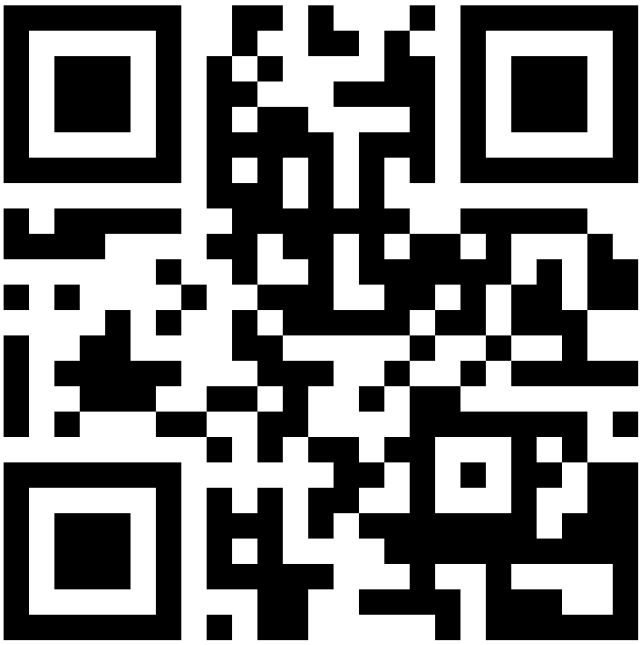
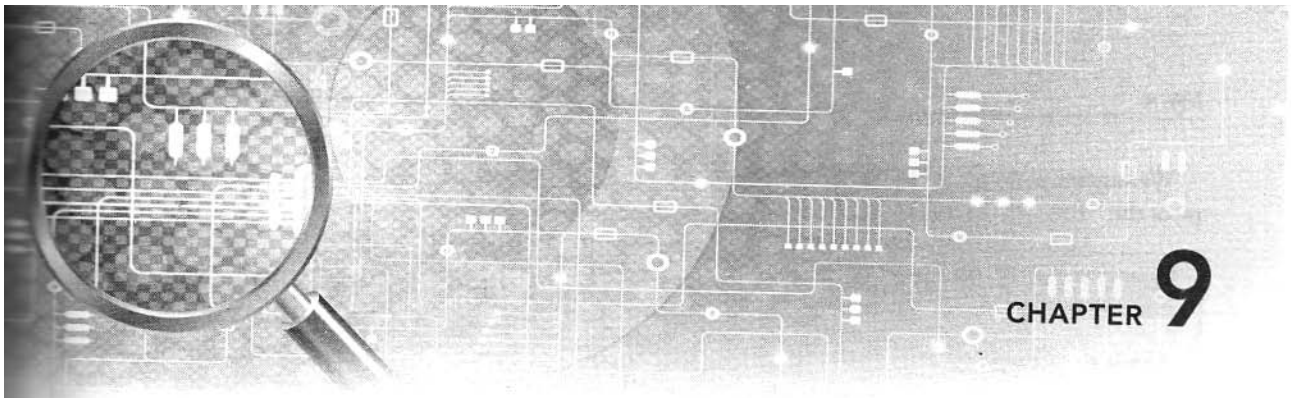
Scan for Upcoming MSRIT Connect (in beta)



bit.ly/ritconnectbeta



CHAPTER 9

Introduction to Hive

BRIEF CONTENTS

- What's in Store?
- What is Hive?
  - History of Hive and Recent Releases of Hive
  - Hive Features
  - Hive Integration and Work Flow
  - Hive Data Units
- Hive Architecture
- Hive Data Types
  - Primitive Data Types
  - Collection Data Types
- Hive File Format
  - Text File
  - Sequential File
  - RCFile (Record Columnar File)
- Hive Query Language (HQL)
  - DDL (Data Definition Language) Statements
  - DML (Data Manipulation Language) Statements
  - Starting Hive Shell
  - Database
  - Tables
  - Partitions
  - Bucketing
  - Views
  - Sub-Query
  - Joins
  - Aggregation
  - Group By and Having
- RCFILE Implementation
- SERDE
- User-Defined Function (UDF)

"Information is the oil of the 21st century, and analytics is the combustion engine."

– Peter Sondergaard, Gartner Research

WHAT'S IN STORE?

We assume that you are already familiar with commercial database systems. In this chapter, we will try to use that knowledge as our base to build a structure on Hadoop for effective analysis. We will discuss the importance of Hive with the help of use cases. We will also enrich your knowledge by working with Hive Query Language.

We suggest you refer to some of the learning resources suggested at the end of this chapter and also complete the "Test Me" exercises.

CASE STUDY: RETAIL LOG PROCESSING

About the Company

TENTOTEN is a Retail Store which has a chain of hypermarkets in India. They have 250+ stores across 95 cities and towns. About 45,000+ people are working in **TENTOTEN**. **TENTOTEN** deals in a wide range of products including fashion apparels, food products, books, furniture, etc. Around 1500+ customers visit and/or purchase products every day from each of these stores.

Problem Scenario

The approximate size of **TENTOTEN** log datasets is 12 TB. Information about the various stores is stored in the form of semi-structured data. Traditional Business Intelligence (BI) tools are good when data is present in pre-defined schema and datasets are just several hundreds of gigabytes. But the **TENTOTEN** dataset is mostly log dataset, which does not conform to any particular schema. Querying such large dataset is difficult and immensely time consuming.

The challenges are:

1. Moving the log dataset to HDFS (Hadoop Distributed File System).
2. Performing analysis on HDFS data.

Hadoop MapReduce can be used to resolve these issues. However we will still have to deal with the below constraints:

1. Writing complex MapReduce jobs in Java can be tedious and error prone.
2. Joining across large datasets is quite tricky.

Enter Hive to counter the above challenges.

9.1 WHAT IS HIVE?

Hive is a Data Warehousing tool that sits on top of Hadoop. Refer Figure 9.1. Hive is used to process structured data in Hadoop. The three main tasks performed by Apache Hive are:

1. Summarization
2. Querying
3. Analysis

Facebook initially created Hive component to manage their ever-growing volumes of log data. Later Apache software foundation developed it as open-source and it came to be known as Apache Hive.

Hive makes use of the following:

1. HDFS for Storage.
2. MapReduce for execution.
3. Stores metadata/schemas in an RDBMS.

Hive provides HQL (Hive Query Language) or HiveQL which is similar to SQL. Hive compiles SQL queries into MapReduce jobs and then runs the job in the Hadoop Cluster. It is designed to support

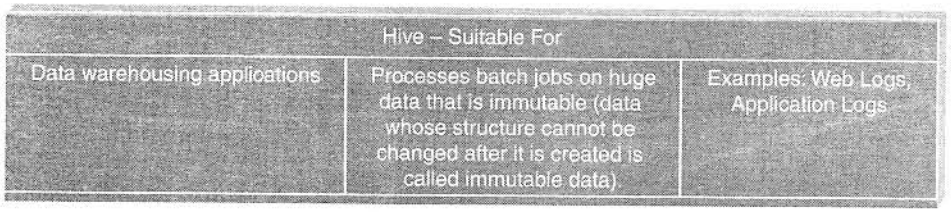


Figure 9.1 Hive – a data warehousing tool.

OLAP (Online Analytical Processing). Hive provides extensive data type functions and formats for data summarization and analysis.

Note:

1. Hive is not RDBMS.
2. It is not designed to support OLTP (Online Transaction Processing).
3. It is not designed for real-time queries.
4. It is not designed to support row-level updates.

9.1.1 History of Hive and Recent Releases of Hive

The history of Hive and recent releases of Hive are illustrated pictorially in Figures 9.2 and 9.3, respectively.

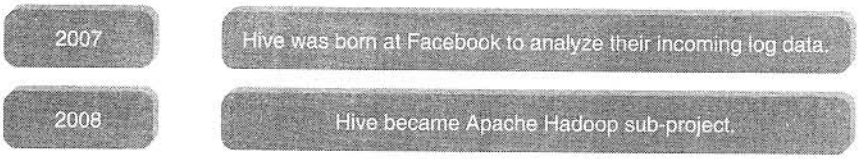


Figure 9.2 History of Hive.

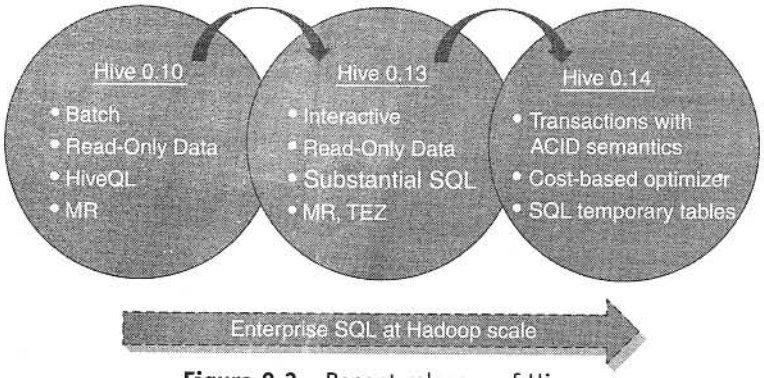


Figure 9.3 Recent releases of Hive.

9.1.2 Hive Features

1. It is similar to SQL.
2. HQL is easy to code.
3. Hive supports rich data types such as structs, lists and maps.
4. Hive supports SQL filters, group-by and order-by clauses.
5. Custom Types, Custom Functions can be defined.

9.1.3 Hive Integration and Work Flow

Figure 9.4 depicts the flow of log file analysis.

Explanation of the workflow. Hourly Log Data can be stored directly into HDFS and then data cleansing is performed on the log file. Finally, Hive table(s) can be created to query the log file.

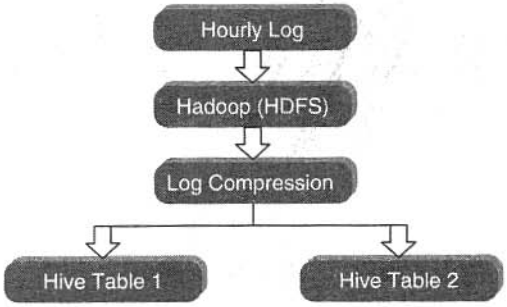


Figure 9.4 Flow of log analysis file.

9.1.4 Hive Data Units

1. **Databases:** The namespace for tables.
2. **Tables:** Set of records that have similar schema.
3. **Partitions:** Logical separations of data based on classification of given information as per specific attributes. Once hive has partitioned the data based on a specified key, it starts to assemble the records into specific folders as and when the records are inserted.
4. **Buckets (or Clusters):** Similar to partitions but uses hash function to segregate data and determines the cluster or bucket into which the record should be placed.

Let us take an example to understand partitioning and bucketing.

PICTURE THIS...

"XYZ Corp" has their customer base spread across 190+ countries. There are 5 million records/entities available. If it is required to fetch the entities pertaining to a particular country, in the absence of partitioning, there is no choice but to go through all of the 5 million entities. This despite the fact our query will eventually result in few thousand entities of the particular country. However, creating partitions based on country will greatly help to alleviate the performance issue by checking the data belonging to the partition for the country in question.

Partitioning tables changes how Hive structures the data storage. Hive will create subdirectories reflecting the partitioning structure like

.../customers/*country=ABC*

Although partitioning helps in enhancing performance and is recommended, having too many partitions may prove detrimental for few queries.

Bucketing is another technique of managing large datasets. If we partition the dataset based on customer_ID, we would end up with far too many partitions. Instead, if we bucket the customer table and use customer_id as the bucketing column, the value of this column will be hashed by a user-defined number

into buckets. Records with the same customer_id will always be placed in the same bucket. Assuming we have far more customer_ids than the number of buckets, each bucket will house many customer_ids. While creating the table you can specify the number of buckets that you would like your data to be distributed in using the syntax "CLUSTERED BY (customer_id) INTO XX BUCKETS"; here XX is the number of buckets.

When to Use Partitioning/Bucketing?

Bucketing works well when the field has high cardinality (cardinality is the number of values a column or field can have) and data is evenly distributed among buckets. Partitioning works best when the cardinality of the partitioning field is not too high. Partitioning can be done on multiple fields with an order (Year/Month/Day) whereas bucketing can be done on only one field.

Figure 9.5 shows how these data units are arranged in a Hive Cluster. Figure 9.6 describes the semblance of Hive structure with database.

A database contains several tables. Each table is constituted of rows and columns. In Hive, tables are stored as a folder and partition tables are stored as a sub-directory. Bucketed tables are stored as a file.

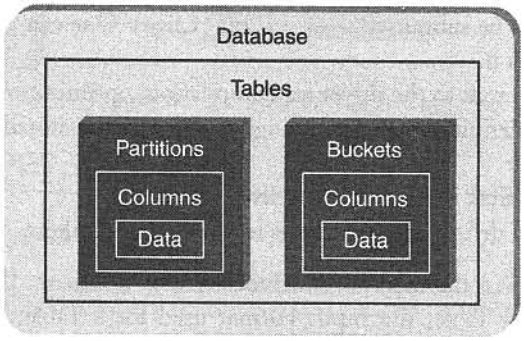


Figure 9.5 Data units as arranged in a Hive.

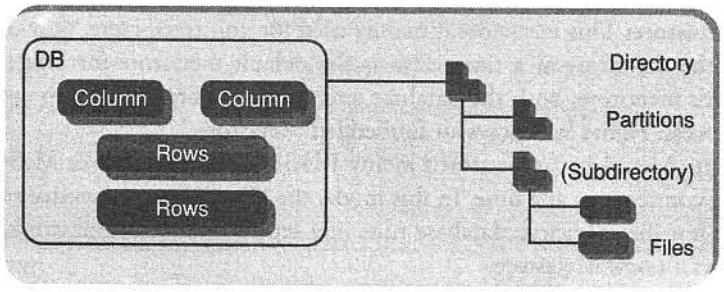


Figure 9.6 Semblance of Hive structure with database.

