**HIVE**

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

This is a brief tutorial that provides an introduction on how to use Apache Hive HiveQL with Hadoop Distributed File System. This tutorial can be your first step towards becoming a successful Hadoop Developer with Hive.

**Hive vs SQL**

SQL is a general purpose database language that has extensively been used for both transactional and analytical queries. Hive, on the other hand, is built with an analytical focus. What this means is Hive lacks update and delete functions but is superfast in reading and processing huge volumes of data faster than SQL. Hence, even though Hive SQL is SQL-like, lack of support for modifying or deleting data is a major difference.

Despite the working differences, once you enter the Hive world from SQL, similarity in language ensures smooth transition but it is important to note the differences in constructs and syntax, else you’re in for frustrating times.

Now that we have an introduction to all the three data mining languages and their head to head comparisons, we close this article by analyzing different situations and listing the best fit option for each.

**When is it best to use Apache Pig?**

When you are looking to process clusters of unorganized, unstructured, decentralized data and don’t want to deviate too much from your solid SQL foundation, Pig is the option to go with. You no longer need to get into writing core MapReduce jobs. If you already have SQL background, the learning curve will be smooth and development time will be faster.

**When is it best to use Apache Hive?**

Big Data enterprises require fast analysis of data collected over a period of time. Hive is an excellent tool for analytical querying of historical data. It is to be noted that the data needs to be well organized, which would allow Hive to fully unleash its processing and analytical prowess. Querying real time data with Hive might not be the best idea, as it would be a time consuming job, defeating the original goal of fast processing (HBase is the answer for real time analytics, successfully utilized by Facebook).

**When is it best to use SQL?**

SQL is the oldest data analysis option among the three and its ability to update itself in line with growing user expectations make it relevant even today. While it is definitely a better option than excel (or any other open source spreadsheet application) for serious data analysts, it comes short when business demands fast paced processing and analysis. However, when the requirements are not too demanding, SQL is an excellent tool. Its familiarity and flexibility find favor with developers. Given that a large portion of developer community around the globe is familiar with SQL, its usage makes them productive from day one. It provides developers the facility to extend and optimize it, which make it highly customizable.

**HIVE ARCHITECTURE:**



Figure 1 shows the major components of Hive and its interactions with Hadoop. As shown in that figure, the main components of Hive are:

* UI – The user interface for users to submit queries and other operations to the system. As of 2011 the system had a command line interface and a web based GUI was being developed.
* Driver – The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modelled on JDBC/ODBC interfaces.
* Compiler – The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.
* Metastore – The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
* Execution Engine – The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

Figure 1 also shows how a typical query flows through the system. The UI calls the execute interface to the Driver (step 1 in Figure 1).

The Driver creates a session handle for the query and sends the query to the compiler to generate an execution plan (step 2).

The compiler gets the necessary metadata from the metastore (steps 3 and 4).

This metadata is used to typecheck the expressions in the query tree as well as to prune partitions based on query predicates. The plan generated by the compiler (step 5) is a DAG of stages with each stage being either a map/reduce job, a metadata operation or an operation on HDFS.

For map/reduce stages, the plan contains map operator trees (operator trees that are executed on the mappers) and a reduce operator tree (for operations that need reducers). The execution engine submits these stages to appropriate components (steps 6, 6.1, 6.2 and 6.3). In each task (mapper/reducer) the deserializer associated with the table or intermediate outputs is used to read the rows from HDFS files and these are passed through the associated operator tree. Once the output is generated, it is written to a temporary HDFS file though the serializer (this happens in the mapper in case the operation does not need a reduce).

The temporary files are used to provide data to subsequent map/reduce stages of the plan. For DML operations the final temporary file is moved to the table's location. This scheme is used to ensure that dirty data is not read (file rename being an atomic operation in HDFS). For queries, the contents of the temporary file are read by the execution engine directly from HDFS as part of the fetch call from the Driver (steps 7, 8 and 9).

## **Hive Data Model**

Data in Hive is organized into:

* **Tables** – These are analogous to Tables in Relational Databases. Tables can be filtered, projected, joined and unioned. Additionally all the data of a table is stored in a directory in HDFS. Hive also supports the notion of external tables wherein a table can be created on prexisting files or directories in HDFS by providing the appropriate location to the table creation DDL. The rows in a table are organized into typed columns similar to Relational Databases.
* **Partitions** – Each Table can have one or more partition keys which determine how the data is stored, for example a table T with a date partition column ds had files with data for a particular date stored in the <table location>/ds=<date> directory in HDFS. Partitions allow the system to prune data to be inspected based on query predicates, for example a query that is interested in rows from T that satisfy the predicate T.ds = '2008-09-01' would only have to look at files in <table location>/ds=2008-09-01/ directory in HDFS.
* **Buckets** – Data in each partition may in turn be divided into Buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory. Bucketing allows the system to efficiently evaluate queries that depend on a sample of data (these are queries that use the SAMPLE clause on the table).

