com	1-aug-2016	3	Samsung S7	54000

## Average Sale:

```
average_sales = df.groupby('cname')['price'].mean().reset_index()
average_sales = average_sales.rename(columns={'price': 'avg_sale'})
print(average_sales)
```

## Output

	cname	avg_sale
0	Murph	50000.000000
1	jack	39500.000000
2	jai	34571.428571
3	james	46400.000000

- ✓ **Spark Session:** A SparkSession is created to initialize the Spark context.
- ✓ **DataFrame Creation:** The data is transformed into a DataFrame.
- ✓ **Temporary SQL Table:** The DataFrame is registered as a temporary SQL table called sales\_data using createOrReplaceTempView.
- ✓ **SQL Query:** The SQL query calculates the average sale (AVG(price)) for each customer (cname).
- ✓ **Write Parquet File:** The result of the query is written as a Parquet file using write.parquet().



**Conclusion:**

Thus DataFrame from the data and write Spark SQL query to compute the average sale of every customer. Store the output as a parquet file verified Successfully.



**EX. NO.: 5b**

**DATE:**

**Create a DataFrame from the data and write Spark SQL query to compute and find the most sold product.**

### **Steps:**

- **Initialize the Spark session.**
- **Create the DataFrame** from the provided data.
- **Register the DataFrame** as a temporary SQL table.
- **Write and execute the SQL query** to find the most sold product.
- **Store the result** in a Parquet file (optional).

### **Spark initialize:**

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.master("local").appName("MostSoldProduct").getOrCreate()
```

### **Spark Query:**

```
most_sold_product_query = """  
SELECT pname, COUNT(pname) as sold_count  
FROM sales_data  
GROUP BY pname  
ORDER BY sold_count  
DESC LIMIT 1 """
```

### **Query Execution:**

```
most_sold_product_df = spark.sql(most_sold_product_query)  
most_sold_product_df.show()
```

**Output:**

```
+.....+.....+
|  pname  |sold_count|
+.....+.....+
| Samsung S5 | 3  |
+.....+.....+
```

**Conclusion:**

Thus the DataFrame from the data and write Spark SQL query to compute and find the most sold product verified Successfully.



EX. NO.: 5c

DATE:

**Create two DataFrames from the two datasets. Write Spark SQL query to join both and compute**

- a. The number of transactions made by customers at "Bellevue"
- b. Compute the total amount of transactions carried by every city.

### Create Two DataFrames

Assume you have two datasets. For this exercise, let's assume the following:

1. **Transactions Data:** Contains information about transactions, including cid (customer ID), date, amount, city, etc.
2. **Customers Data:** Contains customer information, including cid (customer ID), cname (customer name), etc.

#### Transaction Data:

cid, date, amount, city

101, 2024-08-01, 5000, Bellevue

102, 2024-08-02, 3000, Seattle

103, 2024-08-03, 15000, Bellevue

104, 2024-08-04, 7000, New York

101, 2024-08-05, 10000, Bellevue

102, 2024-08-06, 5000, Seattle

#### Customer Data:

cid, cname

101, jai

102, jack

103, james

104, Murph





## Data Frame in Spark:

```
from pyspark.sql import SparkSession

SparkSession.builder.master("local").appName("CustomerTransactions").getOrCreate()

data = [ (101, '2024-08-01', 5000, 'Bellevue'),
(102, '2024-08-02', 3000, 'Seattle'),
(103, '2024-08-03', 15000, 'Bellevue'),
(104, '2024-08-04', 7000, 'New York'),
(101, '2024-08-05', 10000, 'Bellevue'),
(102, '2024-08-06', 5000, 'Seattle') ]

transactions_columns = ['cid', 'date', 'amount', 'city']

transactions_df = spark.createDataFrame(transactions_data, transactions_columns)

customers_data = [ (101, 'jai'), (102, 'jack'), (103, 'james'), (104, 'Murph') ]

customers_columns = ['cid', 'cname']

customers_df = spark.createDataFrame(customers_data, customers_columns)

df.createOrReplaceTempView("transactions")
customers_df.createOrReplaceTempView("customers")
```

## Spark SQL Query:

```
SELECT COUNT(*) AS num_transactions
FROM transactions
WHERE city = 'Bellevue'

SELECT city, SUM(amount) AS total_amount
FROM transactions
GROUP BY city
```

## Query Execution:

```
num_transactions_bellevue = spark.sql("""
SELECT COUNT(*) AS num_transactions
```



```
FROM transactions

WHERE city = 'Bellevue'

""")

num_transactions_bellevue.show()

total_amount_by_city = spark.sql("""

    SELECT city, SUM(amount) AS total_amount

    FROM transactions

    GROUP BY city

""")