9.2 HIVE ARCHITECTURE

Hive Architecture is depicted in Figure 9.7. The various parts are as follows:

1. **Hive Command-Line Interface (Hive CLI):** The most commonly used interface to interact with Hive.
2. **Hive Web Interface:** It is a simple Graphic User Interface to interact with Hive and to execute query.
3. **Hive Server:** This is an optional server. This can be used to submit Hive Jobs from a remote client.

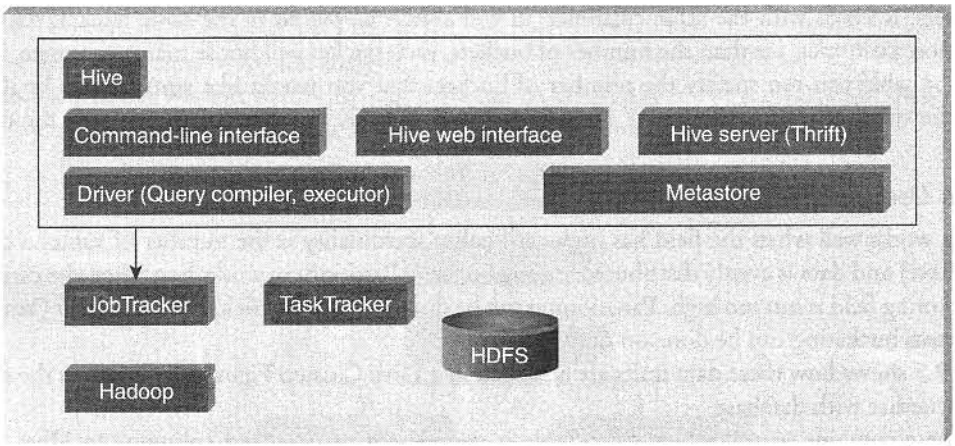


Figure 9.7 Hive architecture.

4. **JDBC/ODBC:** Jobs can be submitted from a JDBC Client. One can write a Java code to connect to Hive and submit jobs on it.
5. **Driver:** Hive queries are sent to the driver for compilation, optimization and execution.
6. **Metastore:** Hive table definitions and mappings to the data are stored in a Metastore. A Metastore consists of the following:
   - **Metastore service:** Offers interface to the Hive.
   - **Database:** Stores data definitions, mappings to the data and others.

The metadata which is stored in the metastore includes IDs of Database, IDs of Tables, IDs of Indexes, etc., the time of creation of a Table, the Input Format used for a Table, the Output Format used for a Table, etc. The metastore is updated whenever a table is created or deleted from Hive. There are three kinds of metastore.

1. **Embedded Metastore:** This metastore is mainly used for unit tests. Here, only one process is allowed to connect to the metastore at a time. This is the default metastore for Hive. It is Apache Derby Database. In this metastore, both the database and the metastore service run embedded in the main Hive Server process. Figure 9.8 shows an Embedded Metastore.
2. **Local Metastore:** Metadata can be stored in any RDBMS component like MySQL. Local metastore allows multiple connections at a time. In this mode, the Hive metastore service runs in the main Hive Server process, but the metastore database runs in a separate process, and can be on a separate host. Figure 9.9 shows a Local Metastore.
3. **Remote Metastore:** In this, the Hive driver and the metastore interface run on different JVMs (which can run on different machines as well) as in Figure 9.10. This way the database can be fire-walled from the Hive user and also database credentials are completely isolated from the users of Hive.

Figure 9.8 Embedded Metastore.

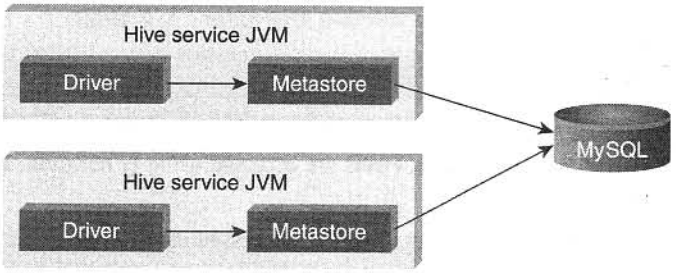


Figure 9.9 Local Metastore.

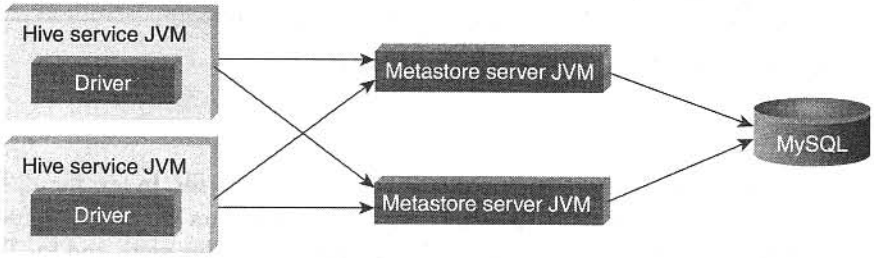


Figure 9.10 Remote Metastore.

9.3 HIVE DATA TYPES

9.3.1 Primitive Data Types

Numeric Data Type	
TINYINT	1-byte signed integer
SMALLINT	2-byte signed integer
INT	4-byte signed integer
BIGINT	8-byte signed integer
FLOAT	4-byte single-precision floating-point
DOUBLE	8-byte double-precision floating-point number
String Types	
STRING	
VARCHAR	Only available starting with Hive 0.12.0
CHAR	Only available starting with Hive 0.13.0
Strings can be expressed in either single quotes (') or double quotes (")	
Miscellaneous Types	
BOOLEAN	
BINARY	Only available starting with Hive

9.3.2 Collection Data Types

Collection Data Types	
STRUCT	Similar to 'C' struct. Fields are accessed using dot notation. E.g.: struct('John', 'Doe')
MAP	A collection of key-value pairs. Fields are accessed using [] notation. E.g.: map('first', 'John', 'last', 'Doe')
ARRAY	Ordered sequence of same types. Fields are accessed using array index. E.g.: array('John', 'Doe')

9.4 HIVE FILE FORMAT

The file formats in Hive specify how records are encoded in a file.

9.4.1 Text File

The default file format is text file. In this format, each record is a line in the file. In text file, different control characters are used as delimiters. The delimiters are ^A (octal 001, separates all fields), ^B (octal 002, separates the elements in the array or struct), ^C (octal 003, separates key–value pair), and \n. The term **field** is used when overriding the default delimiter. The supported text files are CSV and TSV. JSON or XML documents too can be specified as text file.

9.4.2 Sequential File

Sequential files are flat files that store binary key–value pairs. It includes compression support which reduces the CPU, I/O requirement.

9.4.3 RCFile (Record Columnar File)

RCFile stores the data in **Column Oriented Manner** which ensures that **Aggregation** operation is not an expensive operation. For example, consider a table which contains four columns as shown in Table 9.1.

Instead of only partitioning the table horizontally like the row-oriented DBMS (row-store), RCFile partitions this table first horizontally and then vertically to serialize the data. Based on the user-specified value, first the table is partitioned into multiple row groups horizontally. Depicted in Table 9.2, Table 9.1 is partitioned into two row groups by considering three rows as the size of each row group.

Next, in every row group RCFile partitions the data vertically like column-store. So the table will be serialized as shown in Table 9.3.

Table 9.1 A table with four columns

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44
51	52	53	54

Table 9.2 Table with two row groups

Row Group 1				Row Group 2			
C1	C2	C3	C4	C1	C2	C3	C4
11	12	13	14	41	42	43	44
21	22	23	24	51	52	53	54
31	32	33	34				

Table 9.3 Table in RCFile Format

Row Group 1	Row Group 2
11, 21, 31;	41, 51;
12, 22, 32;	42, 52;
13, 23, 33;	43, 53;
14, 24, 34;	44, 54;

9.5 HIVE QUERY LANGUAGE (HQL)

Hive query language provides basic SQL like operations. Here are few of the tasks which HQL can do easily.

1. Create and manage tables and partitions.
2. Support various Relational, Arithmetic, and Logical Operators.
3. Evaluate functions.
4. Download the contents of a table to a local directory or result of queries to HDFS directory.

9.5.1 DDL (Data Definition Language) Statements

These statements are used to build and modify the tables and other objects in the database. The DDL commands are as follows:

1. Create/Drop/Alter Database
2. Create/Drop/Truncate Table
3. Alter Table/Partition/Column
4. Create/Drop/Alter View
5. Create/Drop/Alter Index
6. Show
7. Describe

9.5.2 DML (Data Manipulation Language) Statements

These statements are used to retrieve, store, modify, delete, and update data in database. The DML commands are as follows:

1. Loading files into table.
2. Inserting data into Hive Tables from queries.

Note: Hive 0.14 supports update, delete, and transaction operations.

9.5.3 Starting Hive Shell

To start Hive, go to the installation path of Hive and type as below:

```
[root@volgalnx005 ~]# hive

Logging initialized using configuration in jar:file:/root/Desktop/VMDATA/Hive/hi
ve/lib/hive-common-0.14.0.jar!/hive-log4j.properties
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/root/Desktop/VMDATA/Hadoop/hadoop/share/hadoo
p/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/root/Desktop/VMDATA/Hive/hive/lib/hive-jdbc-0
.14.0-standalone.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive>
```

The sections have been designed as follows:

Objective:	What is it that we are trying to achieve here?
Input (optional):	What is the input that has been given to us to act upon?
Act:	The actual statement/command to accomplish the task at hand.
Outcome:	The result/output as a consequence of executing the statement.

9.5.4 Database

A database is like a container for data. It has a collection of tables which houses the data.

Objective: To create a database named "STUDENTS" with comments and database properties.

Act:

```
CREATE DATABASE IF NOT EXISTS STUDENTS COMMENT 'STUDENT Details'
WITH DBPROPERTIES ('creator' = 'JOHN');
```

Outcome:

```
hive> CREATE DATABASE IF NOT EXISTS STUDENTS COMMENT 'STUDENT Details' WITH DBPROPERTIES ('creato
r' = 'JOHN');
OK
Time taken: 0.536 seconds
hive>
```

Explanation of the syntax:

IF NOT EXIST: It is an optional clause. The create database statement with "IF Not EXISTS" clause creates a database if it does not exist. However, if the database already exists then it will notify the user that a database with the same name already exists and will not show any error message.

COMMENT: This is to provide short description about the database.

WITH DBPROPERTIES: It is an optional clause. It is used to specify any properties of database in the form of (key, value) separated pairs. In the above example, "Creator" is the "Key" and "JOHN" is the value.

We can use "**SCHEMA**" in place of "**DATABASE**" in this command.

Note: We have not specified the location where the Hive database will be created. By default all the Hive databases will be created under default warehouse directory (set by the property hive.metastore.warehouse.dir) as /user/hive/warehouse/database_name.db. But if we want to specify our own location, then the LOCATION clause can be specified. This clause is optional.

Objective: To display a list of all databases.

Act:

```
SHOW DATABASES;
```

Outcome:

```
hive> SHOW DATABASES;
OK
students
Time taken: 0.082 seconds, Fetched: 22 row(s)
hive>
```

By default, **SHOW DATABASES** lists all the databases available in the metastore. We can use "**SCHEMAS**" in place of "**DATABASES**" in this command. The command has an optional "Like" clause. It can be used to filter the database names using regular expressions such as "*" , "?", etc.

SHOW DATABASES LIKE "Stu*"

SHOW DATABASES like "Stud???s"

Objective: To describe a database.

Act:

```
DESCRIBE DATABASE STUDENTS;
```

Note: Shows only DB name, comment, and DB directory.

Outcome:

```
hive> DESCRIBE DATABASE STUDENTS;
OK
students        STUDENT Details hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/studen
ts.db   root    USER
Time taken: 0.03 seconds, Fetched: 1 row(s)
hive>
```

Objective: To describe the extended database.

Act:

```
DESCRIBE DATABASE EXTENDED STUDENTS;
```

Note: Shows DB properties also.

Outcome:

```
hive> DESCRIBE DATABASE EXTENDED STUDENTS;
OK
students        STUDENT Details hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/studen
ts.db   root    USER    {creator=JOHN}
Time taken: 0.027 seconds, Fetched: 1 row(s)
hive>
```

DESCRIBE DATABASE EXTENDED shows database's properties given under DBPROPERTIES argument at the time of creation.

We can use "**SCHEMA**" in place of "**DATABASE**", "**DESC**" in place of "**DESCRIBE**" in this command.

Objective: To alter the database properties.

Act:

```
ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');
```

Note: We can use the "ALTER DATABASE" command to

- Assign any new (key, value) pairs into DBPROPERTIES.
- Set owner user or role to the Database.

In Hive, it is not possible to unset the DB properties.

Outcome:

```
hive> ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');
OK
Time taken: 0.086 seconds
hive>
```

In the above example, the ALTER DATABASE command is used to assign new ('edited-by' = 'JAMES') pair into DBPROPERTIES. This can be verified by using the 'describe *extended*'.

Hive> DESCRIBE DATABASE Student EXTENDED

Objective: To make the database as current working database.

Act:

```
USE STUDENTS;
```

Outcome:

```
hive> USE STUDENTS;
OK
Time taken: 0.02 seconds
hive>
```

There is no command to show the current database, but use the below command statement to keep printing the current database name as suffix in the command line prompt.

set hive.cli.print.current.db=true;

Objective: To drop database.