Apart from primitive column types (integers, floating point numbers, generic strings, dates and booleans), Hive also supports arrays and maps. Additionally, users can compose their own types programmatically from any of the primitives, collections or other user-defined types. The typing system is closely tied to the SerDe (Serailization/Deserialization) and object inspector interfaces. User can create their own types by implementing their own object inspectors, and using these object inspectors they can create their own SerDes to serialize and deserialize their data into HDFS files). These two interfaces provide the necessary hooks to extend the capabilities of Hive when it comes to understanding other data formats and richer types. Builtin object inspectors like ListObjectInspector, StructObjectInspector and MapObjectInspector provide the necessary primitives to compose richer types in an extensible manner. For maps (associative arrays) and arrays useful builtin functions like size and index operators are provided. The dotted notation is used to navigate nested types, for example a.b.c = 1 looks at field c of field b of type a and compares that with 1.

## **Hive Query Language**

HiveQL is an SQL-like query language for Hive. It mostly mimics SQL syntax for creation of tables, loading data into tables and querying the tables. HiveQL also allows users to embed their custom map-reduce scripts. These scripts can be written in any language using a simple row-based streaming interface – read rows from standard input and write out rows to standard output. This flexibility comes at a cost of a performance hit caused by converting rows from and to strings. However, we have seen that users do not mind this given that they can implement their scripts in the language of their choice. Another feature unique to HiveQL is multi-table insert. In this construct, users can perform multiple queries on the same input data using a single HiveQL query. Hive optimizes these queries to share the scan of the input data, thus increasing the throughput of these queries several orders of magnitude.

**Hive Data Types**

Hive supports different data types to be used in table columns. The data types supported by Hive can be broadly classified in Primitive and Complex data types.

**A. Primitive data types:**  
The primitive data types supported by Hive are listed below:

#### **Numeric Types**

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

1. **Date/Time Types**

* TIMESTAMP
* DATE

#### **String Types**

* STRING
* VARCHAR
* CHAR

#### **Misc Types**

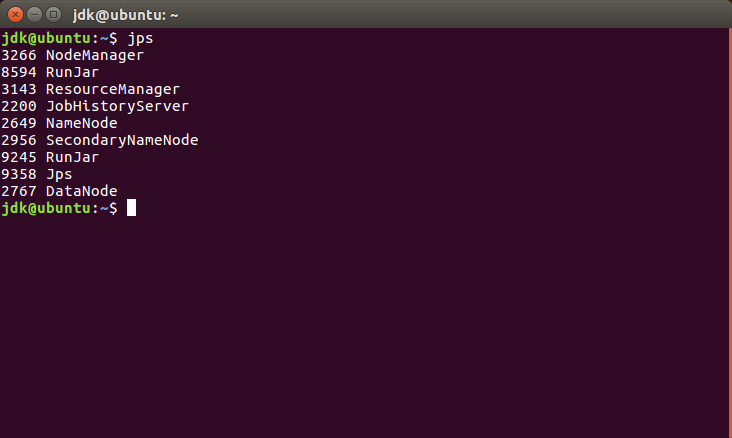
* BOOLEAN
* BINARY

Apart from these primitive data types Hive offers some complex data types which are listed below:  
  
**B. Complex data types**

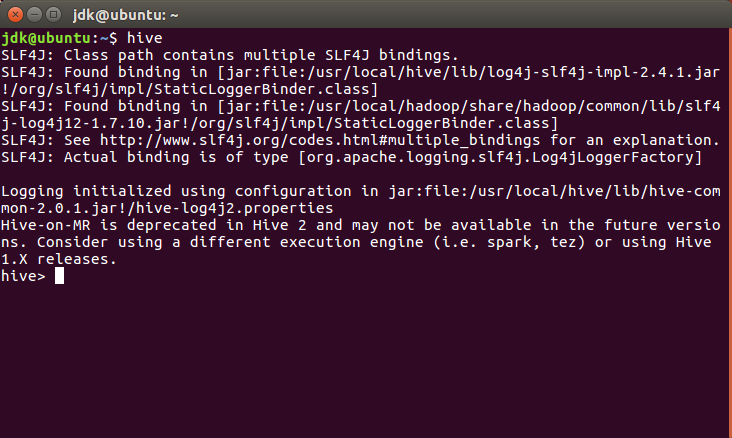
#### **Complex Types**

* arrays: ARRAY<data\_type>
* maps: MAP<primitive\_type, data\_type>
* structs: STRUCT<col\_name : data\_type [COMMENT col\_comment], ...>
* union: UNIONTYPE<data\_type, data\_type, ...>