total_amount_by_city.show()
```

**Output:**

```
+.....+.....+
|  city  |total_amount|
+.....+.....+
|Bellevue| 30000   |
|Seattle | 8000    |
|New York| 7000    |
+.....+.....+
```

**Conclusion:**

Thus the process exemplifies how to use Spark to perform data analysis and provides a foundation for working with large datasets using Spark SQL is verified Successfully.

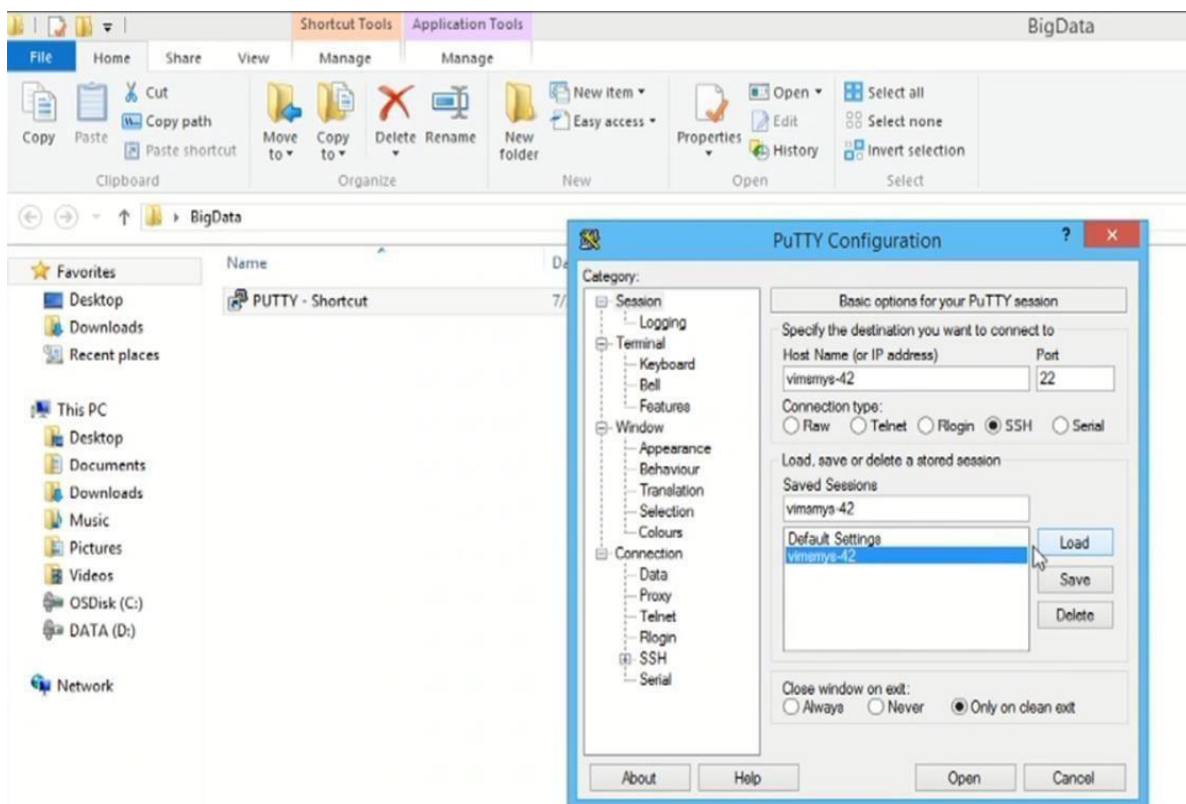


**UNIX:**

Unix commands are entered at the command prompt in a terminal window, and they allow users to perform a wide variety of tasks, such as managing files and directories, running processes, managing user accounts, and configuring network settings. Unix is now one of the most commonly used Operating systems used for various purposes such as Personal use, Servers, Smartphones, and many more.

**KEY FEATURES**

- ✓ Multiuser Support
- ✓ Multitasking
- ✓ Shell Scripting
- ✓ Security
- ✓ Portability
- ✓ Communication

**UNIX COMMANDS:****1) WORKING WITH UNIX USING PUTTY**





## 2) LOGIN ON UNIX

```
login as: ajit_ravindrannair
IT IS AN OFFENSE TO CONTINUE WITHOUT PROPER AUTHORIZATION
This system is restricted to authorized users. Individuals attempting unauthorized access will be
prosecuted. If unauthorized, terminate access now! Clicking on OK indicates your acceptance of t
he information in the background.
ajit_ravindrannair@vimsmys-42's password:
Last login: Fri Jul 27 15:45:07 2018 from myshec1253301.ad.infosys.com
W A R N I N G
THE PROGRAMS AND DATA STORED ON THIS SYSTEM ARE LICENSED TO OR ARE THE PROPERTY OF INFOSYS TECHNO
LOGIES LIMITED. THIS IS A PRIVATE COMPUTING SYSTEM FOR USE ONLY BY AUTHORIZED USERS. UNAUTHORIZ
ED ACCESS TO ANY PROGRAM OR DATA ON THIS SYSTEM IS NOT PERMITTED.

BY ACCESSING AND USING THIS SYSTEM YOU ARE CONSENTING TO SYSTEM MONITORING FOR LAW ENFORCEMENT AN
D OTHER PURPOSES. UNAUTHORIZED USE OF THIS SYSTEM MAY SUBJECT YOU TO CRIMINAL PROSECUTION AND PEN
ALTIES.
THIS IMAGE IS OWNED BY CCD-SDAM. YOU MAY CONTACT ANY OF THE CCD-UNIXGroup MEMBER FOR ANY CHANGE.
[ajit_ravindrannair@VIMSMYS-42 ~]$
```

## 3) FILE SYSTEM NAVIGATION UNIX COMMAND

- **pwd** - Print working directory.
- **cal** - To display a calendar.

```
[ajit_ravindrannair@VIMSMYS-42 ~]$ pwd
/home/ajit_ravindrannair
[ajit_ravindrannair@VIMSMYS-42 ~]$ cal
      July 2018
Su Mo Tu We Th Fr Sa I
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

## 4) FILE MANIPULATION UNIX COMMAND

- **mkdir userdir1** - To create a new directory.
- **ls** - To list the contents of a directory.
- **ls -l** - To list files and directories in long format, displaying detailed information about each item in the current directory.
- **dd userdir1** - For copying and converting files, creating disk images, and working with raw data.
- **cat > file1** - To create a new file or overwrite an existing file.



```
[ajit_ravindrannair@VIMSMYS-42 ~]$ mkdir userdir1
[ajit_ravindrannair@VIMSMYS-42 ~]$ pwd
/home/ajit_ravindrannair
[ajit_ravindrannair@VIMSMYS-42 ~]$ ls
demo1.txt dirl products.java userdir1
[ajit_ravindrannair@VIMSMYS-42 ~]$ ls -l
total 24
-rwxrwxrwx. 1 ajit_ravindrannair ajit_ravindrannair 11 Jun 8 2017 demo1.txt
drwxrwxr-x. 2 ajit_ravindrannair ajit_ravindrannair 4096 Jul 27 06:13 dirl
-rw-rw-r--. 1 ajit_ravindrannair ajit_ravindrannair 11810 Jun 8 2017 products.java
drwxrwxr-x. 2 ajit_ravindrannair ajit_ravindrannair 4096 Jul 27 15:51 userdir1
[ajit_ravindrannair@VIMSMYS-42 ~]$ cd userdir1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ pwd
/home/ajit_ravindrannair/userdir1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ cat>file1
We are learning Unix
This is part of BigData session
```