Act:

```
DROP DATABASE STUDENTS;
```

Note: Hive creates database in the warehouse directory of Hive as shown below:

Contents of directory /user/hive/warehouse

Goto : /user/hive/warehouse go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
students.db	dir				2015-02-24 21:50	rwxr-xr-x	root	supergroup

Now assume that the database "STUDENTS" has 10 tables within it. How do we delete the complete database along with the tables contained therein?

Use the command:

```
DROP DATABASE STUDENTS CASCADE;
```

By default the mode is RESTRICT which implies that the database will NOT be dropped if it contains tables.

Note: The complete syntax is as follows:

```
DROP DATABASE [IF EXISTS] database_name [RESTRICT | CASCADE]
```

9.5.5 Tables

Hive provides two kinds of table:

1. Internal or Managed Table
2. External Table

9.5.5.1 *Managed Table*

1. Hive stores the Managed tables under the warehouse folder under Hive.
2. The complete life cycle of table and data is managed by Hive.
3. When the internal table is dropped, it drops the data as well as the metadata.

When you create a table in Hive, by default it is internal or managed table. If one needs to create an external table, one will have to use the keyword "EXTERNAL".

Objective: To create managed table named 'STUDENT'.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT) ROW
FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMAT DELIMITED F
IELDS TERMINATED BY '\t';
OK
Time taken: 0.355 seconds
hive>
```

Objective: To describe the "STUDENT" table.

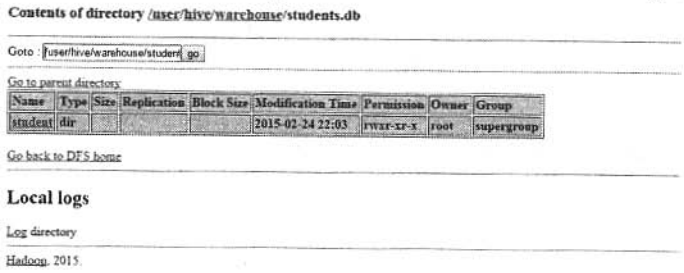
Act:

```
DESCRIBE STUDENT;
```

Outcome:

```
hive> DESCRIBE STUDENT;
OK
rollno                  int
name                    string
gpa                     float
Time taken: 0.163 seconds, Fetched: 3 row(s)
hive>
```

Note: Hive creates managed table in the warehouse directory of Hive as shown below:



Contents of directory /user/hive/warehouse/students.db

Goto : /user/hive/warehouse/student go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
student	dir				2015-02-24 22:03	rwxr-xr-x	root	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2015.

To check whether an existing table is managed or external, use the below syntax:

DESCRIBE FORMATTED tablename;

It displays complete metadata of a table. You will see one row called table type which will display either MANAGED_TABLE OR EXTERNAL_TABLE.

DESCRIBE FORMATTED STUDENT;

9.5.5.2 External or Self-Managed Table

1. When the table is dropped, it retains the data in the underlying location.
2. **External** keyword is used to create an external table.
3. **Location** needs to be specified to store the dataset in that particular location.

Objective: To create external table named 'EXT_STUDENT'.

Act:

```
CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LOCATION '/STUDENT_INFO;
```

Outcome:

```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMA
T DELIMITED FIELDS TERMINATED BY '\t'  LOCATION '/STUDENT_INFO';
OK
Time taken: 0.123 seconds
hive>
```

Note: Hive creates the external table in the specified location.

9.5.5.3 *Loading Data into Table from File*

Objective: To load data into the table from file named student.tsv.

Act:

```
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE EXT_STUDENT;
```

Note: Local keyword is used to load the data from the local file system. In this case, the file is copied. To load the data from HDFS, remove local key word from the statement. In this case, the file is moved from the original location.

Outcome:

```
hive> LOAD DATA LOCAL INPATH  '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE EXT_STUDENT;
Loading data to table students.ext_student
Table students.ext_student stats: [numFiles=0, numRows=0, totalSize=0, rawDataSize=0]
OK
Time taken: 5.034 seconds
hive>
```

Hive loads the file in the specified location as shown below:

Contents of directory /STUDENT_INFO

Goto : /STUDENT_INFO go

Go to parent directory

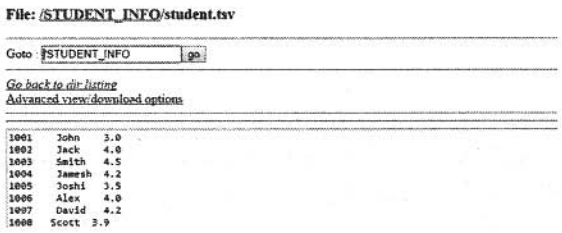
Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
student.tsv	file	121 B	3	128 MB	2015-02-24 22:19	rw-r--r--	root	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2015.


File: /STUDENT_INFO/student.tsv

Goto /STUDENT_INFO go

Go back to dir listing
Advanced view/download options

```
1001    John    3.0
1002    Jack    4.0
1003    Smith   4.5
1004    Jamesh  4.2
1005    Joshi   3.5
1006    Alex    4.0
1007    David   4.2
1008   Scott  3.9
```

Let us understand the difference between INTO TABLE and OVERWRITE TABLE with an example:

Assume the "EXT_STUDENT" table already had 100 records and the "student.tsv" file has 10 records. After issuing the LOAD DATA statement with the INTO TABLE clause, the table "EXT_STUDENT" will contain 110 records; however, the same LOAD DATA statement with the OVERWRITE clause will wipe out all the former content from the table and then load the 10 records from the data file.

9.5.5.4 Collection Data Types

Objective: To work with collection data types.

Input:
1001,John,Smith:Jones,Mark1!45:Mark2!46:Mark3!43
1002,Jack,Smith:Jones,Mark1!46:Mark2!47:Mark3!42

Act:

```
CREATE TABLE STUDENT_INFO (rollno INT,name String, sub ARRAY<STRING>,marks
MAP<STRING,INT>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY ':'
MAP KEYS TERMINATED BY '!';
LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO TABLE
STUDENT_INFO;
```

Outcome:

```
hive> CREATE TABLE STUDENT_INFO (rollno INT,name String, sub ARRAY<STRING>,marks MAP<STRING,FLOAT>)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    > COLLECTION ITEMS TERMINATED BY ':'
    > MAP KEYS TERMINATED BY '!';
OK
Time taken: 0.112 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO TABLE STUDENT_INFO;
Loading data to table students.student_info
Table students.student_info stats: [numFiles=1, totalSize=109]
OK
Time taken: 0.397 seconds
hive>
```

9.5.5.5 Querying Table

Objective: To retrieve the student details from "EXT_STUDENT" table.

Act:

```
SELECT * from EXT_STUDENT;
```

Outcome:

```
hive> select * from EXT_STUDENT;
OK
1001    John    3.0
1002    Jack    4.0
1003    Smith   4.5
1004    Scott   4.2
1005    Joshi   3.5
1006    Alex    4.5
1007    David   4.2
1008    James   4.0
1009    John    3.0
1010    Joshi   3.5
Time taken: 0.054 seconds, Fetched: 10 row(s)
hive>
```

Objective: Querying Collection Data Types.

Act:

```
SELECT * from STUDENT_INFO;
SELECT NAME,SUB FROM STUDENT_INFO;
// To retrieve value of Mark1
SELECT NAME, MARKS['Mark1'] from STUDENT_INFO;
// To retrieve subordinate (array) value
SELECT NAME,SUB[0] FROM STUDENT_INFO;
```

Outcome:

```
hive> SELECT * from STUDENT_INFO;
OK
1001    John    ["Smith","Jones"]       {"Mark1":45,"Mark2":46,"Mark3":43}
1002    Jack    ["Smith","Jones"]       {"Mark1":46,"Mark2":47,"Mark3":42}
Time taken: 0.044 seconds, Fetched: 2 row(s)
```

```
hive> SELECT NAME,SUB FROM STUDENT_INFO;
OK
John    ["Smith","Jones"]
Jack    ["Smith","Jones"]
Time taken: 0.061 seconds, Fetched: 2 row(s)
hive>
```

```
hive> SELECT NAME, MARKS['Mark1']  from STUDENT_INFO;
OK
John    45
Jack    46
Time taken: 0.06 seconds, Fetched: 2 row(s)
hive>
```

```
hive> SELECT NAME,SUB[0] FROM STUDENT_INFO;
OK
John    Smith
Jack    Smith
Time taken: 0.071 seconds, Fetched: 2 row(s)
hive>
```

9.5.6 Partitions

In Hive, the query reads the entire dataset even though a where clause filter is specified on a particular column. This becomes a bottleneck in most of the MapReduce jobs as it involves huge degree of I/O. So it is necessary to reduce I/O required by the MapReduce job to improve the performance of the query. A very common method to reduce I/O is data partitioning.

Partitions split the larger dataset into more meaningful chunks.

We will try to understand partitioning with the help of a simple example.

PICTURE THIS...

"XYZ enterprise" has a wide customer base spread across several states in the US. The data has been fed to the central system. The senior leadership team would like to get a report providing the statewise Sales %.

The IT team has proposed two options to help service this request:

1. Run the query with the where clause of each state name (such as where StateName = "A"). This will mean that the entire dataset is scanned/read through for each and every state. If we have data for 25 states, the dataset is read 25 times (once for each state). Assuming we have 5 million records, it would mean that 5 million records are read 25 times. This can be a major performance bottle neck and will worsen as the dataset grows.
2. The second option stated here can help alleviate this problem. The IT team can start off with creating 25 folders (one for each state) and instruct to have the data pertaining to states place into the folder of the respective states. This arrangement will greatly help at the time of querying the data. To get the Sales % of a particular state, the folder belonging to that state only has to be scanned/read through.

This intelligent way of grouping data during data load is termed as PARTITIONING in Hive.

This brings us to the next question.

If you are looking through the folder for state = "A", is there any possibility of you coming across data for state = "B" or state = "C"? Think through! I am sure your answer will be No! And we know the reason.

A point to note here is that as we create the partitions, there is no need to include the partitioned column along with the other columns of the dataset as this is something that is automatically taken care of "BY PARTITIONED" clause.

In our example above, we will refrain from adding the partitioned column along with other columns of the dataset and trust Hive to automatically manage this.

Partition is of two types:

1. **STATIC PARTITION:** It is upon the user to mention the partition (the segregation unit) where the data from the file is to be loaded.
2. **DYNAMIC PARTITION:** The user is required to simply state the column, basis which the partitioning will take place. Hive will then create partitions basis the unique values in the column on which partition is to be carried out.

Points to consider as you create partitions:

1. STATIC PARTITIONING implies that the user controls everything from defining the PARTITION column to loading data into the various partitioned folders.
2. As in our example above, if STATIC partition is done over the STATE column and assume by mistake the data for state "B" is placed inside the partition for state "A", our query for data for state "B" is

bound to return zilch records. The reason is obvious. A Select fired on STATIC partition just takes into consideration the partition name, and does not consider the data held inside the partition.

3. DYNAMIC PARTITIONING means Hive will intelligently get the distinct values for partitioned column and segregate data into respective partitions. There is no manual intervention.

   By default, dynamic partitioning is enabled in Hive. Also by default it is strictly implying that one is required to do one level of STATIC partitioning before Hive can perform DYNAMIC partitioning inside this STATIC segregation unit.

   In order to go with full dynamic partitioning, we have to set below property to non-strict in Hive.

   hive> set hive.exec.dynamic.partition.mode=nonstrict

9.5.6.1 Static Partition

Static partitions comprise columns whose values are known at compile time.

Objective: To create static partition based on "gpa" column.

Act:

```
CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT (rollno INT, name STRING)
PARTITIONED BY (gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED
BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) RO
W FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.105 seconds
hive>
```

Objective: Load data into partition table from table.

Act:

```
INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0)
SELECT rollno, name from EXT_STUDENT where gpa=4.0;
```

Outcome:

```
hive> INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0) SELECT rollno,name from EXT_STUDENT
where gpa=4.0;
Query ID = root_20150224230404_4500d58a-cb21-4912-ba40-788e5cf8f9da
Total jobs = 3
```

Hive creates the folder for the value specified in the partition.

Contents of directory /user/hive/warehouse/students.db

Goto: /user/hive/warehouse/student | go |

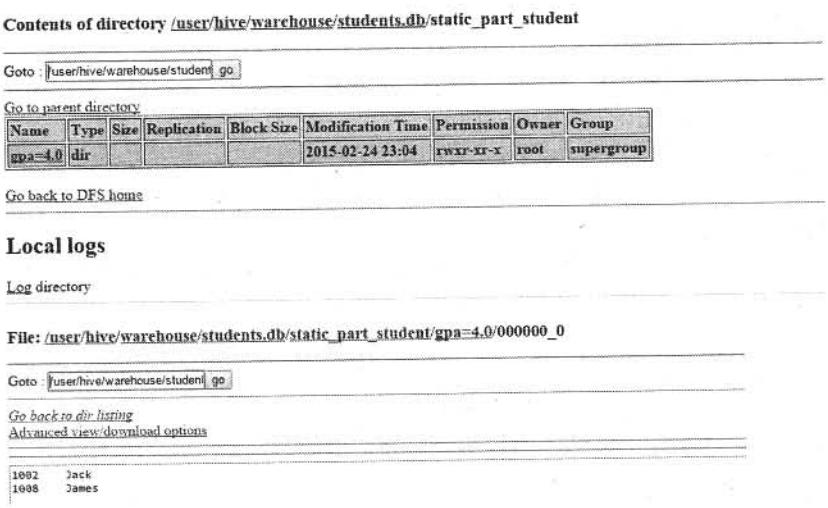
Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
static_part_student	dir				2015-02-24 23:04	rwxr-xr-x	root	supergroup
student	dir				2015-02-24 22:03	rwxr-xr-x	root	supergroup
student_info	dir				2015-02-24 22:54	rwxr-xr-x	root	supergroup

Go back to DFS home

Local logs

Log directory



Contents of directory /user/hive/warehouse/students.db/static_part_student

Goto : /user/hive/warehouse/student go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=4.0	dir				2015-02-24 23:04	rwxr-xr-x	root	supergroup

Go back to DFS home

Local logs

Log directory

File: /user/hive/warehouse/students.db/static_part_student/gpa=4.0/000000_0

Goto : /user/hive/warehouse/student go

Go back to dir listing
Advanced view/download options

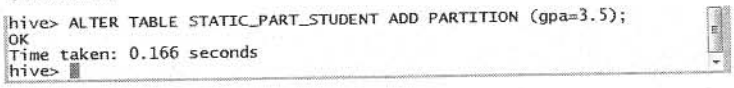
```
1002    Jack
1008    James
```

Objective: To add one more static partition based on "gpa" column using the "alter" statement.

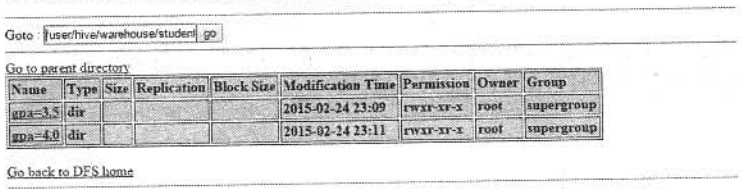
Act:

```
ALTER TABLE STATIC_PART_STUDENT ADD PARTITION (gpa=3.5);
INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0) SELECT
rollno,name from EXT_STUDENT where gpa=4.0;
```

Outcome:



```
hive> ALTER TABLE STATIC_PART_STUDENT ADD PARTITION (gpa=3.5);
OK
Time taken: 0.166 seconds
hive>
```



Contents of directory /user/hive/warehouse/students.db/static_part_student

Goto : /user/hive/warehouse/student go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.5	dir				2015-02-24 23:09	rwxr-xr-x	root	supergroup
gpa=4.0	dir				2015-02-24 23:11	rwxr-xr-x	root	supergroup

Go back to DFS home

9.5.6.2 Dynamic Partition

Dynamic partition have columns whose values are known only at Execution Time.

Objective: To create dynamic partition on column date.

Act:

```
CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno INT,name STRING)
PARTITIONED BY (gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED
BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) R
OW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.166 seconds
hive>
```

Objective: To load data into a dynamic partition table from table.

Act:

SET hive.exec.dynamic.partition = true;

SET hive.exec.dynamic.partition.mode = nonstrict;

Note: The dynamic partition strict mode requires at least one static partition column. To turn this off, set hive.exec.dynamic.partition.mode=nonstrict

INSERT OVERWRITE TABLE DYNAMIC_PART_STUDENT PARTITION (gpa) SELECT rollno,name,gpa from EXT_STUDENT;

Outcome:

Contents of directory /user/hive/warehouse/students.db/dynamic_part_student

Goto : /user/hive/warehouse/student go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.0	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=3.5	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.0	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.2	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.5	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup

Go back to DFS home

Note: Create partition for all values.

9.5.7 Bucketing

Bucketing is similar to partition. However, there is a subtle difference between partition and bucketing. In a partition, you need to create partition for each unique value of the column. This may lead to situations where you may end up with thousands of partitions. This can be avoided by using Bucketing in which you can limit the number of buckets that will be created. A bucket is a file whereas a partition is a directory.

Objective: To learn the concept of bucket in hive.

Act:

CREATE TABLE IF NOT EXISTS STUDENT (rollno INT,name STRING,grade FLOAT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' INTO TABLE STUDENT;

Set below property to enable bucketing.

set hive.enforce.bucketing=true;

```
// To create a bucketed table having 3 buckets
CREATE TABLE IF NOT EXISTS STUDENT_BUCKET (rollno INT,name STRING,grade
FLOAT)
CLUSTERED BY (grade) into 3 buckets;
// Load data to bucketed table
FROM STUDENT
INSERT OVERWRITE TABLE STUDENT_BUCKET
SELECT rollno,name,grade;
// To display content of first bucket
SELECT DISTINCT GRADE FROM STUDENT_BUCKET
TABLESAMPLE(BUCKET 1 OUT OF 3 ON GRADE);
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STUDENT (rollno INT,name STRING,grade FLOAT)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.101 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' INTO TABLE STUDENT;
Loading data to table book.student
Table book.student stats: [numFiles=1, totalSize=145]
OK
Time taken: 0.536 seconds
hive>
```

```
hive> set hive.enforce.bucketing=true;
hive>
```

```
hive> CREATE TABLE IF NOT EXISTS STUDENT_BUCKET (rollno INT,name STRING,grade FLOAT)
    > CLUSTERED BY (grade) into 3 buckets;
OK
Time taken: 0.101 seconds
hive>
```

```
hive> FROM STUDENT
    > INSERT OVERWRITE TABLE STUDENT_BUCKET
    > SELECT rollno,name,grade;
```

3 buckets have been created as shown below:

Contents of directory /user/hive/warehouse/book.db/student_bucket

Goto : /user/hive/warehouse/book.db go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
000000_0	file	59 B	3	128 MB	2015-03-10 22:29	rw-r--r--	root	supergroup
000001_0	file	59 B	3	128 MB	2015-03-10 22:29	rw-r--r--	root	supergroup
000002_0	file	28 B	3	128 MB	2015-03-10 22:29	rw-r--r--	root	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2015.

```
hive>
    > SELECT DISTINCT GRADE FROM STUDENT_BUCKET
    > TABLESAMPLE(BUCKET 1 OUT OF 3 ON GRADE);
OK
4.0
4.2
Time taken: 21.117 seconds, Fetched: 2 row(s)
hive>
```

9.5.8 Views

In Hive, view support is available only in version starting from 0.6. Views are purely logical object.

Objective: To create a view table named "STUDENT_VIEW".

Act:

```
CREATE VIEW STUDENT_VIEW AS SELECT rollno, name FROM EXT_STUDENT;
```

Outcome:

```
hive> CREATE VIEW STUDENT_VIEW AS SELECT rollno,name FROM EXT_STUDENT;
OK
Time taken: 0.606 seconds
hive>
```

Objective: Querying the view "STUDENT_VIEW".

Act:

```
SELECT * FROM STUDENT_VIEW LIMIT 4;
```

Outcome:

```
hive> SELECT * FROM STUDENT_VIEW LIMIT 4;
OK
1001    John
1002    Jack
1003    Smith
1004    Scott
Time taken: 0.279 seconds, Fetched: 4 row(s)
hive>
```

Objective: To drop the view "STUDENT_VIEW".

Act:

```
DROP VIEW STUDENT_VIEW;
```

Outcome:

```
hive> DROP VIEW STUDENT_VIEW;
OK
Time taken: 0.452 seconds
hive>
```

9.5.9 Sub-Query

In Hive, sub-queries are supported only in the FROM clause (Hive 0.12). You need to specify name for sub-query because every table in a FROM clause has a name. The columns in the sub-query select list should have unique names. The columns in the subquery select list are available to the outer query just like columns of a table.

Objective: Write a sub-query to count occurrence of similar words in the file.

Act:

```
CREATE TABLE docs (line STRING);
LOAD DATA LOCAL INPATH '/root/hivedemos/lines.txt' OVERWRITE INTO TABLE docs;
CREATE TABLE word_count AS
SELECT word, count(1) AS count FROM
(SELECT explode (split (line, ' ')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
SELECT * FROM word_count;
```

Outcome:

```
hive> CREATE TABLE docs (line STRING);
OK
Time taken: 0.118 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/lines.txt' OVERWRITE INTO TABLE docs;
Loading data to table students.docs
Table students.docs stats: [numFiles=1, numRows=0, totalSize=91, rawDataSize=0]
OK
Time taken: 2.697 seconds
hive>
```

```
hive> CREATE TABLE word_count AS
    > SELECT word, count(1) AS count FROM
    > (SELECT explode (split (line, ' ')) AS word FROM docs) w
    > GROUP BY word
    > ORDER BY word;
```

```
hive> SELECT * FROM word_count;
OK
Hadoop  2
Hive    2
Introducing     1
Introduction    1
Pig     1
Session 3
Welcome 1
to      2
Time taken: 0.062 seconds, Fetched: 8 row(s)
hive>
```

Note: The explode() function takes an array as input and outputs the elements of the array as separate rows.

In Hive 0.13, sub-queries are supported in the where clause as well.

9.5.10 Joins

Joins in Hive is similar to the SQL Join.

Objective: To create JOIN between Student and Department tables where we use RollNo from both the tables as the join key.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE STUDENT;
CREATE TABLE IF NOT EXISTS DEPARTMENT(rollno INT,deptno int,name STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/department.tsv' OVERWRITE INTO TABLE DEPARTMENT;
SELECT a.rollno, a.name, a.gpa, b.deptno FROM STUDENT a JOIN DEPARTMENT b ON a.rollno = b.rollno;
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMAT D
ELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.115 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE STUDENT
;
Loading data to table students.student
Table students.student stats: [numFiles=1, numRows=0, totalSize=145, rawDataSize=0]
OK
Time taken: 0.723 seconds
hive>
```

```
hive> CREATE TABLE IF NOT EXISTS DEPARTMENT(rollno INT,deptno int,name STRING) ROW FORM
AT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.099 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/department.tsv' OVERWRITE INTO TABLE DEPA
RTMENT;
Loading data to table students.department
Table students.department stats: [numFiles=1, numRows=0, totalSize=120, rawDataSize=0]
OK
Time taken: 0.442 seconds
hive>
```

```
hive> SELECT a.rollno, a.name, a.gpa, b.deptno  FROM STUDENT  a JOIN DEPARTMENT b ON a.
rollno = b.rollno;
OK
1001    John    3.0     101
1002    Jack    4.0     102
1003    Smith   4.5     103
1004    Scott   4.2     104
1005    Joshi   3.5     105
1006    Alex    4.5     101
1007    David   4.2     104
1008    James   4.0     102
Time taken: 115.282 seconds, Fetched: 8 row(s)
hive>
```

9.5.11 Aggregation

Hive supports aggregation functions like avg, count, etc.

Objective: To write the average and count aggregation functions.

Act:

```
SELECT avg(gpa) FROM STUDENT;
SELECT count(*) FROM STUDENT;
```

Outcome:

```
hive> SELECT avg(gpa) FROM STUDENT;

OK
3.839999961853027
Time taken: 28.639 seconds, Fetched: 1 row(s)
hive>

hive> SELECT count(*) FROM STUDENT;

OK
10
Time taken: 26.218 seconds, Fetched: 1 row(s)
hive>
```

9.5.12 Group By and Having

Data in a column or columns can be grouped on the basis of values contained therein by using "Group By". "Having" clause is used to filter out groups NOT meeting the specified condition.

Objective: To write group by and having function.

Act:

```
SELECT rollno, name,gpa FROM STUDENT GROUP BY rollno,name,gpa HAVING gpa >
4.0;
```

Outcome:

```
1003    Smith   4.5
1004    Scott   4.2
1006    Alex    4.5
1007    David   4.2
Time taken: 78.972 seconds, Fetched: 4 row(s)
hive>
```

9.6 RCFILE IMPLEMENTATION

RCFile (Record Columnar File) is a data placement structure that determines how to store relational tables on computer clusters.

Objective: To work with RCFILE Format.

Act:

```
CREATE TABLE STUDENT_RC( rollno int, name string,gpa float ) STORED AS RCFILE;
INSERT OVERWRITE table STUDENT_RC SELECT * FROM STUDENT;
SELECT SUM(gpa) FROM STUDENT_RC;
```

Outcome:

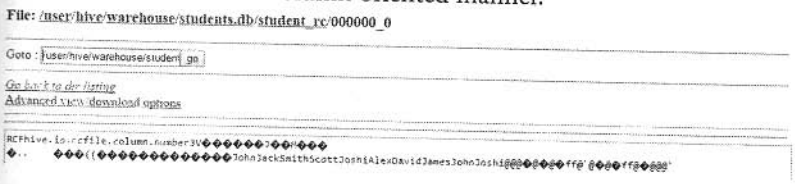
```
hive> CREATE TABLE STUDENT_RC( rollno int, name string,gpa float ) STORED AS RCFILE;
OK
Time taken: 0.093 seconds
hive>

hive> INSERT OVERWRITE table STUDENT_RC SELECT * from STUDENT;

hive> SELECT SUM(gpa) from STUDENT_RC;

OK
38.39999961853027
Time taken: 25.41 seconds, Fetched: 1 row(s)
hive>
```

Note: Stores the data in column oriented manner.



9.7 SERDE

SerDe stands for Serializer/Deserializer.

1. Contains the logic to convert unstructured data into records.
2. Implemented using Java.
3. Serializers are used at the time of writing.
4. Deserializers are used at query time (SELECT Statement).

Deserializer interface takes a binary representation or string of a record, converts it into a java object that Hive can then manipulate. Serializer takes a java object that Hive has been working with and translates it into something that Hive can write to HDFS.

Objective: To manipulate the XML data.