1. To work on hive, we first check if all the services are running. We check this by the command **jps.**

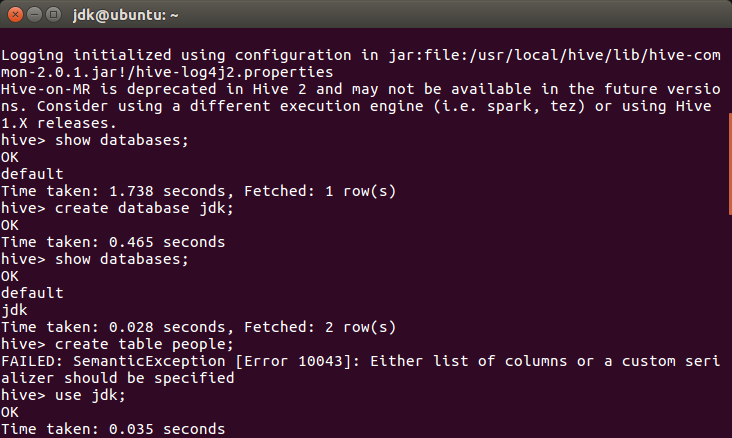


1. Now, we will start the hive shell using the command **hive**.

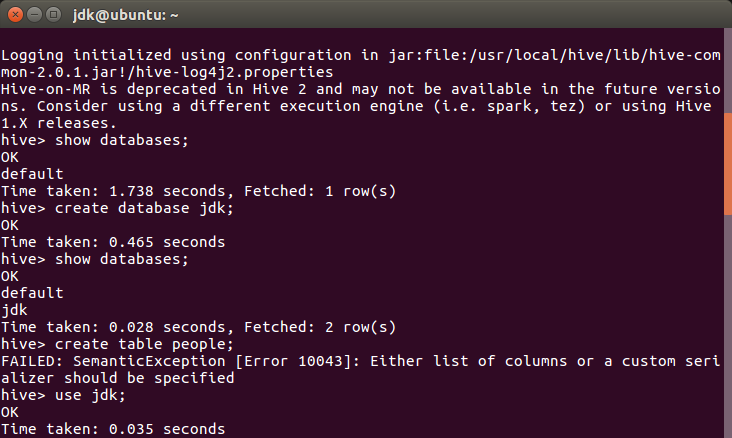


1. We can see the already present databases by the command:

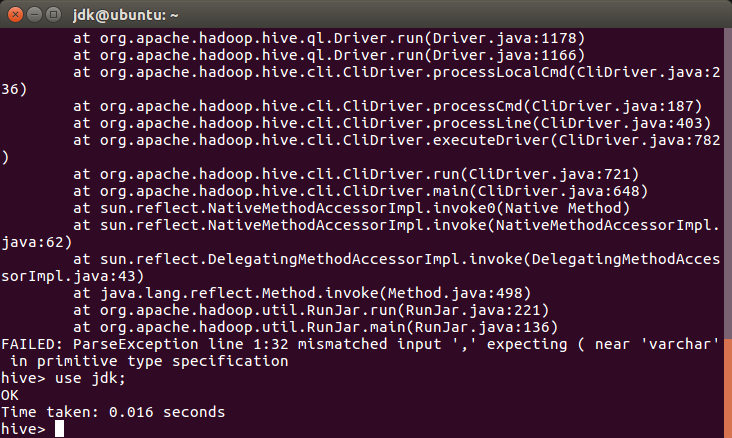
**show databases;**



1. To create a new database we use the command **create database db\_name;**



1. We then use the newly created database by the **use** command.

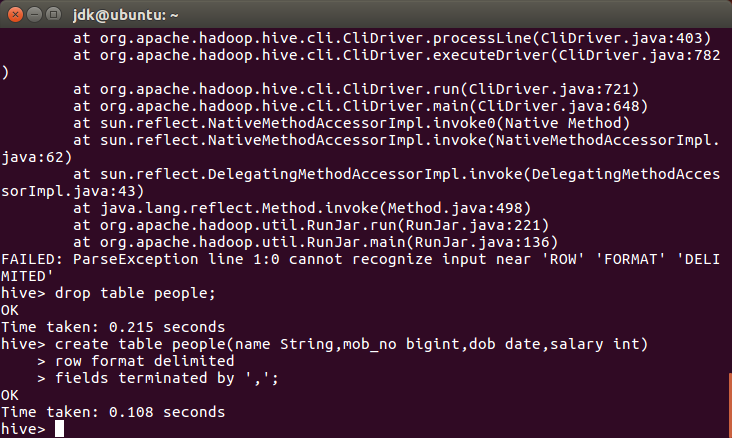


1. We then use the command of creating table along with defining the schema as:

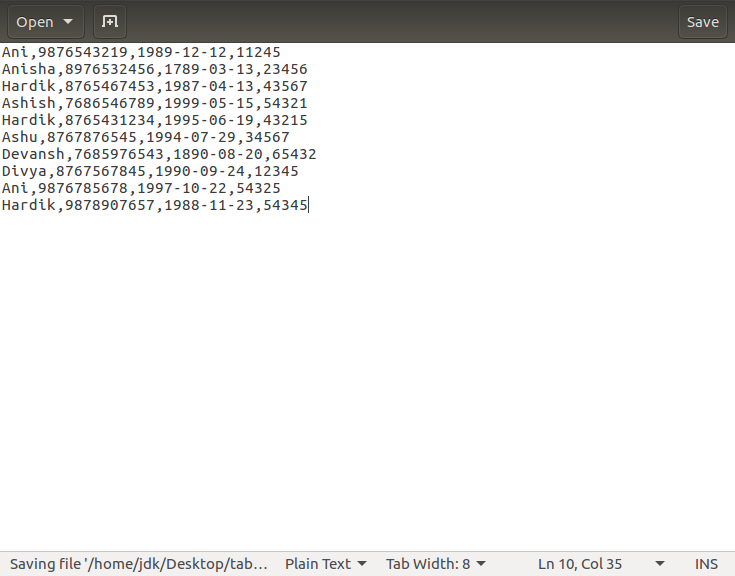
**create table table\_name(Schema)** //this defines the table name along with the schema