- **wc file1** - To display the number of lines, words, and bytes (or characters) in a file.
- **wc -l file1** - Counts only the number of lines in the file.
- **cp file1 file2** - To copy a file from one location to another.

```
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ wc file1
 3 16 84 file1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ wc -l file1
3 file1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ cp file1 file2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ ls
file1 file2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ ls -l
total 8
-rw-rw-r--. 1 ajit_ravindrannair ajit_ravindrannair 84 Jul 27 15:55 file1
-rw-rw-r--. 1 ajit_ravindrannair ajit_ravindrannair 84 Jul 27 15:57 file2
```

- **man ls** - It opens the manual page for the ls command.

```
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ pwd
/home/ajit_ravindrannair/userdir1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ mkdir dir2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ ls
dir2 file1 file2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ rmdir dir2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ man ls
```

```

ls [OPTION]... [FILE]...

DESCRIPTION
List information about the FILES (the current directory by default). Sort
entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.

-a, --all
    do not ignore entries starting with .

-A, --almost-all
    do not list implied . and ..

--author
    with -l, print the author of each file

-b, --escape
    print octal escapes for nongraphic characters

--block-size=SIZE
```



- **head -2 file1** - Head command is used to display the first few lines of a file.  
The -2 option specifies that you want to display the first 2 lines of file1.
- **tail -2 file1** - To display the last few lines of a file.  
The -2 option specifies that you want to display the last 2 lines of file1
- **tail -2 file1 > file3** - Redirects the output of the tail command to a new file, file3.
- **rm file3** - To remove a file

## 5) TEXT PROCESSING UNIX COMMAND

- **grep unix file1** - To search for a specific pattern within a file.
- **grep -i Unix file1** - -i option makes the search case-insensitive.

```

[ajit_ravindrannair@VIMSMYS-42 userdir1]$ grep unix file1
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ grep -i Unix file1
We are learning Unix
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ grep -i unix file1
We are learning Unix
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ cat file1
We are learning Unix
This is part of BigData session
We are learning Hadoop as well
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ head -2 file1
We are learning Unix
This is part of BigData session
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ tail -2 file1
This is part of BigData session
We are learning Hadoop as well
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ tail -2 file1>file3
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ ls
file1 file2 file3
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ rm file3

```

- **exit** - To exit the current terminal session

```

[ajit_ravindrannair@VIMSMYS-42 userdir1]$ rmdir dir2
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ man ls
[ajit_ravindrannair@VIMSMYS-42 userdir1]$ exit

```

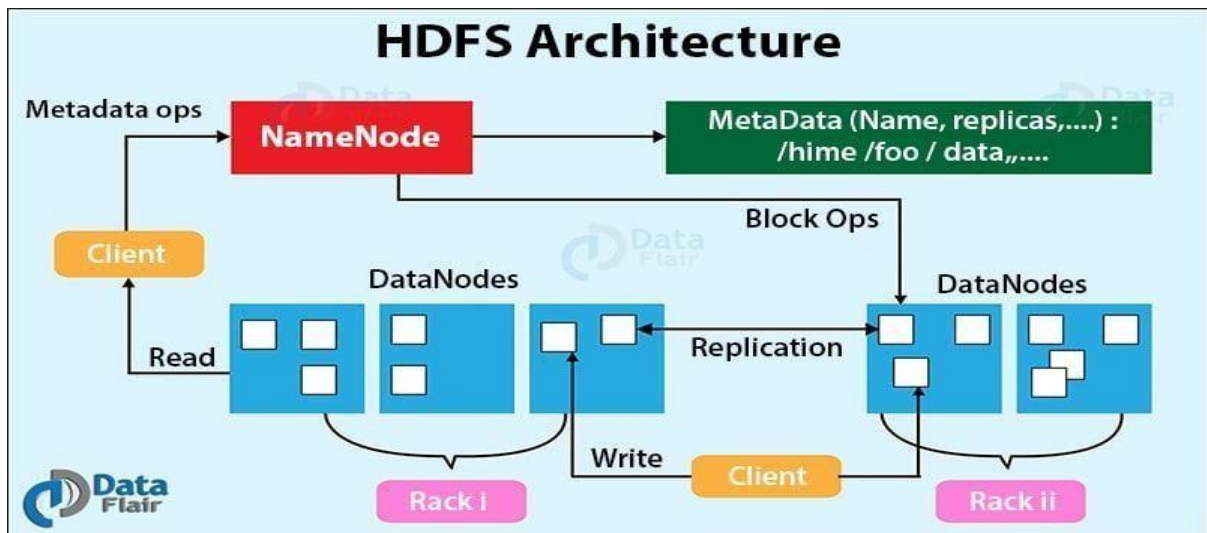


# HDFS (Hadoop Distributed File System)

## INTRODUCTION:

HDFS, or **Hadoop Distributed File System**, is the primary storage system used by Hadoop applications. It is designed to store **large files** across multiple machines in a **distributed** fashion, providing high **fault tolerance** and **scalability**. HDFS is a key component of the **Hadoop ecosystem**, which is widely used for processing and storing **big data**.

## HDFS Architecture:



HDFS follows a **master-slave** architecture with two main components:

### 1. NameNode (Master):

- The **NameNode** is the central server that manages the **metadata** and the **namespace** of the file system. It knows which files are stored, where the blocks of those files are located across the cluster, and the structure of directories and files.
- The NameNode **does not store actual data** but manages the mapping of data blocks to **DataNodes**.

### 2. DataNode (Slave):

- **DataNodes** are the worker nodes in the HDFS cluster that store the actual data. Each DataNode is responsible for managing the data blocks assigned to it.
- They provide the **read/write** operations when a client requests data or when data is written to HDFS.
- DataNodes report the status of their stored blocks (i.e., how many replicas are available) to the NameNode periodically.





### 3. **Block:**

- Data in HDFS is split into **fixed-size blocks** (by default, 128MB or 256MB in size). Each block is replicated across multiple DataNodes to ensure data reliability and fault tolerance.