Input:

<employee> <empid>1001</empid> <name>John</name> <designation>Team Lead</designation> </employee>
<employee> <empid>1002</empid> <name>Smith</name> <designation>Analyst</designation> </employee>

Act:

```
CREATE TABLE XMLSAMPLE(xmldata string);
LOAD DATA LOCAL INPATH '/root/hivedemos/input.xml' INTO TABLE XMLSAMPLE;

CREATE TABLE xpath_table AS
SELECT xpath_int(xmldata,'employee/empid'),
xpath_string(xmldata,'employee/name'),
xpath_string(xmldata,'employee/designation')
FROM xmlsample;

SELECT * FROM xpath_table;
```

Outcome:

```
hive> CREATE TABLE XMLSAMPLE(xmldata string);
OK
Time taken: 0.244 seconds
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/input.xml' INTO TABLE XMLSAMPLE;
Loading data to table students.xmlsample
Table students.xmlsample stats: [numFiles=1, totalSize=194]
OK
Time taken: 0.889 seconds
hive>
```

```
hive> CREATE TABLE xpath_table AS
    > SELECT xpath_int(xmldata,'employee/empid'),
    > xpath_string(xmldata,'employee/name'),
    > xpath_string(xmldata,'employee/designation')
    > FROM xmlsample;
```

```
hive> SELECT * FROM xpath_table;
OK
1001    John    Team Lead
1002    Smith   Analyst
Time taken: 0.064 seconds, Fetched: 2 row(s)
hive>
```

9.8 USER-DEFINED FUNCTION (UDF)

In Hive, you can use custom functions by defining the User-Defined Function (UDF).

Objective: Write a Hive function to convert the values of a field to uppercase.

Act:

```
package com.example.hive.udf;
import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDF;
@Description(
 name="SimpleUDFExample")
```

```
public final class MyLowerCase extends UDF {
  public String evaluate(final String word) {
   return word.toLowerCase();
  }
}
```

Note: Convert this Java Program into Jar.

```
ADD JAR /root/hivedemos/UpperCase.jar;
CREATE TEMPORARY FUNCTION touppercase AS 'com.example.hive.udf.MyUpperCase';
SELECT TOUPPERCASE(name) FROM STUDENT;
```

Outcome:

```
hive> ADD JAR /root/hivedemos/UpperCase.jar;
Added [/root/hivedemos/UpperCase.jar] to class path
Added resources: [/root/hivedemos/UpperCase.jar]
hive> CREATE TEMPORARY FUNCTION touppercase AS 'com.example.hive.udf.MyUpperCase';
OK
Time taken: 0.014 seconds
hive>
```

```
hive> Select touppercase (name) from STUDENT;
OK
JOHN
JACK
SMITH
SCOTT
JOSHI
ALEX
DAVID
JAMES
JOHN
JOSHI
Time taken: 0.061 seconds, Fetched: 10 row(s)
hive>
```

REMIND ME

- Hive is a Data Warehousing tool.
- Hive is used to query structured data built on top of Hadoop.
- Hive provides HQL (Hive Query Language) which is similar to SQL.
- A Hive database contains several tables. Each table is constituted of rows and columns. In Hive, tables are stored as a folder and partition tables are stored as a sub-directory.
- Bucketed tables are stored as a file.

POINT ME (BOOKS)

- Programming Hive, Jason Rutherglen, O'Reilly Publication.

CONNECT ME (INTERNET RESOURCES)

- **http://en.wikipedia.org/wiki/RCFile**
- **https://cwiki.apache.org/confluence/display/Hive/DynamicPartitions**
- **https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL**
- **https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML**

TEST ME

A. Match Me

Column A	Column B
HQL	Web Logs
Database	struct, map
Complex Data Types	Set of records
Hive Application	Hive Query Language
Table	Namespace

Answers:

Column A	Column B
HQL	Hive Query Language
Database	Namespace
Complex Data Types	struct, map
Hive Application	Web Logs
Table	Set of records

B. Fill Me

1. The metastore consists of _____ and a _____.
2. The most commonly used interface to interact with Hive is _____.
3. The default metastore for Hive is _____.
4. Metastore contains _____ of Hive tables.
5. _____ is responsible for compilation, optimization, and execution of Hive queries.

Answers:

1. Metaservices, database
2. Command Line Interface
3. Derby
4. System Catalog
5. Driver

ASSIGNMENTS FOR HANDS-ON PRACTICE

ASSIGNMENT 1: PARTITION

Objective: To learn about partitions in hive.

Problem Description:

Create a partition table for customer schema to reward the customers based on their life time values.

Input:

Customer ID	Customers	Life Time Value
1001	Jack	25000
1002	Smith	8000
1003	David	12000
1004	John	15000
1005	Scott	12000
1006	Joshi	28000
1007	Ajay	12000
1008	Vinay	30000
1009	Joseph	21000

- Create a partition table if life time value is 12000.
- Create a partition table for all life time values.

ASSIGNMENT 2: PARTITION

1. Create Table:

```
CREATE EXTERNAL TABLE Emp_Proj (
   EmpID INT,
   TechnologyID INT,
   StartData date,
    EndDate date
) PARTITIONED BY (ProjectID INT)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '#'
  STORED AS TEXTFILE;
```

2. Load partitioned data:

```
LOAD DATA INPATH '/hive/data/Employee' INTO TABLE Emp_Proj PARTITION (ProjectID=1)
```

3. Verify data load:

```
SELECT * from Emp_Proj where ProjectID = 1;
```

Did you notice, it prints all data, that is, data with ProjectID = 2,3,4,5 as well.
Why did this happen? Was partitioning NOT performed?

4. Verify file location:
   hadoop fs -ls /user/hive/warehouse/Employee.db/Emp_Proj/
   It should have folder named ProjectID=1.

DYNAMIC PARTITIONING:

1. Create Table:

```
CREATE EXTERNAL TABLE Emp_Proj_DP (
    EmpID INT,
   TechnologyID INT,
   StartData date,
    EndDate date
) PARTITIONED BY (ProjectID INT)
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '#'
 STORED AS TEXTFILE;
```

2. Create TEMP table:

```
CREATE EXTERNAL TABLE Temp_ProjectID (
    EmpID INT,
   TechnologyID INT,
   StartData date,
    EndDate date,
    ProjectID INT
)
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '#'
 STORED AS TEXTFILE;
```

3. Load data in temp table from file:

LOAD DATA LOCAL INPATH '/home/Seema/bigdata/hivedata/Employee' INTO TABLE Temp_ProjectID;

4. Load data in dynamic partitioned table:

INSERT INTO TABLE Emp_Proj_DP PARTITION (ProjectID) SELECT EmpID, TechnologyID , StartDate , EndDate FROM Temp_ProjectID;

Note: The column order while select should be maintained except partitioned column, which should be selected last and if there are multiple partitioned columns then they should be cited in the order of creation.

5. Verify data load:

```
SELECT * from Emp_Proj_DP where ProjectID = 1;
SELECT * from Emp_Proj_DP where ProjectID = 2;
SELECT * from Emp_Proj_DP where ProjectID = 3;
SELECT * from Emp_Proj_DP where ProjectID = 4;
```

6. Verify file location:

```
hadoop fs -ls /user/hive/warehouse/Employee.db/Emp_Proj_DP/
```

It should have folder named ProjectID=1, ProjectID=2, ProjectID=3, etc..

ASSIGNMENT 3: HIVEQL

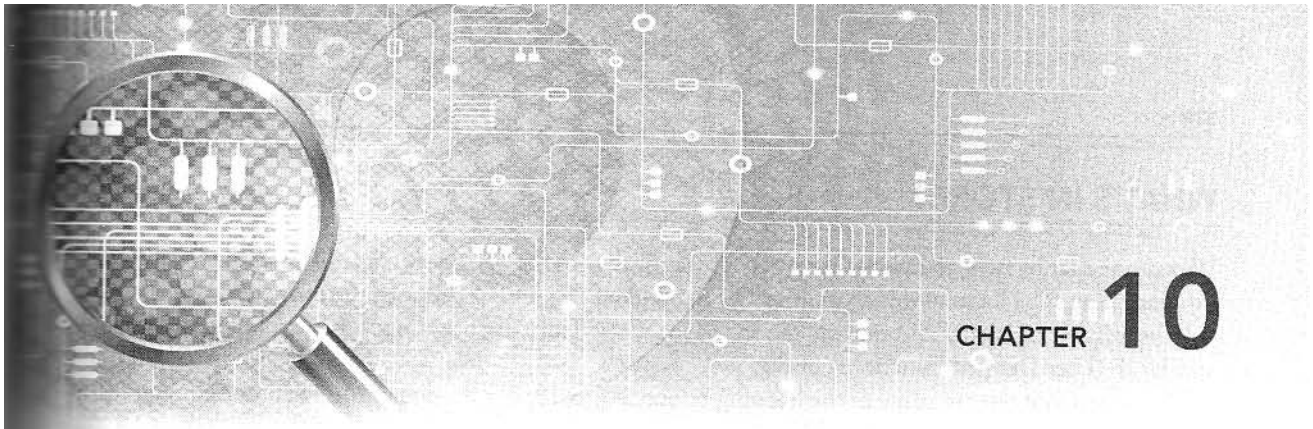
Objective: To learn about HiveQL statement.

Problem Description:

Create a data file for below schemas:

- **Order:** CustomerId, ItemId, ItemName, OrderDate, DeliveryDate
- **Customer:** CustomerId, CustomerName, Address, City, State, Country

1. Create a table for Order and Customer Data.
2. Write a HiveQL to find number of items bought by each customer.



CHAPTER 10

Introduction to Pig

BRIEF CONTENTS

- What's in Store?
- What is Pig?
  - Key Features of Pig
- The Anatomy of Pig
- Pig on Hadoop
- Pig Philosophy
- Use Case for Pig: ETL Processing
- Pig Latin Overview
  - Pig Latin Statements
  - Pig Latin: Keywords
  - Pig Latin: Identifiers
  - Pig Latin: Comments
  - Pig Latin: Case Sensitivity
  - Operators in Pig Latin
- Data Types in Pig
  - Simple Data Types
  - Complex Data Types
- Running Pig
  - Interactive Mode
  - Batch Mode
- Execution Modes of Pig
  - Local Mode
  - MapReduce Mode
- HDFS Commands
- Relational Operators
- EVAL Function
- Complex Data Types
  - Tuple
  - Map
- Piggy Bank
- User-Defined Functions (UDF)
- Parameter Substitution
- Diagnostic Operator
- Word Count Example using Pig
- When to use Pig?
- When NOT to use Pig?
- Pig at Yahoo!
- Pig versus Hive

"If you can't explain it simply, you don't understand it well enough."

– Albert Einstein, Physicist

WHAT'S IN STORE?

We assume that by now you would have become familiar with the basic concepts of HDFS and MapReduce Programming. The focus of this chapter will be to build on this knowledge to perform analysis using Pig. We will discuss few relational and eval operators of Pig. We will also discuss Complex Data Types, Piggy Bank, and UDF (User Defined Functions) of Pig.

We suggest you refer to some of the learning resources provided at the end of this chapter for better learning. We also suggest you to practice "Test Me" exercises.

10.1 WHAT IS PIG?

Apache Pig is a platform for data analysis. It is an alternative to MapReduce Programming. Pig was developed as a research project at Yahoo.

10.1.1 Key Features of Pig

1. It provides an **engine** for executing **data flows** (how your data should flow). Pig processes data in parallel on the Hadoop cluster.
2. It provides a language called "**Pig Latin**" to express data flows.
3. Pig Latin contains operators for many of the traditional data operations such as join, filter, sort, etc.
4. It allows users to develop their own functions (User Defined Functions) for reading, processing, and writing data.

10.2 THE ANATOMY OF PIG

The main components of Pig are as follows:

1. Data flow language (**Pig Latin**).
2. Interactive shell where you can type Pig Latin statements (**Grunt**).
3. Pig interpreter and execution engine.

Refer Figure 10.1.

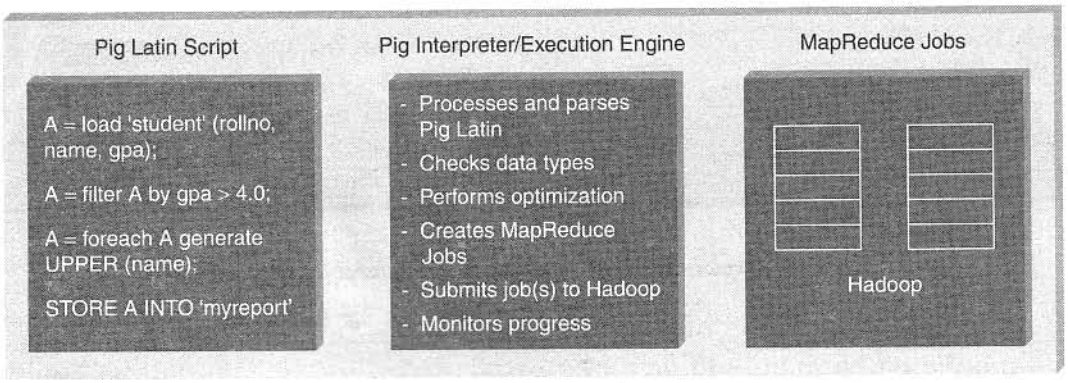


Figure 10.1 The anatomy of Pig.

10.3 PIG ON HADOOP

Pig runs on Hadoop. Pig uses both Hadoop Distributed File System and MapReduce Programming. By default, Pig reads input files from HDFS. Pig stores the intermediate data (data produced by MapReduce jobs) and the output in HDFS. However, Pig can also read input from and place output to other sources.

Pig supports the following:

1. HDFS commands.
2. UNIX shell commands.
3. Relational operators.
4. Positional parameters.
5. Common mathematical functions.
6. Custom functions.
7. Complex data structures.

10.4 PIG PHILOSOPHY

Figure 10.2 describes the Pig philosophy.