**row format delimited** // this shows that the data is entered in row format

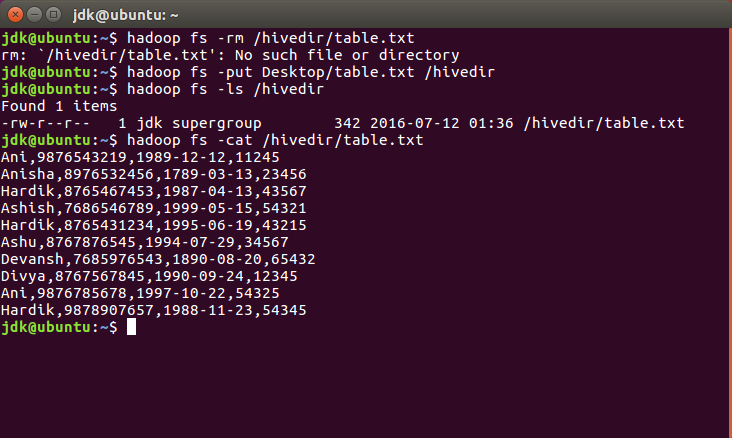
**fields terminated by ‘,’;** //this shows the each field in the row is separated by comma



1. Then we create the table as following:



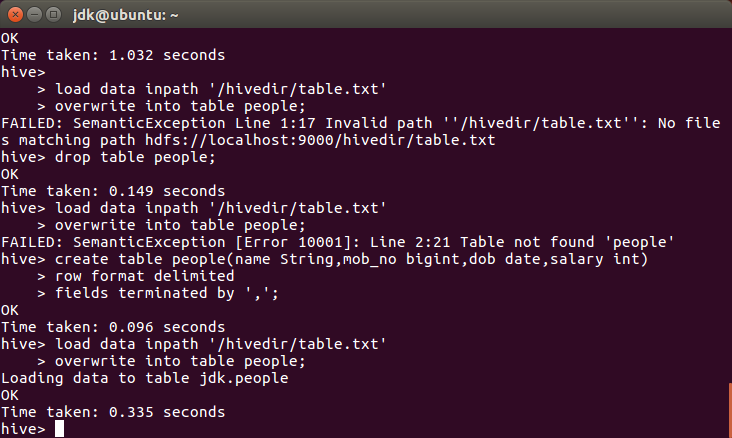
1. We then copy this text file into the hdfs hive directory using the **put** command. After that we can check the successful copying of the file using the **ls** command.
2. We also use the **cat** command to see the output of the file on the screen.