### 4. **Secondary NameNode:**

- The **Secondary NameNode** is responsible for periodically merging the **edit logs** (record of changes made to the file system) with the **fsimage** (a snapshot of the file system). This helps in checkpointing and reducing the startup time of the NameNode.

## **HDFS Case Study:**

### **Storing and Processing Customer Data for E-commerce**

This case study will involve storing and processing **customer transaction data** for an e-commerce company using the **Hadoop Distributed File System (HDFS)**. We will use **Hadoop**, **MapReduce**, and **Hive** to show the process of data ingestion, storage, and basic analysis using HDFS.

### **Objective:**

The goal is to:

1. **Store large customer transaction data** (structured CSV) in **HDFS**.
2. **Process the data** using **MapReduce** to compute the total amount spent by each customer.
3. Perform a **Hive query** on the processed data to generate summary statistics.

### **Assumptions:**

- We assume you have a Hadoop cluster set up with **HDFS** and **Hive**.
- The customer transaction data is available in a CSV format (transactions.csv), where each row contains:
  - ✓ CustomerID (unique ID of the customer)
  - ✓ TransactionID (unique ID of the transaction)
  - ✓ Amount (the amount spent in the transaction)

Sample data (transactions.csv):

CustomerID,TransactionID,Amount		
	1,101,50	
	2,102,30	
	1,103,20	
	3,104,40	
	2,105,60	
	1,106,25	



## Step 1: Storing Data in HDFS

1. **Upload the data to HDFS:** We will start by uploading the transactions.csv file to HDFS.

```
hadoop fs -mkdir /user/hadoop/transactions
hadoop fs -put transactions.csv /user/hadoop/transactions/
```

2. **Check if the file is stored correctly in HDFS**

```
hadoop fs -ls /user/hadoop/transactions
```

## Step 2: MapReduce Program to Calculate Total Amount Spent by Each Customer

**MapReduce** program in Java to calculate the total amount spent by each customer.

### Mapper Code (CustomerSpendMapper.java):

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

public class CustomerSpendMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable amount = new IntWritable();
    private Text customerId = new Text();

    public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {
        // Skip the header line
        String[] columns = value.toString().split(",");
        if (columns.length == 3 && !columns[0].equals("CustomerID")) {
            customerId.set(columns[0]);
            amount.set(Integer.parseInt(columns[2]));
            context.write(customerId, amount);
        }
    }
}
```

### Reducer Code (CustomerSpendReducer.java):

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class CustomerSpendReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    private IntWritable totalAmount = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
        InterruptedException {
        int total = 0;
```



```

        for (IntWritable val : values) {
            total += val.get();
        }
        totalAmount.set(total);
        context.write(key, totalAmount);
    }
}

```

#### **Driver Code (CustomerSpendDriver.java):**

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class CustomerSpendDriver {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Customer Spend Calculation");
        job.setJarByClass(CustomerSpendDriver.class);
        job.setMapperClass(CustomerSpendMapper.class);
        job.setReducerClass(CustomerSpendReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path("/user/hadoop/transactions"));
        FileOutputFormat.setOutputPath(job, new Path("/user/hadoop/output"));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

#### **Compile and Run the Program:**

1. **Compile the code** (Make sure Hadoop and HDFS libraries are in the classpath):

```

javac -classpath `hadoop classpath` -d . CustomerSpendMapper.java
javac -classpath `hadoop classpath` -d . CustomerSpendReducer.java
javac -classpath `hadoop classpath` -d . CustomerSpendDriver.java

```

2. **Package the classes into a JAR file:**

```

jar -cvf customer_spend.jar *.class

```

3. **Run the MapReduce job:**

```

hadoop jar customer_spend.jar CustomerSpendDriver

```



#### 4. **Check the output** in the output directory

```
hadoop fs -ls /user/hadoop/output
hadoop fs -cat /user/hadoop/output/part-r-000000
```

The **output** should look like this:

```
1 95
2 90
3 40
```

This indicates that:

- Customer 1 spent a total of 95.
- Customer 2 spent a total of 90.
- Customer 3 spent a total of 40.

### Step 3: Analyzing Data with Hive

Analyze this data using **Hive** to generate summary statistics (e.g., total amount spent by each customer).

#### 1. **Create a Hive table for the customer spend data:**

```
CREATE EXTERNAL TABLE customer_spend (
  CustomerID INT,
  TotalSpend INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'

LOCATION '/user/hadoop/output';
```

#### 2. **Query the table in Hive** to get a summary:

```
SELECT * FROM customer_spend;
```

**Output:**

CustomerID	TotalSpend
1	95
2	90
3	40



## Step 4: Result Interpretation and Benefits

### 1. Result Analysis:

- We can now see the total amount spent by each customer based on the transaction data stored in HDFS. This data can be used for targeted marketing, customer segmentation, and predictive analytics.
- For instance, Customer 1 is the highest spender, and this data could be used for special promotions or offers for this customer.

### 2. Advantages of Using HDFS:

- **Scalability:** HDFS allows storing large volumes of data and processing it in parallel, making it suitable for an e-commerce company with millions of transactions.
- **Fault Tolerance:** Even if some DataNodes fail, HDFS ensures that data is not lost due to replication.
- **Cost-Effective:** By using commodity hardware and the distributed nature of Hadoop, E-Shop Inc. can scale up their storage and processing power without significant upfront costs.



## **Conclusion:**

In this case study, we demonstrated how HDFS can be used to store, process, and analyze customer transaction data in a distributed environment. By leveraging **MapReduce** for data processing and **Hive** for querying, E-Shop Inc. was able to efficiently handle large datasets, scale as their business grew, and derive valuable insights from their customer data.

This case study highlights the power of **HDFS** for big data storage and processing, enabling businesses to perform advanced analytics on large datasets that were previously unmanageable using traditional database systems.



## INTRODUCTION

Hadoop MapReduce is a robust framework designed to process massive datasets efficiently by distributing the workload across a cluster of computers. Leveraging the Hadoop Distributed File System (HDFS), it breaks down large files into smaller blocks and processes them in parallel, enabling high-speed data handling for terabytes and petabytes of information.