1. **Pigs Eat Anything:** Pig can process different kinds of data such as structured and unstructured data.
2. **Pigs Live Anywhere:** Pig not only processes files in HDFS, it also processes files in other sources such as files in the local file system.
3. **Pigs are Domestic Animals:** Pig allows you to develop user-defined functions and the same can be included in the script for complex operations.
4. **Pigs Fly:** Pig processes data quickly.

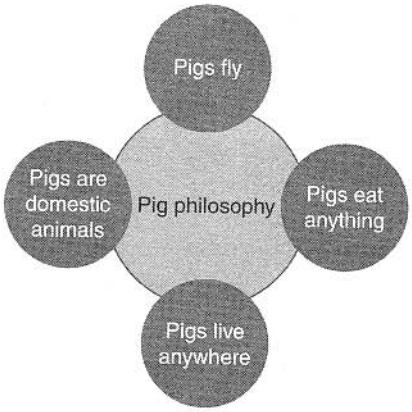


Figure 10.2 Pig philosophy.

10.5 USE CASE FOR PIG: ETL PROCESSING

Pig is widely used for "ETL" (Extract, Transform, and Load). Pig can extract data from different sources such as ERP, Accounting, Flat Files, etc. Pig then makes use of various operators to perform transformation on the data and subsequently loads it into the data warehouse. Refer Figure 10.3.

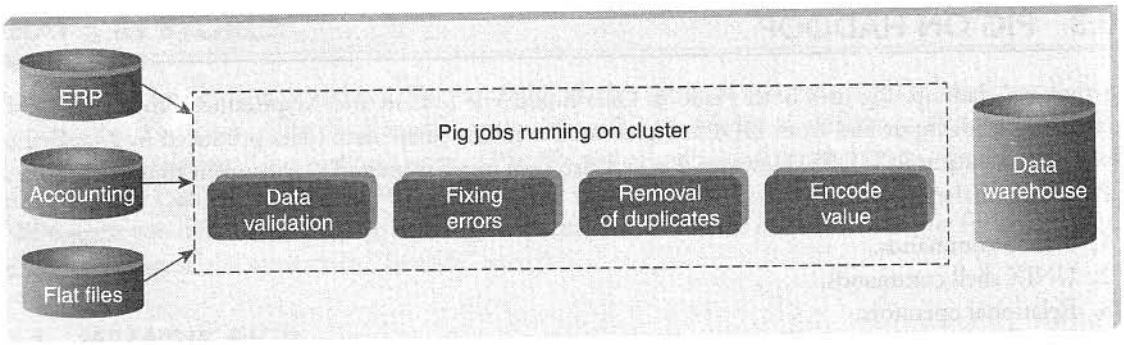


Figure 10.3 Pig: ETL Processing.

10.6 PIG LATIN OVERVIEW

10.6.1 Pig Latin Statements

1. Pig Latin statements are basic constructs to process data using Pig.
2. Pig Latin statement is an operator.
3. An operator in Pig Latin takes a relation as input and yields another relation as output.
4. Pig Latin statements include schemas and expressions to process data.
5. Pig Latin statements should end with a semi-colon.

Pig Latin Statements are generally ordered as follows:

1. **LOAD** statement that reads data from the file system.
2. Series of statements to perform transformations.
3. **DUMP** or **STORE** to display/store result.

The following is a simple Pig Latin script to load, filter, and store "student" data.

```
A = load 'student' (rollno, name, gpa);
A = filter A by gpa > 4.0;
A = foreach A generate UPPER (name);
STORE A INTO 'myreport'
```

Note: In the above example **A** is a relation and NOT a variable.

10.6.2 Pig Latin: Keywords

Keywords are reserved. It cannot be used to name things.

10.6.3 Pig Latin: Identifiers

1. Identifiers are names assigned to fields or other data structures.
2. It should begin with a letter and should be followed only by letters, numbers, and underscores.

Table 10.1 Valid and invalid identifiers

Valid Identifier	**Y**	**A1**	**A1_2014**	**Sample**
Invalid Identifier	**5**	**Sales$**	**Sales%**	**_Sales**

Table 10.1 describes valid and invalid identifiers.

10.6.4 Pig Latin: Comments

In Pig Latin two types of comments are supported:

1. Single line comments that begin with "--".
2. Multiline comments that begin with "/* and end with */".

10.6.5 Pig Latin: Case Sensitivity

1. Keywords are *not* case sensitive such as LOAD, STORE, GROUP, FOREACH, DUMP, etc.
2. Relations and paths are case-sensitive.
3. Function names are case sensitive such as PigStorage, COUNT.

10.6.6 Operators in Pig Latin

Table 10.2 describes operators in Pig Latin.

Table 10.2 Operators in Pig Latin

Arithmetic	Comparison	Null	Boolean
+	==	IS NULL	AND
−	!=	IS NOT NULL	OR
*	<		NOT
/	>		
%	<=		
	>=		

10.7 DATA TYPES IN PIG

10.7.1 Simple Data Types

Table 10.3 describes simple data types supported in Pig. In Pig, fields of unspecified types are considered as an array of bytes which is known as bytearray.

Null: In Pig Latin, **NULL** denotes a value that is unknown or is non-existent.

10.7.2 Complex Data Types

Table 10.4 describes complex data types in Pig.

Table 10.3 Simple data types supported in Pig

Name	Description
Int	Whole numbers
Long	Large whole numbers
Float	Decimals
Double	Very precise decimals
Chararray	Text strings
Bytearray	Raw bytes
Datetime	Datetime
Boolean	true or false

Table 10.4 Complex data types in Pig

Name	Description
Tuple	An ordered set of fields. **Example: (2,3)**
Bag	A collection of tuples. **Example: {(2,3),(7,5)}**
map	key, value pair **(open # Apache)**

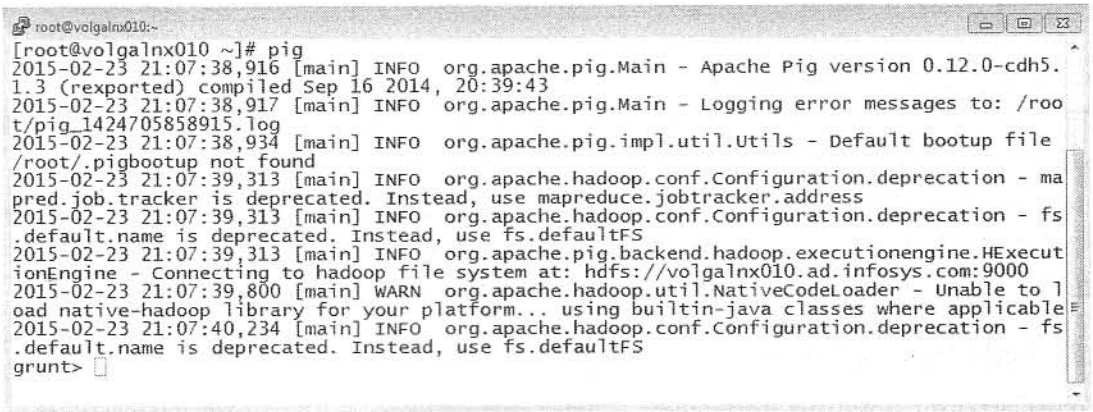
10.8 RUNNING PIG

You can run Pig in two ways:

1. Interactive Mode.
2. Batch Mode.

10.8.1 Interactive Mode

You can run Pig in interactive mode by invoking **grunt** shell. Type pig to get grunt shell as shown below.



Once you get the grunt prompt, you can type the Pig Latin statement as shown below.

```
grunt> A = load '/pigdemo/student.tsv' as (rollno, name, gpa);
grunt> DUMP A;
```

Here, the path refers to HDFS path and DUMP displays the result on the console as shown below.

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
(1004,Scott,4.2)
(1005,Joshi,3.5)
grunt>
```

10.8.2 Batch Mode

You need to create **"Pig Script"** to run pig in batch mode. Write Pig Latin statements in a file and save it with **.pig** extension.

10.9 EXECUTION MODES OF PIG

You can execute pig in two modes:

1. Local Mode.
2. MapReduce Mode.

10.9.1 Local Mode

To run pig in local mode, you need to have your files in the local file system.

Syntax:

```
pig –x local filename
```

10.9.2 MapReduce Mode

To run pig in MapReduce mode, you need to have access to a Hadoop Cluster to read /write file. This is the default mode of Pig.

Syntax:

```
pig filename
```

10.10 HDFS COMMANDS

You can work with all HDFS commands in Grunt shell. For example, you can create a directory as shown below.

```
grunt> fs -mkdir /piglatindemos;
grunt>
```

The sections have been designed as follows:

Objective: What is it that we are trying to achieve here?
Input: What is the input that has been given to us to act upon?
Act: The actual statement/command to accomplish the task at hand.
Outcome: The result/output as a consequence of executing the statement.

10.11 RELATIONAL OPERATORS

10.11.1 FILTER

FILTER operator is used to select tuples from a relation based on specified conditions.

Objective: Find the tuples of those student where the GPA is greater than 4.0.
Input:
Student (rollno:int,name:chararray,gpa:float)
Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = filter A by gpa > 4.0;
DUMP B;
```

Output:

```
(1003,Smith,4.5)
(1004,Scott,4.2)
[root@volgalnx010 pigdemos]#
```

10.11.2 FOREACH

Use ***FOREACH*** when you want to do data transformation based on columns of data.

Objective: Display the name of all students in uppercase.
Input:
Student (rollno:int,name:chararray,gpa:float)
Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = foreach A generate UPPER (name);
DUMP B;
```

Output:

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
[root@volgalnx010 pigdemos]#
```

10.11.3 GROUP

GROUP operator is used to group data.

Objective: Group tuples of students based on their GPA.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = GROUP A BY gpa;
DUMP B;
```

Output:

```
(3.0,{(1001,John,3.0),(1001,John,3.0)})
(3.5,{(1005,Joshi,3.5),(1005,Joshi,3.5)})
(4.0,{(1008,James,4.0),(1002,Jack,4.0)})
(4.2,{(1007,David,4.2),(1004,Scott,4.2)})
(4.5,{(1006,Alex,4.5),(1003,Smith,4.5)})
[root@volgalnx010 pigdemos]#
```

10.11.4 DISTINCT

DISTINCT operator is used to remove duplicate tuples. In Pig, DISTINCT operator works on the entire tuple and NOT on individual fields.

Objective: To remove duplicate tuples of students.

Input:

Student (rollno:int,name:chararray,gpa:float)

Input:

1001	John	3.0
1002	Jack	4.0
1003	Smith	4.5
1004	Scott	4.2
1005	Joshi	3.5
1006	Alex	4.5
1007	David	4.2
1008	James	4.0
1001	John	3.0
1005	Joshi	3.5

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = DISTINCT A;
DUMP B;
```

Output:

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
(1004,Scott,4.2)
(1005,Joshi,3.5)
(1006,Alex,4.5)
(1007,David,4.2)
(1008,James,4.0)
[root@volgalnx010 pigdemos]#
```

10.11.5 LIMIT

LIMIT operator is used to limit the number of output tuples.

Objective: Display the first 3 tuples from the "student" relation.
Input:
Student (rollno:int,name:chararray,gpa:float)
Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = LIMIT A 3;
DUMP B;
```

Output:

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
[root@volgalnx010 pigdemos]#
```

10.11.6 ORDER BY

ORDER BY is used to sort a relation based on specific value.

Objective: Display the names of the students in Ascending Order.
Input:
Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = ORDER A BY name;
DUMP B;
```

Output:

```
(1006,Alex,4.5)
(1007,David,4.2)
(1002,Jack,4.0)
(1008,James,4.0)
(1001,John,3.0)
(1001,John,3.0)
(1005,Joshi,3.5)
(1005,Joshi,3.5)
(1004,Scott,4.2)
(1003,Smith,4.5)
[root@volgalnx010 pigdemos]#
```

10.11.7 JOIN

It is used to join two or more relations based on values in the common field. It always performs inner Join.

Objective: To join two relations namely, "student" and "department" based on the values contained in the "rollno" column.

Input:

Student (rollno:int,name:chararray,gpa:float)

Department(rollno:int,deptno:int,deptname:chararray)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = load '/pigdemo/department.tsv' as (rollno:int, deptno:int,deptname:chararray);
C = JOIN A BY rollno, B BY rollno;
DUMP C;
DUMP B;
```

Output:

```
(1001,John,3.0,1001,101,B.E.)
(1001,John,3.0,1001,101,B.E.)
(1002,Jack,4.0,1002,102,B.Tech)
(1003,Smith,4.5,1003,103,M.Tech)
(1004,Scott,4.2,1004,104,MCA)
(1005,Joshi,3.5,1005,105,MBA)
(1005,Joshi,3.5,1005,105,MBA)
(1006,Alex,4.5,1006,101,B.E)
(1007,David,4.2,1007,104,MCA)
(1008,James,4.0,1008,102,B.Tech)
[root@volgalnx010 pigdemos]#
```

10.11.8 UNION

It is used to merge the contents of two relations.

Objective: To merge the contents of two relations "student" and "department".

Input:

Student (rollno:int,name:chararray,gpa:float)

Department(rollno:int,deptno:int,deptname:chararray)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno, name, gp);
B = load '/pigdemo/department.tsv' as (rollno, deptno,deptname);
C = UNION A,B;
STORE C INTO '/pigdemo/uniondemo';
DUMP B;
```

Output:

"Store" is used to save the output to a specified path. The output is stored in two files: part-m-00000 contains "student" content and part-m-00001 contains "department" content.