1. Now we load this table into the table created in the hive shell using the commad:

**load data inpath ‘/path’**

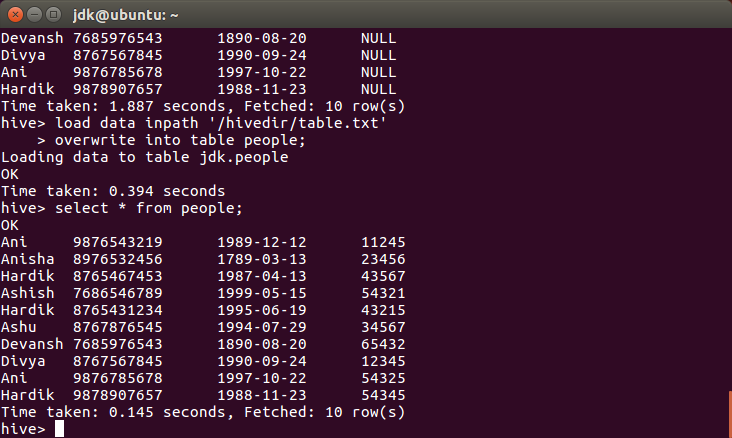
**overwrite into table table\_name;**



**EXERCISE:**

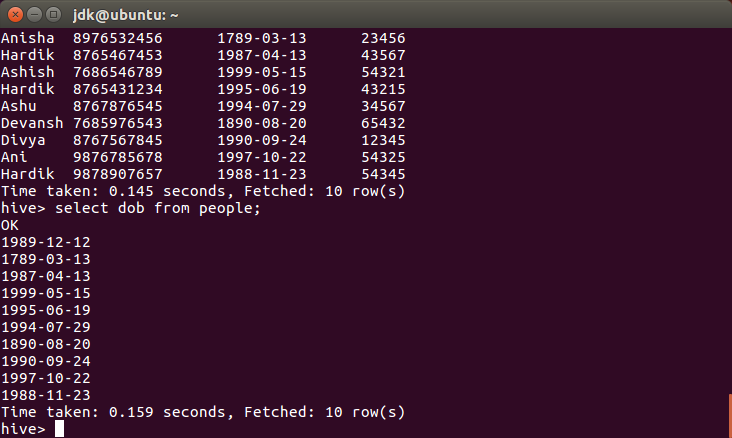
**Output the whole table:**

1. Then we can see the output of the file using the **select** command as **select \* from people;**



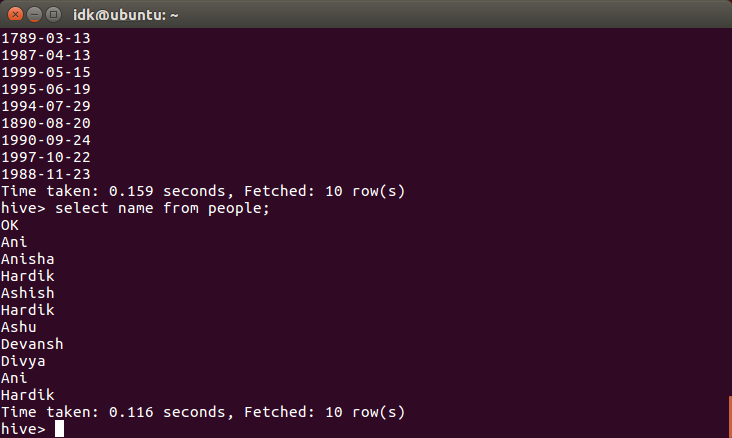
**Output only the DOB:**

1. We use the command as **select dob from people;**



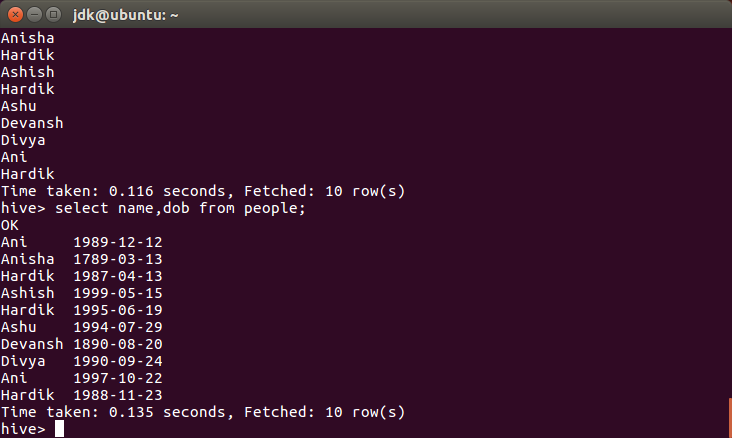
**Output only the name:**

1. We use the command as **select name from people;**



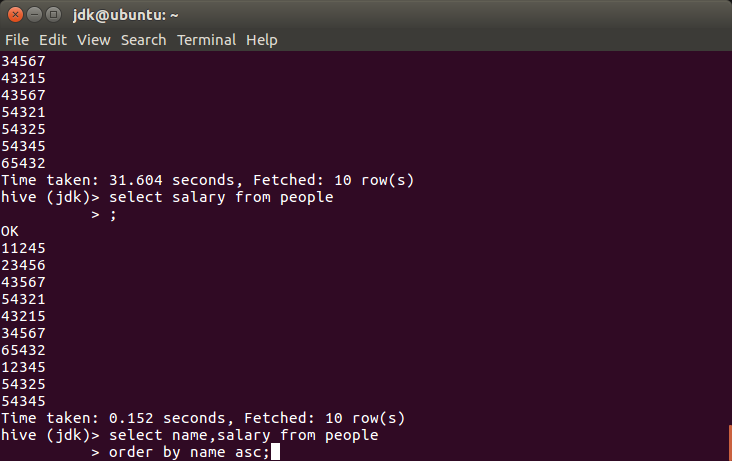
**Output name and DOB:**

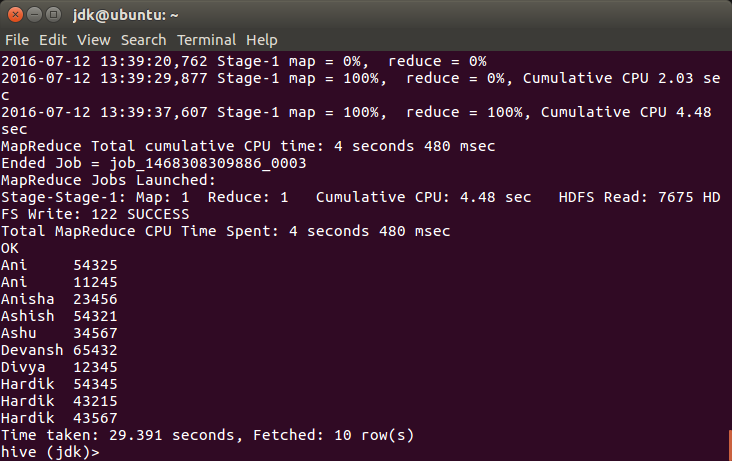
1. We use the command as **select name,dob from people;**