Each block of data is processed simultaneously by mapper tasks, and the results can be consolidated by reducer tasks. This parallel computation significantly outperforms traditional single-system processing models, making MapReduce ideal for large-scale data processing. By default, data blocks in HDFS are 128 MB in size, but this can be adjusted to suit specific requirements, ensuring scalability and flexibility in handling diverse workloads.

### Java as a Language for MapReduce Programming

- MapReduce algorithm contains two important tasks namely Map and Reduce
- Both these tasks need to be written inside the MapReduce program by the developers
- Map task takes the input data(file blocks),applies the processing logic, converts it in to a suitable format
- Map task output is given as input to Reduce task which consolidates the output of many mapper tasks to produce the final output
- MapReduce programs are usually written in Java but however it can be written in other languages like C, Python etc.

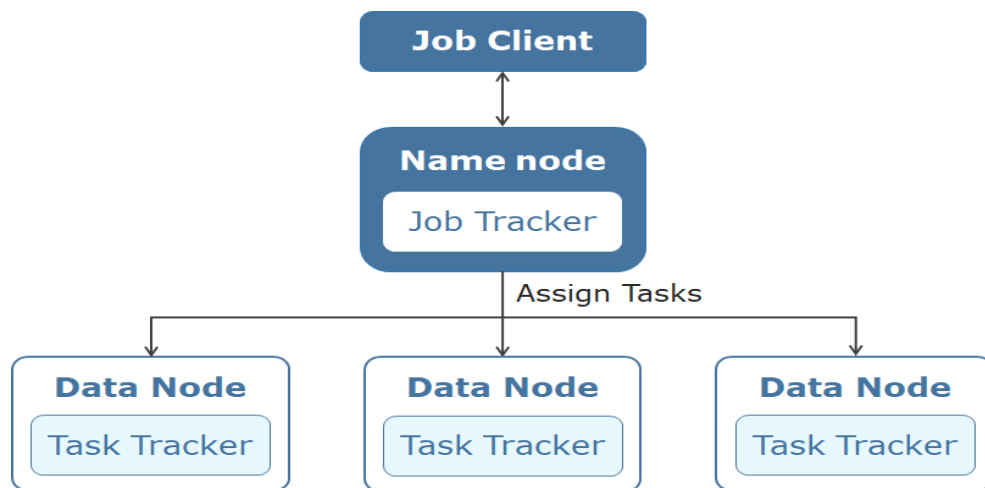
### MapReduce - Architecture

MapReduce architecture contains two core components as Daemon services (background services) responsible for performing tasks like running mapper logic and reducer logic on data blocks, monitoring the machine's failures, managing the mapper/reducer tasks failures against data blocks etc.

### Hadoop 1.x Daemons and Architecture Overview

- **NameNode:** The master service of HDFS that stores metadata (e.g., file splits, locations, and replication details).
- **DataNode:** The slave service of HDFS that stores actual data blocks. Data blocks are replicated across nodes for fault tolerance.
- **JobTracker:** The master service in MapReduce responsible for assigning and monitoring tasks by communicating with the NameNode.
- **TaskTracker:** The slave service running on DataNodes to execute tasks (Mapper and Reducer) on data blocks. It sends heartbeat signals to the JobTracker to confirm node health and task status.



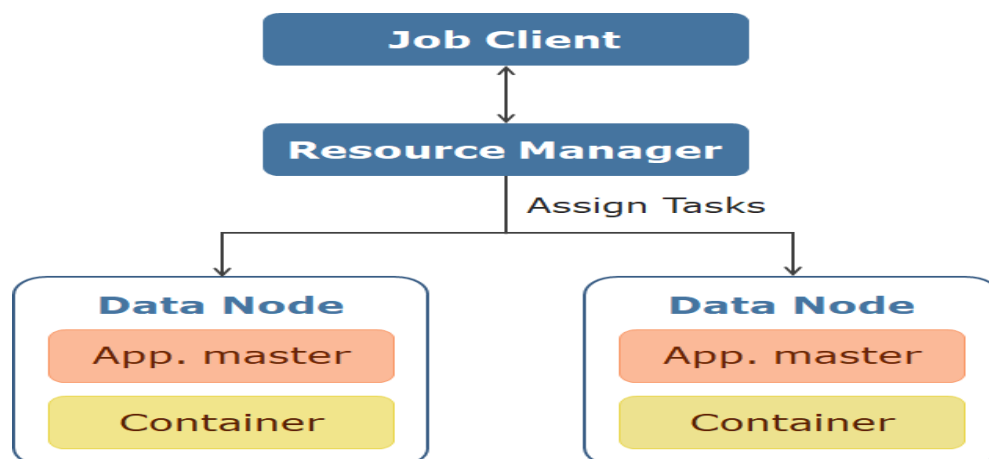


**Hadoop 1.x architecture diagram with daemons**

## Hadoop 2.x Daemons and Architecture diagram

In Hadoop 2.x, YARN (Yet Another Resource Negotiator) is used to segregate resource management and job scheduling/monitoring functionalities as two separate daemons.

- **Resource Manager:** Scheduler allocates resources (i.e. slots and cores) of the cluster among the running applications.
- **NodeManager:** Runs on each node in the cluster coordinating with the Resource Manager. Node manager manages resources on node level where each node has only one Node manager.
- **ApplicationMaster:** Instance of a framework-specific library which runs a specific YARN job. It negotiates resources from the Resource Manager and uses NodeManager to execute MapReduce tasks.



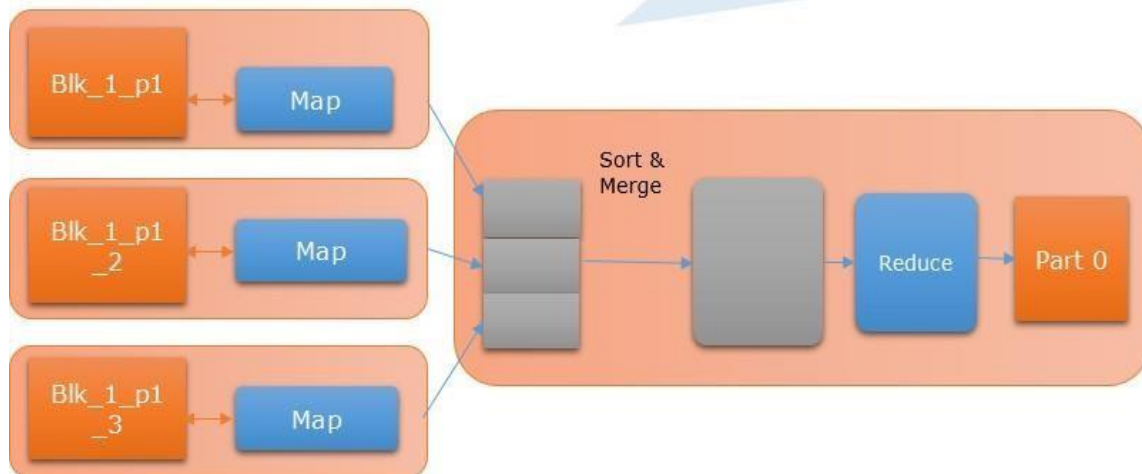
**Hadoop 2.x architecture diagram**





## MapReduce Programming model and its various phases

1. Mappers gets applied against each blocks in mapper phase.
2. Multiple mappers output(k,v) shuffled and sorted by framework
3. Shuffle and Sort's output is reduced by the reducer phase to produce the final output



### WordCount Program in MapReduce

The WordCount program is a fundamental example in Hadoop MapReduce, often used to demonstrate its functionality. It counts the occurrences of each word in a given text dataset by leveraging the distributed processing capabilities of Hadoop.

### Java Program for Word Count

```
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class WordCount_MR {

    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
```



```

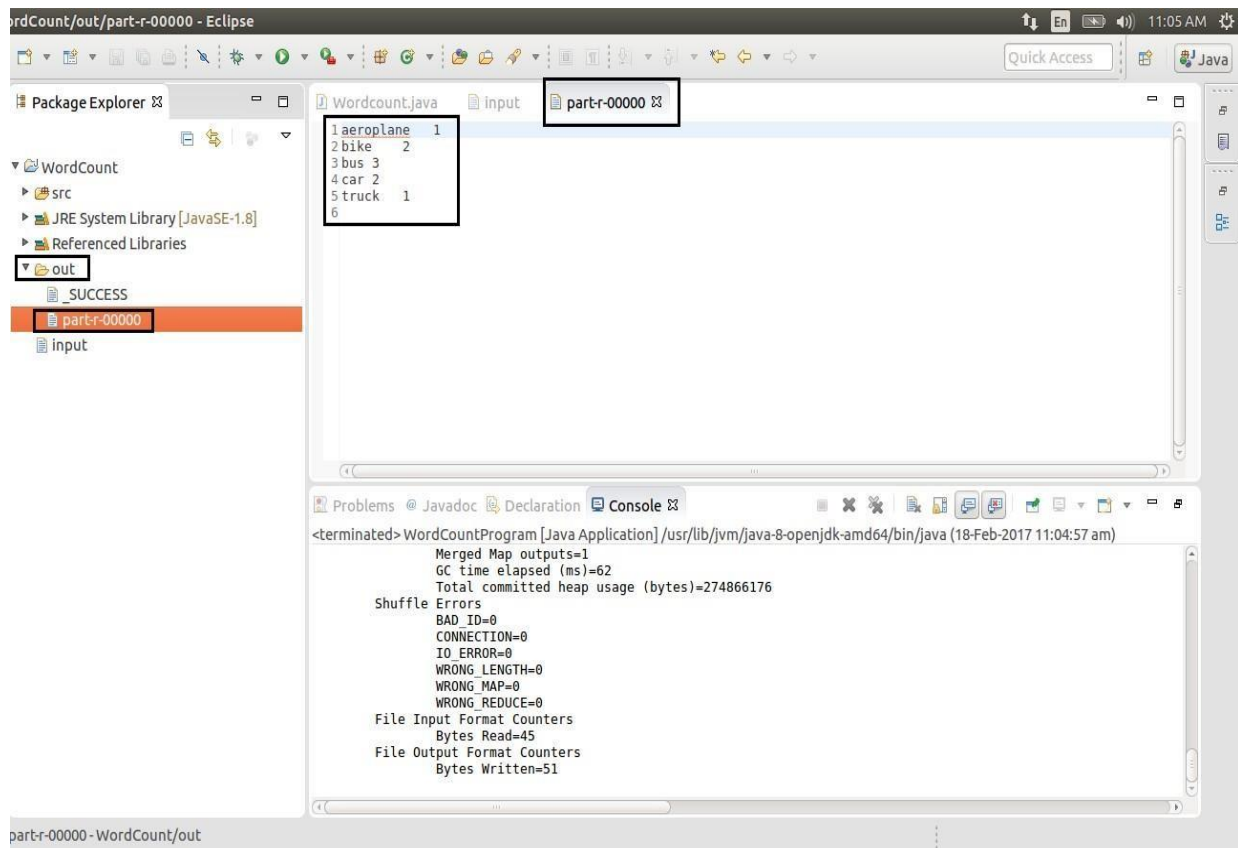
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text,
IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }

        public static void main(String[] args) throws Exception {
            JobConf conf = new JobConf(wcmr.class);
            conf.setJobName("wordcount");
            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(IntWritable.class);
            conf.setMapperClass(Map.class);
            conf.setCombinerClass(Reduce.class);
            conf.setReducerClass(Reduce.class);
            conf.setInputFormat(TextInputFormat.class);
            conf.setOutputFormat(TextOutputFormat.class);
            FileInputFormat.setInputPaths(conf, new Path(args[0]));
            FileOutputFormat.setOutputPath(conf, new Path(args[1]));
            JobClient.runJob(conf);
        }
    }
}

```

## OUTPUT:



WordCount/out/part-r-00000 - Eclipse

Package Explorer

- WordCount
  - src
  - JRE System Library [JavaSE-1.8]
  - Referenced Libraries
  - out
    - \_SUCCESS
    - part-r-00000
    - input

Wordcount.java input part-r-00000

```
1 aeroplane 1
2 bike 2
3 bus 3
4 car 2
5 truck 1
6
```

Problems Javadoc Declaration Console

```
<terminated> WordCountProgram [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (18-Feb-2017 11:04:57 am)
Merged Map outputs=1
GC time elapsed (ms)=62
Total committed heap usage (bytes)=274866176
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=45
File Output Format Counters
  Bytes Written=51
```

part-r-00000 - WordCount/out

**AIM:**

Implementing Database Operations on Hive.