**Output Name, Salary with Name in ascending order:**

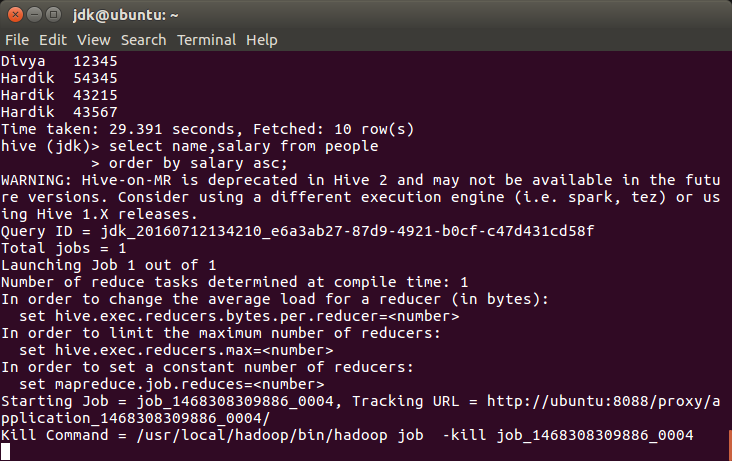
1. We use the command as **select name,salary from people order by name asc;**

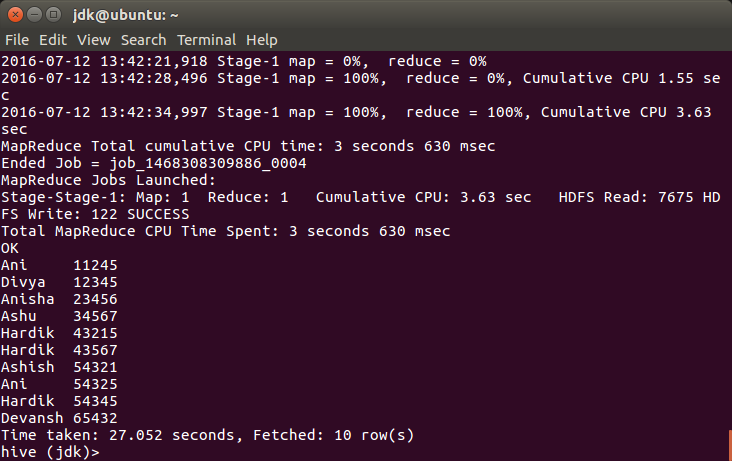




**Output Name, Salary with Salary in ascending order:**

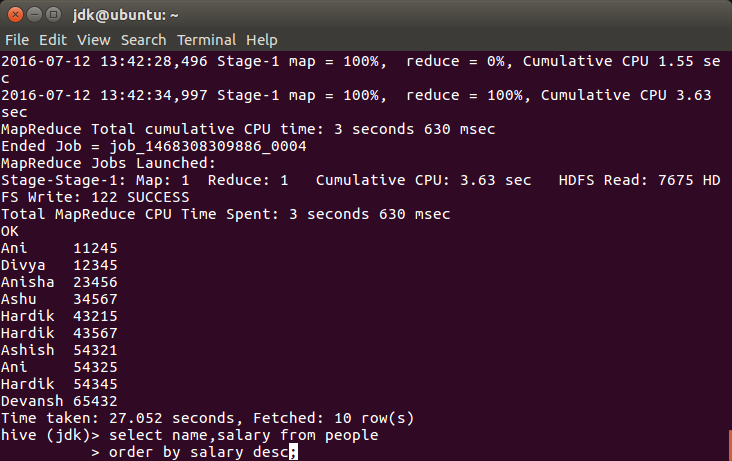
1. We use the command as **select name,salary from people order by salary asc;**