**THEORY:**

Hive defines a simple SQL-like query language to querying and managing large datasets called Hive-QL (HQL). It's easy to use if you're familiar with SQL Language. Hive allows programmers who are familiar with the language to write the custom MapReduce framework to perform more sophisticated analysis.

**USES OF HIVE:**

- The Apache Hive distributed storage.
- Hive provides tools to enable easy data extract/transform/load (ETL)
- It provides the structure on a variety of data formats.
- By using Hive, we can access files stored in Hadoop Distributed File System (HDFS is used to querying and managing large datasets residing in) or in other data storage systems such as Apache HBase.

**LIMITATIONS OF HIVE:**

- i. Hive is not designed for Online transaction processing (OLTP), it is only used for the Online Analytical Processing.
- ii. Hive supports overwriting or apprehending data, but not updates and deletes.
- iii. In Hive, sub queries are not supported.

**WHY HIVE IS USED IN SPITE OF PIG?**

The following are the reasons why Hive is used in spite of Pig's availability:

- i. Hive-QL is a declarative language like SQL, PigLatin is a data flow language.
- ii. Pig: a data-flow language and environment for exploring very large datasets.
- iii. Hive: a distributed data warehouse.

**COMPONENTS OF HIVE:****1. Metastore:**

Hive stores the schema of the Hive tables in a Hive Metastore. Metastore is used to hold all the information about the tables and partitions that are in the warehouse. By default, the metastore is run in the same process as the Hive service and the default Metastore is Derby Database.

**2. SerDe:**

Serializer, Deserializer gives instructions to hive on how to process a record.



## Hive Commands:

### 1. Data Definition Language (DDL)

DDL statements are used to build and modify the tables and other objects in the database.

#### Example:

CREATE, DROP, TRUNCATE, ALTER, SHOW, DESCRIBE Statements.

Go to Hive shell by giving the command **sudo hive** and enter the command '**create database<data base name>**' to create the new database in the Hive.

```
hive> create database retail;
OK
Time taken: 5.275 seconds
hive> █
```

To list out the databases in Hive warehouse, enter the command '**show databases**.'

```
hive> show databases;
OK
default
retail
Time taken: 0.228 seconds
hive> █
```

The database creates in a default location of the Hive warehouse. In Cloudera, Hive database store in a /user/ hive/warehouse.

The command to use the database is **USE <data base name>**



The screenshot shows the HDFS directory listing for /user/hive/warehouse. The table below represents the data shown in the image.

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
nl	dir				2014-02-28 19:35	rwkr-xr-x	root	supergroup
nl	dir				2014-02-28 19:35	rwkr-xr-x	root	supergroup
hive_test.db	dir				2014-02-21 20:51	rwkr-xr-x	cloudera	supergroup
hive_test	dir				2014-02-28 19:53	rwkr-xr-x	root	supergroup
test	dir				2014-02-27 18:00	rwkr-xr-x	root	supergroup
testing	dir				2014-02-27 16:28	rwkr-xr-x	root	supergroup

Copy the input data to HDFS from local by using the copy From Local command.

```
hive> use retail;
OK
Time taken: 0.023 seconds
hive> █
```





```

txns1.txt
00000000,06-26-2011,4007024,040.33,Exercise & Fitness,Cardio Machine Accessories,Clarksville,Tennessee,credit
00000001,05-26-2011,4006742,198.44,Exercise & Fitness,Weightlifting Gloves,Long Beach,California,credit
00000002,06-01-2011,4009775,005.58,Exercise & Fitness,Weightlifting Machine Accessories,Anaheim,California,credit
00000003,06-05-2011,4002199,198.19,Gymnastics,Gymnastics Rings,Milwaukee,Wisconsin,credit
00000004,12-17-2011,4002613,098.81,Team Sports,Field Hockey,Nashville,Tennessee,credit
00000005,02-14-2011,4007591,193.63,Outdoor Recreation,Camping & Backpacking & Hiking,Chicago,Illinois,credit
00000006,10-28-2011,4002190,027.89,Puzzles,Jigsaw Puzzles,Charleston,South Carolina,credit
00000007,07-14-2011,4002964,096.01,Outdoor Play Equipment,Sandboxes,Columbus,Ohio,credit
00000008,01-17-2011,4007361,010.44,Winter Sports,Snowmobiling,Des Moines,Iowa,credit
00000009,05-17-2011,4004798,152.46,Jumping,Bungee Jumping,St. Petersburg,Florida,credit

cloudera@cloudera-vm:~$ hadoop dfs -copyFromLocal Desktop/blog/txns1.txt hdfs:/
cloudera@cloudera-vm:~$

```

When we create a table in hive, it creates in the default location of the hive warehouse.  
 – “/user/hive/warehouse”, after creation of the table we can move the data from HDFS to hive table.

The following command creates a table with in location of “/user/hive/warehouse/retail.db”  
 Note : retail.db is the database created in the Hive warehouse.

```

hive> create table txnrecords(txnno INT, txndate STRING, custno INT, amount DOUBLE,category STRING, product STRING, city STRING, state STRING, spendby STRING) row format delimited fields terminated by ',' stored as textfile;
OK
Time taken: 1.163 seconds
hive>
hive> describe txnrecords;
OK
txnno      int
txndate    string
custno     int
amount     double
category   string
product    string
city       string
state      string
spendby    string
Time taken: 0.122 seconds
hive>

```

## 2. Data Manipulation Language (DML)

DML statements are used to retrieve, store, modify, delete, insert and update data in the database.

### Example:

LOAD, INSERT Statements.


### Syntax:

**LOAD data <LOCAL> inpath <file path> into table [tablename]**

The Load operation is used to move the data into corresponding Hive table. If the keyword **local** is specified, then in the load command will give the local file system path. If the keyword local is not specified we have to use the HDFS path of the file.



```
hive> LOAD DATA INPATH '/txns1.txt' OVERWRITE INTO TABLE txnrecords;
Loading data to table retail.txnrecords
Deleted hdfs://localhost/user/hive/warehouse/retail.db/txnrecords
OK
Time taken: 0.263 seconds
hive>
```



Here are some examples for the LOAD data LOCAL command

```
hive> create table customer(custno string, firstname string, lastname string, age int,profession string) row format delimited
fields terminated by ',';
OK
Time taken: 0.102 seconds
hive>
```

```
hive> load data local inpath '/home/cloudera/Desktop/blog/custs' into table customer;
Copying data from file:/home/cloudera/Desktop/blog/custs
Copying file: file:/home/cloudera/Desktop/blog/custs
Loading data to table retail.customer
OK
Time taken: 0.227 seconds
hive>
```

After loading the data into the Hive table, we can apply the Data Manipulation Statements or aggregate functions retrieve the data.

### Example to count number of records:

Count aggregate function is used count the total number of the records in a table.

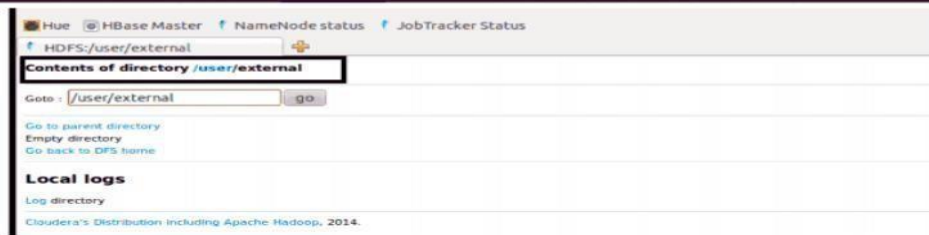
```
hive> select count(*) from txnrecords;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201402270420_0005, Tracking URL = http://localhost:50030/jobdetails.jsp?jobid=job_201402270420_0005
Kill Command = /usr/lib/hadoop/bin/hadoop job -Dmapred.job.tracker=localhost:8021 -kill job_201402270420_0005
2014-02-28 20:02:41,231 Stage-1 map = 0%, reduce = 0%
2014-02-28 20:02:48,293 Stage-1 map = 50%, reduce = 0%
2014-02-28 20:02:49,309 Stage-1 map = 100%, reduce = 0%
2014-02-28 20:02:55,350 Stage-1 map = 100%, reduce = 33%
2014-02-28 20:02:56,367 Stage-1 map = 100%, reduce = 100%
Ended Job = job_201402270420_0005
OK
50000
Time taken: 19.027 seconds
hive>
```

### ‘create external’ Table:

The **create external** keyword is used to create a table and provides a location where the table will create, so that Hive does not use a default location for this table. An **EXTERNAL** table points to any HDFS location for its storage, rather than default storage.



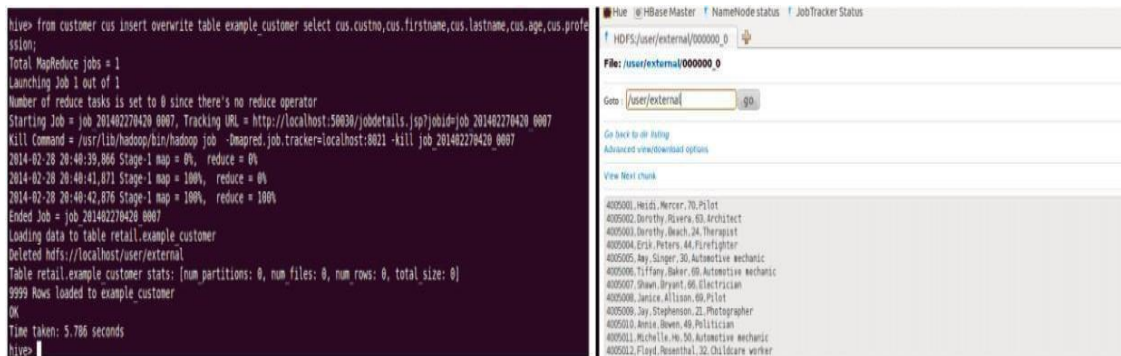
```
hive> create external table example customer(custno string, firstname string, lastname string, age int,profession string) row
format delimited fields terminated by ',' LOCATION '/user/external';
OK
Time taken: 0.059 seconds
hive>
```



### Insert Command:

The **insert** command is used to load the data Hive table. Inserts can be done to a table or a partition.

- **INSERT OVERWRITE** is used to overwrite the existing data in the table or partition.
- **INSERT INTO** is used to append the data into existing data in a table. (Note: INSERT INTO syntax is work from the version 0.8)



### Example for 'Partitioned By' and 'Clustered By' Command:

'**Partitioned by**' is used to divided the table into the Partition and can be divided in to buckets by using the '**Clustered By**' command.

```
hive> create table txnrecsByCat(txnno INT, txndate STRING, custno INT, amount DOUBLE,product STRING, city STRING, state STRING
, spendby STRING) partitioned by (category STRING) clustered by (state) INTO 10 buckets row format delimited fields terminated
by ',' stored as textfile;
OK
Time taken: 0.101 seconds
hive>
```

```
hive> from txnrecords txn INSERT OVERWRITE TABLE record PARTITION(category)select txn.txnno,txn.txndate,txn.custno,txn.amount,
txn.product,txn.city,txn.state,txn.spendby, txn.category;
FAILED: Error in semantic analysis: Dynamic partition strict mode requires at least one static partition column. To turn this
off set hive.exec.dynamic.partition.mode=nonstrict
```

When we insert the data Hive throwing errors, the dynamic partition mode is strict and dynamic partition not enabled (by [Jeff](#) at [dresshead website](#)). So we need to set the following parameters in Hive shell.