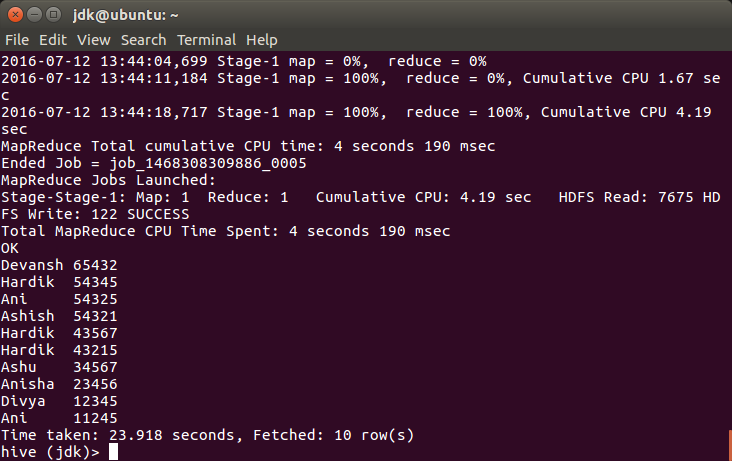




**Output Name, Salary with Salary in descending order:**

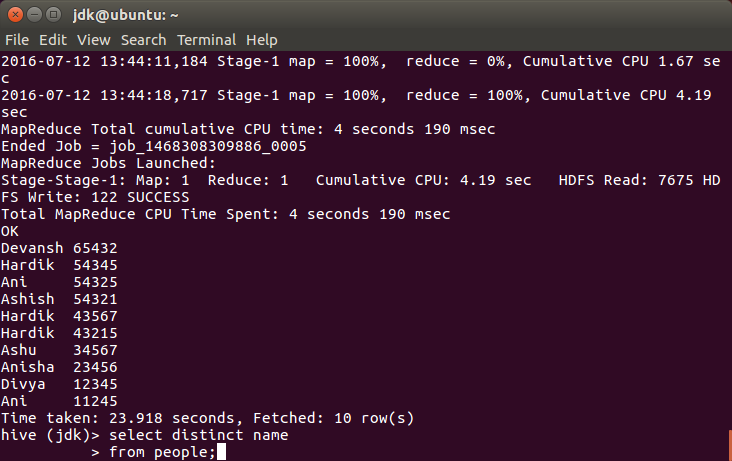
1. We use the command as **select name,salary from people order by salary desc;**

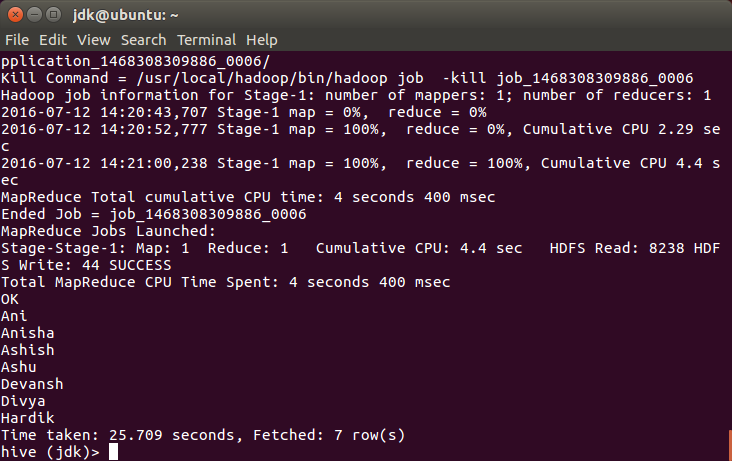




**Output Distinct Names:**

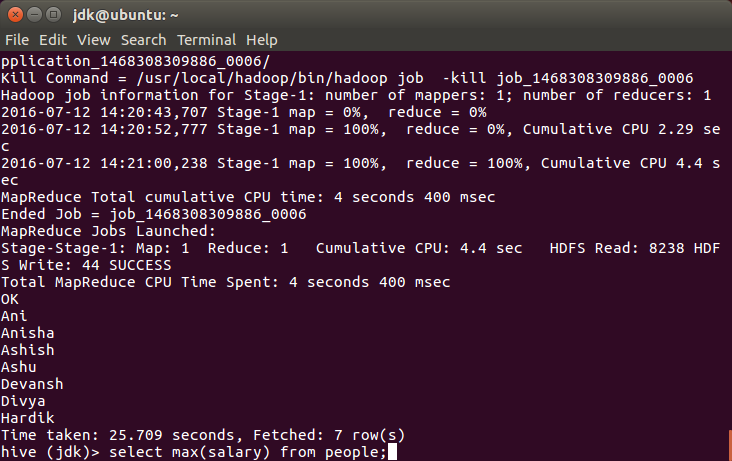
1. We use the command as **select distinct name from people;**

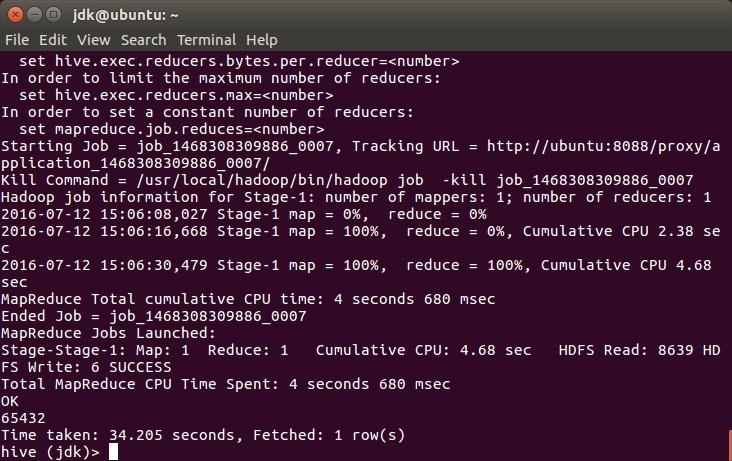




**Output Maximum salary:**

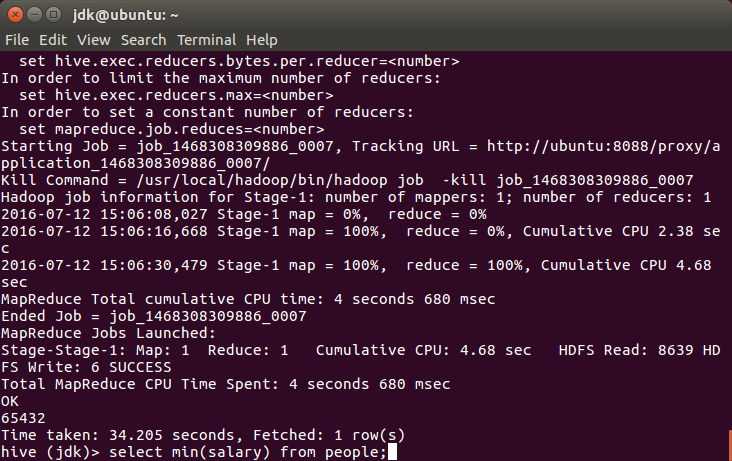
1. We use the command as **select max(salary) from people;**

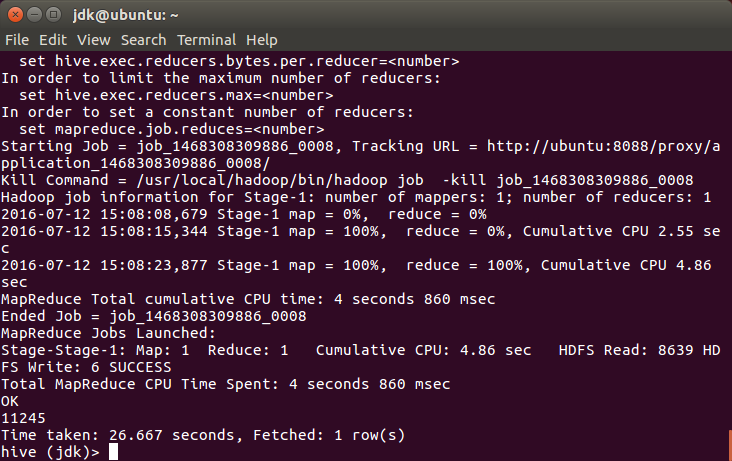




**Output Minimum salary:**

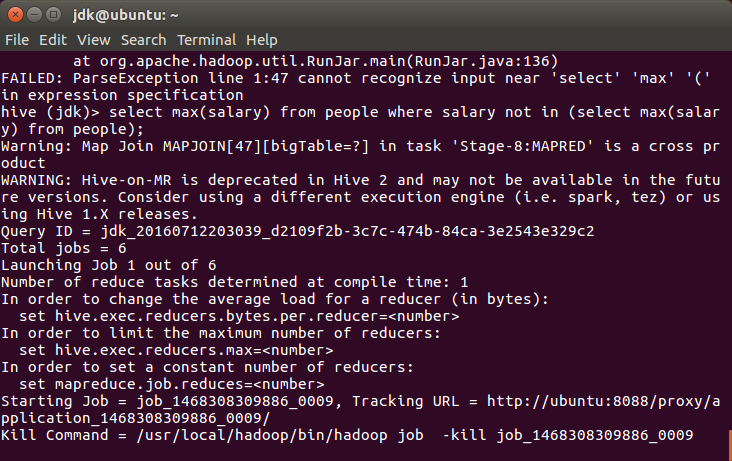
1. We use the command as **select min(salary) from people;**

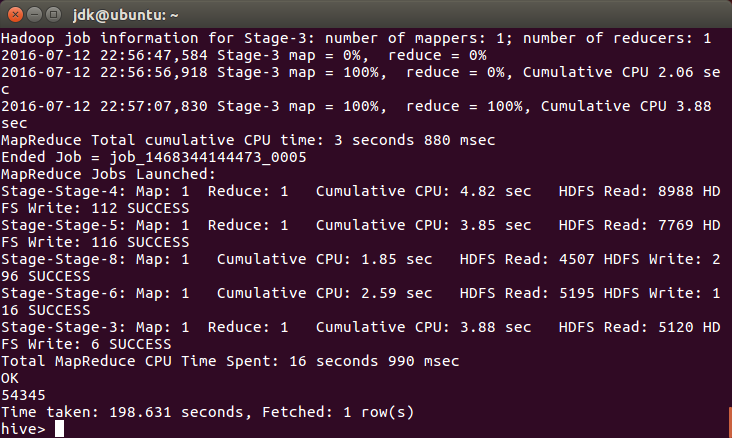




**Output 2nd highest salary:**

1. We use the command as **select max(salary) from people where salary not in (select max(salary) from people);**





**Output 4th highest salary:**

1. We use the command as **select salary from (select salary from people sort by salary desc limit 4) result sort by salary asc limit 1;**