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
_SUCCESS	file	0 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup
part-m-00000	file	146 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup
part-m-00001	file	114 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup

File: /pigdemo/uniondemo/part-m-00000

Goto : /pigdemo/uniondemo go

Go back to dir listing
Advanced view/download options

```
1001    John    3.0
1002    Jack    4.0
1003    Smith   4.5
1004    Scott   4.2
1005    Joshi   3.5
1006    Alex    4.5
1007    David   4.2
1008    James   4.0
1001    John    3.0
1005    Joshi   3.5
```

File: /pigdemo/uniondemo/part-m-00001

Goto : /pigdemo/uniondemo go

Go back to dir listing
Advanced view/download options

```
1001    101     B.E.
1002    102     B.Tech
1003    103     M.Tech
1004    104     MCA
1005    105     MBA
1006    101     B.E
1007    104     MCA
1008    102     B.Tech
```

10.11.9 SPLIT

It is used to partition a relation into two or more relations.

Objective: To partition a relation based on the GPAs acquired by the students.

- GPA = 4.0, place it into relation X.
- GPA is < 4.0, place it into relation Y.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
SPLIT A INTO X IF gpa==4.0, Y IF gpa<=4.0;
DUMP X;
```

Output: Relation X

```
(1002,Jack,4.0)
(1008,James,4.0)
[root@volgalnx010 pigdemos]#
```

Output: Relation Y

```
(1001,John,3.0)
(1002,Jack,4.0)
(1005,Joshi,3.5)
(1008,James,4.0)
(1001,John,3.0)
(1005,Joshi,3.5)
[root@volgalnx010 pigdemos]#
```

10.11.10 SAMPLE

It is used to select random sample of data based on the specified sample size.

Objective: To depict the use of *SAMPLE*.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = SAMPLE A 0.01;
DUMP B;
```

10.12 EVAL FUNCTION

10.12.1 AVG

AVG is used to compute the average of numeric values in a single column bag.

Objective: To calculate the average marks for each student.
Input:
Student (studname:chararray,marks:int)
Act:

```
A = load '/pigdemo/student.csv' USING PigStorage (',') as (studname:chararray,marks:int);
B = GROUP A BY studname;
C = FOREACH B GENERATE A.studname, AVG(A.marks);
DUMP C;
```

Output:

```
({(Jack),(Jack),(Jack),(Jack)},39.75)
({(John),(John),(John),(John)},39.0)
[root@volgalnx010 pigdemos]#
```

Note: You need to use PigStorage function if you wish to manipulate files other than .tsv.

10.12.2 MAX

MAX is used to compute the maximum of numeric values in a single column bag.

Objective: To calculate the maximum marks for each student.
Input:
Student (studname:chararray,marks:int)
Act:

```
A = load '/pigdemo/student.csv' USING PigStorage (',') as (studname:chararray, marks:int);
B = GROUP A BY studname;
C = FOREACH B GENERATE A.studname, MAX(A.marks);
DUMP C;
```

Output:

```
({(Jack),(Jack),(Jack),(Jack)},46)
({(John),(John),(John),(John)},45)
[root@volgalnx010 pigdemos]#
```

Note: Similarly, you can try the MIN and the SUM functions as well.

10.12.3 COUNT

COUNT is used to count the number of elements in a bag.

Objective: To count the number of tuples in a bag.
Input:
Student (studname:chararray,marks:int)
Act:

```
A = load '/pigdemo/student.csv' USING PigStorage (',') as (studname:chararray, marks:int);
B = GROUP A BY studname;
C = FOREACH B GENERATE  A. studname,COUNT(A);
DUMP C;
```

Output:

```
({(Jack),(Jack),(Jack),(Jack)},4)
({(John),(John),(John),(John)},4)
[root@volgalnx010 pigdemos]#
```

Note: The default file format of Pig is .tsv file. Use PigStorage() to manipulate files other than .tsv file.

10.13 COMPLEX DATA TYPES

10.13.1 TUPLE

A *TUPLE* is an ordered collection of fields.

Objective: To use the complex data type "Tuple" to load data.
Input:

```
(John,12)      (Jack,13)
(James,7)      (Joseph,5)
(Smith,8)      (Scott,12)
```

Act:

```
A = LOAD '/root/pigdemos/studentdata.tsv' AS (t1:tuple(t1a:chararray,
t1b:int),t2:tuple(t2a:chararray,t2b:int));
B = FOREACH A GENERATE t1.t1a, t1.t1b,t2.$0,t2.$1;
DUMP B;
```

Output:

```
(John,12,Jack,13)
(James,7,Joseph,5)
(Smith,8,Scott,12)
[root@volgalnx010 pigdemos]#
```

Note: You can refer to the field using Positional Notation as shown above. The Positional Notation is denoted by $ sign and the position starts with 0 (e.g., $0).

10.13.2 MAP

MAP represents a key/value pair.

Objective: To depict the complex data type "map".

Input:

```
John    [city#Bangalore]
Jack    [city#Pune]
James   [city#Chennai]
```

Act:

```
A = load '/root/pigdemos/studentcity.tsv' Using PigStorage as
(studname:chararray,m:map[chararray]);
B = foreach A generate m#'city' as CityName:chararray;
DUMP B
```

Output:

```
(Bangalore)
(Pune)
(Chennai)
[root@volgalnx010 pigdemos]#
```

10.14 PIGGY BANK

Pig user can use Piggy Bank functions in Pig Latin script and they can also share their functions in Piggy Bank.

Objective: To use Piggy Bank string UPPER function.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
register '/root/pigdemos/piggybank-0.12.0.jar';
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
upper = foreach A generate
org.apache.pig.piggybank.evaluation.string.UPPER(name);
DUMP upper;
```

Output:

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
(ALEX)
(DAVID)
(JAMES)
(JOHN)
(JOSHI)
[root@volgalnx010 pigdemos]#
```

Note: You need to use the "**register**" keyword to use Piggy Bank jar function in your pig script.

10.15 USER-DEFINED FUNCTIONS (UDF)

Pig allows you to create your own function for complex analysis.

Objective: To depict user-defined function.

Java Code to convert name into uppercase:

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;
public class UPPER extends EvalFunc<String>
  {
    public String exec(Tuple input) throws IOException {
      if (input == null || input.size() == 0)
        return null;
        try{
          String str = (String)input.get(0);
          return str.toUpperCase();
        }catch(Exception e){
          throw WrappedIOException.wrap("Caught exception processing input row ", e);
      }
    }
  }
```

Note: Convert above java class into jar to include this function into your code.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
register /root/pigdemos/myudfs.jar;
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Output:

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
(ALEX)
(DAVID)
(JAMES)
(JOHN)
(JOSHI)
[root@volgalnx010 pigdemos]#
```

10.16 PARAMETER SUBSTITUTION

Pig allows you to pass parameters at runtime.

Objective: To depict parameter substitution.
Input:
Student (rollno:int,name:chararray,gpa:float)
Act:

```
A = load '$student' as (rollno:int, name:chararray, gpa:float);
DUMP A;
```

Execute:

```
pig –param student=/pigdemo/student.tsv parameterdemo.pig
```

Output:

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
(1004,Scott,4.2)
(1005,Joshi,3.5)
(1006,Alex,4.5)
(1007,David,4.2)
(1008,James,4.0)
(1001,John,3.0)
(1005,Joshi,3.5)
[root@volgalnx010 pigdemos]#
```

10.17 DIAGNOSTIC OPERATOR

It returns the schema of a relation.

Objective: To depict the use of ***DESCRIBE***.
Input:
Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
DESCRIBE A;
```

Output:

```
A: {rollno: int,name: chararray,gpa: float}
[root@volgalnx010 pigdemos]#
```

10.18 WORD COUNT EXAMPLE USING PIG

Objective: To count the occurrence of similar words in a file.

Input:

```
Welcome to Hadoop Session
Introduction to Hadoop
Introducing Hive
Hive Session
Pig Session
```

Act:

```
lines = LOAD '/root/pigdemos/lines.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

Output:

```
(to,2)
(Pig,1)
(Hive,2)
(Hadoop,2)
(Session,3)
(Welcome,1)
(Introducing,1)
(Introduction,1)
[root@volgalnx010 pigdemos]#
```

Note:

TOKENIZE splits the line into a field for each word.

FLATTEN will take the collection of records returned by TOKENIZE and produce a separate record for each one, calling the single field in the record word.

10.19 WHEN TO USE PIG?

Pig can be used in the following situations:

1. When your data loads are time sensitive.
2. When you want to process various data sources.
3. When you want to get analytical insights through sampling.

10.20 WHEN NOT TO USE PIG?

Pig should not be used in the following situations:

1. When your data is completely in the unstructured form such as video, text, and audio.
2. When there is a time constraint because Pig is slower than MapReduce jobs.

10.21 PIG AT YAHOO!

Yahoo uses Pig for two things:

1. **In Pipelines**, to fetch log data from its web servers and to perform cleansing to remove companies interval views and clicks.
2. **In Research**, script is used to test a theory. Pig provides facility to integrate Perl or Python script which can be executed on a huge dataset.

10.22 PIG versus HIVE

Features	Pig	Hive
Used By	Programmers and Researchers	Analyst
Used For	Programming	Reporting
Language	Procedural data flow language	SQL Like
Suitable For	Semi - Structured	Structured
Schema/Types	Explicit	Implicit
UDF Support	YES	YES
Join/Order/Sort	YES	YES
DFS Direct Access	YES (Implicit)	YES (Explicit)
Web Interface	YES	NO
Partitions	YES	NO
Shell	YES	YES

REMIND ME

- Apache Pig is a platform for data analysis. It is an alternative to MapReduce Programming.
- It provides an **engine** for executing **data flows** (how your data should flow). Pig processes data in parallel on the Hadoop cluster.
- It provides a language called "**Pig Latin**" to express data flows.
- The main components of Pig are as follows:
  - Data flow language (**Pig Latin**).
  - Interactive shell where you can type Pig Latin statements (**Grunt**).
  - Pig interpreter and execution engine.
- You can run Pig in two ways:
  - Interactive Mode.
  - Batch Mode.

POINT ME (BOOK)

- Programming Pig, Alan Gates, O'REILY.

CONNECT ME (INTERNET RESOURCES)

- http://pig.apache.org/docs/r0.12.0/index.html
- http://www.edureka.co/blog/introduction-to-pig/
- http://www.edureka.co/blog/pig-vs-hive/

TEST ME

A. Fill Me

1. Pig is a _____ language.
2. In Pig, _____ is used to specify data flow.
3. Pig provides an _____ to execute data flow.
4. _____, _____ are execution modes of Pig.
5. The interactive mode of Pig is _____.
6. _____ and _____ are case sensitive in Pig.
7. _____, _____, _____ are Complex Data Types of Pig.
8. Pig is used in _____ process.

Answers:

1. Scripting
2. Pig Latin
3. Pig Engine
4. Local Mode, MapReduce Mode
5. Grunt
6. Fields and Aliases
7. Bag, Tuple, Map
8. ETL

B. Match Me

Column A	Column B
Map	Hadoop Cluster
Bag	An Ordered Collection of Fields
Local Mode	Collection of Tuples
Tuple	Key/Value Pair
MapReduce Mode	Local File System

Answers:

Column A	Column B
Map	Key/Value Pair
Bag	Collection of Tuples
Local Mode	Local File System
Tuple	An Ordered Collection of Fields
MapReduce Mode	Hadoop Cluster

C. True or False

1. PigStorage() function is case sensitive.
2. Local Mode is the default mode of Pig.
3. DISTINCT Keyword removes duplicate fields.
4. LIMIT keyword is used to display limited number of tuples in Pig.
5. ORDER BY is used for sorting.

Answers:

1. True
2. False
3. False
4. True
5. True

ASSIGNMENTS FOR HANDS-ON PRACTICE

ASSIGNMENT 1: SPLIT

Objective: To learn about SPLIT relational operator.

Problem Description:
Write a Pig Script to split customers for reward program based on their life time values.

Input:

Customers	Life Time Value
Jack	25000
Smith	8000
David	35000
John	15000
Scott	10000
Joshi	28000
Ajay	12000
Vinay	30000
Joseph	21000

- If Life Time Value is >1000 and <= 2000 → Sliver Program.
- If Life Time Value is >20000 → Gold Program.

ASSIGNMENT 2: GROUP

Objective: To learn about GROUP relational operator.

Problem Description:
Create a data file for below schemas:

- **Order:** CustomerId, ItemId, ItemName, OrderDate, DeliveryDate
- **Customer:** CustomerId, CustomerName, Address, City, State, Country

1. Load Order and Customer Data.
2. Write a Pig Latin Script to determine number of items bought by each customer.

ASSIGNMENT 3: COMPLEX DATA TYPE – BAG

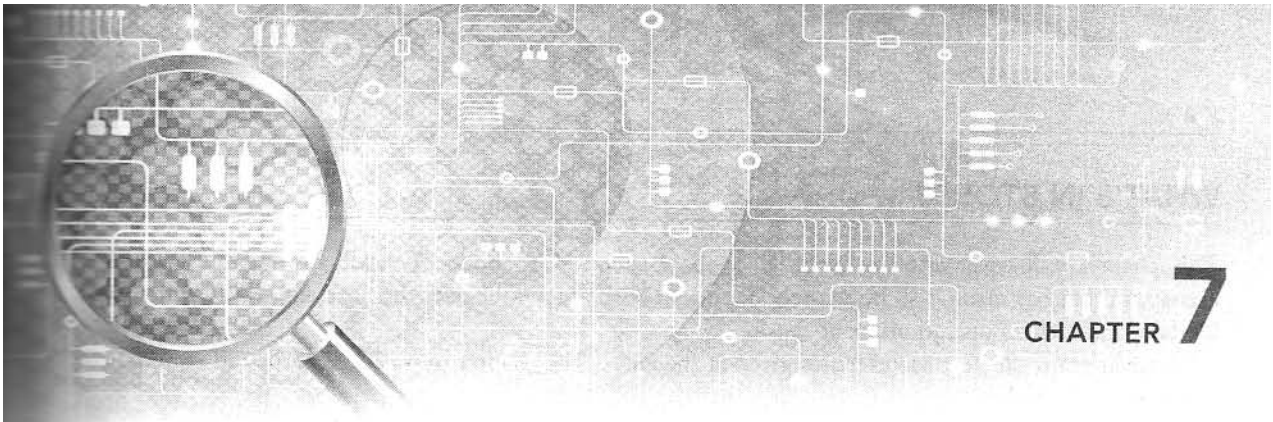
Objective: To learn complex data type – bag in Pig.

Problem Description:

1. Create a file which contains bag dataset as shown below.

User ID	From	To
user1001	user1001@sample.com	{(user003@sample.com),(user004@sample.com), (user006@sample.com)}
user1002	user1002@sample.com	{(user005@sample.com), (user006@sample.com)}
user1003	user1003@sample.com	{(user001@sample.com),(user005@sample.com)}

2. Write a Pig Latin statement to display the names of all users who have sent emails and also a list of all the people that they have sent the email to.
3. Store the result in a file.



CHAPTER 7

Introduction to Cassandra

BRIEF CONTENTS

- What's in Store?
- Apache Cassandra – An Introduction
- Features of Cassandra
  - Peer-to-Peer Network
  - Gossip and Failure Detection
  - Partitioner
  - Replication Factor
  - Anti-Entropy and Read Repair
  - Writes in Cassandra
  - Hinted Handoffs
  - Tunable Consistency: Read Consistency and Write Consistency
- CQL Data Types
- CQLSH
- Keyspaces
- CRUD Operations
  - Insert
  - Update
  - Delete
  - Select
- Collections
  - Set Collection
  - List Collection
  - Map Collection
- Using a Counter
- Time To Live (TTL)
- Alter Commands
  - Alter Table to Change the Data Type of a Column
  - Alter Table to Delete a Column
  - Drop a Table
  - Drop a Database
- Import and Export
  - Export to CSV
  - Import from CSV
  - Import from STDIN
  - Export to STDOUT
- Querying System Tables
- Practice Examples

"Data is a precious thing and will last longer than the systems themselves."

– Tim Berners-Lee, inventor of the World Wide Web.

WHAT'S IN STORE?

This chapter will cover another NoSQL database called "Cassandra". We will explore the features of Cassandra that has made it so immensely popular. The chapter will cover the basic CRUD (Create, Read, Update, and Delete) operations using cqlsh.

Please attempt the Test Me exercises given at the end of the chapter to practice, learn, and comprehend Cassandra effectively.

7.1 APACHE CASSANDRA – AN INTRODUCTION

We shall start this chapter with few points that a reader should know about Cassandra.

1. Apache Cassandra was born at Facebook. After Facebook open sourced the code in 2008, Cassandra became an Apache Incubator project in 2009 and subsequently became a top-level Apache project in 2010.
2. It is built on Amazon's dynamo and Google's BigTable.
3. Cassandra does NOT compromise on availability. Since it does not have a master-slave architecture, there is no question of single point of failure. This proves beneficial for business critical applications that need to be up and running always and cannot afford to go down ever.
4. It is highly scalable (it scales out), high performance distributed database. It distributes and manages gigantic amount of data across commodity servers.

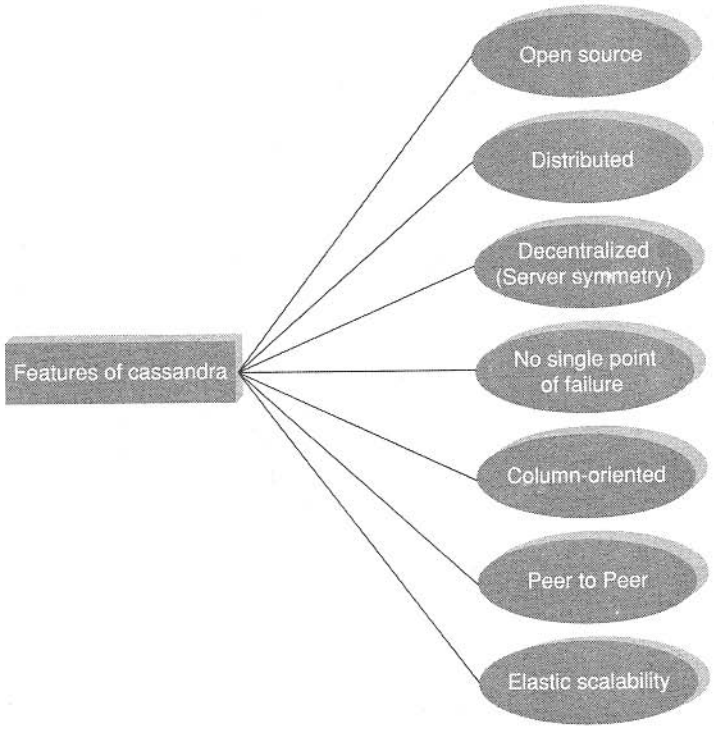


Figure 7.1 Features of Cassandra.

5. It is a column-oriented database designed to support peer-to-peer symmetric nodes instead of the master-slave architecture.
6. It has adherence to the Availability and Partition Tolerance properties of CAP theorem. It takes care of consistency using BASE (Basically Available Soft State Eventual Consistency) approach.

Refer Figure 7.1. Few companies that have successfully deployed Cassandra and have benefitted immensely from it are as follows:

1. Twitter
2. Netflix
3. Cisco
4. Adobe
5. eBay
6. Rackspace

7.2 FEATURES OF CASSANDRA

7.2.1 Peer-to-Peer Network

As with any other NoSQL database, Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware. Cassandra does NOT have a master-slave architecture which means that it does NOT have single point of failure. A node in Cassandra is structurally identical to any other node. Refer Figure 7.2. In case a node fails or is taken offline, it definitely impacts the throughput. However, it is a case of graceful degradation where everything does not come crashing at any given instant owing to a node failure. One can still go about business as usual. It tides over the problem of failure by employing a peer-to-peer distributed system across homogeneous nodes. It ensures that data is distributed across all nodes in the cluster. Each node exchanges information across the cluster every second.

Let us look at how a Cassandra node writes. Each write is written to the commit log sequentially. A write is taken to be successful only if it is written to the commit log. Data is then indexed and pushed to an in-memory structure called "Memtable". When the in-memory data structure, "the Memtable", is full, the contents are flushed to "SSTable" (Sorted String) data file on the disk. The SSTable is immutable and is

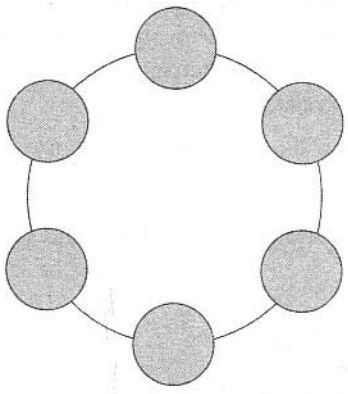


Figure 7.2 Sample Cassandra cluster.

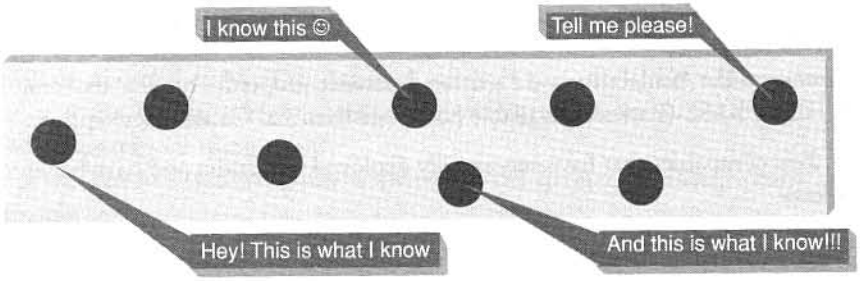


Figure 7.3 Gossip protocol.

append-only. It is stored on disk sequentially and is maintained for each Cassandra table. The partitioning and replication of all writes are performed automatically across the cluster.

7.2.2 Gossip and Failure Detection

Gossip protocol is used for intra-ring communication. It is a peer-to-peer communication protocol which eases the discovery and sharing of location and state information with other nodes in the cluster. Refer Figure 7.3. Although there are quite a few subtleties involved, but at its core it's a simple and robust system. A node only has to send out the communication to a subset of other nodes. For repairing unread data, Cassandra uses what's called an anti-entropy version of the gossip protocol.

7.2.3 Partitioner

A partitioner takes a call on how to distribute data on the various nodes in a cluster. It also determines the node on which to place the very first copy of the data. Basically a partitioner is a hash function to compute the token of the partition key. The partition key helps to identify a row uniquely.

7.2.4 Replication Factor

The replication factor determines the number of copies of data (replicas) that will be stored across nodes in a cluster. If one wishes to store only one copy of each row on one node, they should set the replication factor to one. However, if the need is for two copies of each row of data on two different nodes, one should go with a replication factor of two. The replication factor should ideally be more than one and not more than the number of nodes in the cluster. A replication strategy is employed to determine which nodes to place the data on. Two replication strategies are available:

1. SimpleStrategy.
2. NetworkToplogyStrategy.

The preferred one is NetworkTopologyStrategy as it is simple and supports easy expansion to multiple data centers, should there be a need.

7.2.5 Anti-Entropy and Read Repair

A cluster is made up of several nodes. Since the cluster is constituted of commodity hardware, it is prone to failure. In order to achieve fault tolerance, a given piece of data is replicated on one or more nodes. A client

can connect to any node in the cluster to read data. How many nodes will be read before responding to the client is based on the consistency level specified by the client. If the client-specified consistency is not met, the read operation blocks. There is a possibility that few of the nodes may respond with an out-of-date value. In such a case, Cassandra will initiate a read repair operation to bring the replicas with stale values up to date.

For repairing unread data, Cassandra uses an anti-entropy version of the gossip protocol. Anti-entropy implies comparing all the replicas of each piece of data and updating each replica to the newest version. The read repair operation is performed either before or after returning the value to the client as per the specified consistency level.

7.2.6 Writes in Cassandra

Let us look at behind the scene activities. Here is a client that initiates a write request. Where does his write get written to? It is first written to the commit log. A write is taken as successful only if it is written to the commit log. The next step is to push the write to a memory resident data structure called Memtable. A threshold value is defined in the Memtable. When the number of objects stored in the Memtable reaches a threshold, the contents of Memtable are flushed to the disk in a file called SSTable (Sorted String Table). Flushing is a non-blocking operation. It is possible to have multiple Memtables for a single column family. One out of them is current and the rest are waiting to be flushed.

7.2.7 Hinted Handoffs

The first question that arises is: Why Cassandra is all for availability? It works on the philosophy that it will always be available for writes.

Assume that we have a cluster of three nodes – Node A, Node B, and Node C. Node C is down for some reason. Refer Figure 7.4. We are maintaining a replication factor of 2 which implies that two copies of each row will be stored on two different nodes. The client makes a write request to Node A. Node A is the coordinator and serves as a proxy between the client and the nodes on which the replica is to be placed. The client

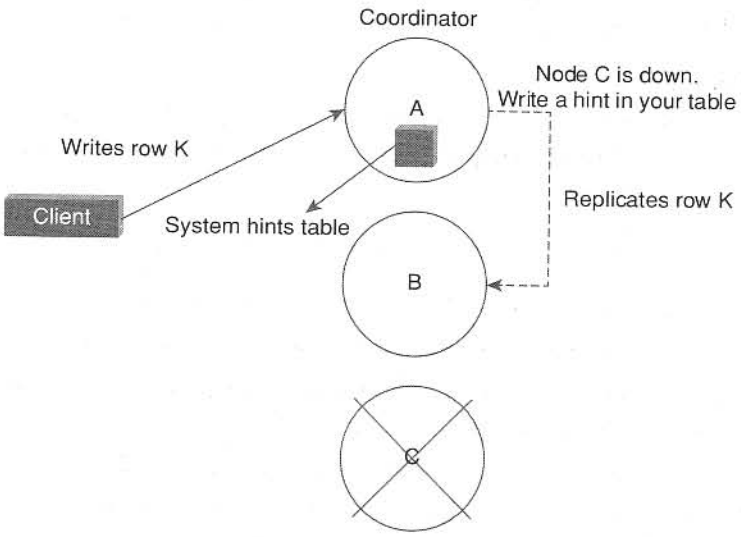


Figure 7.4 Depiction of hinted handoffs.

writes Row K to Node A. Node A then writes Row K to Node B and stores a hint for Node C. The hint will have the following information:

1. Location of the node on which the replica is to be placed.
2. Version metadata.
3. The actual data.

When Node C recovers and is back to the functional self, Node A reacts to the hint by forwarding the data to Node C.

7.2.8 Tunable Consistency

One of the features of Cassandra that has made it immensely popular is its ability to utilize tunable consistency. The database systems can go for either strong consistency or eventual consistency. Cassandra can cash in on either flavor of consistency depending on the requirements. In a distributed system, we work with several servers in the system. Few of these servers are in one data center and others in other data centers. Let us take a look at what it means by strong consistency and eventual consistency.

1. **Strong consistency:** If we work with strong consistency, it implies that each update propagates to all locations where that piece of data resides. Let us assume a single data center setup. Strong consistency will ensure that all of the servers that should have a copy of the data, will have it, before the client is acknowledged with a success. If we are wondering whether it will impact performance, yes it will. It will cost a few extra milliseconds to write to all servers.
2. **Eventual consistency:** If we work with eventual consistency, it implies that the client is acknowledged with a success as soon as a part of the cluster acknowledges the write. When should one go for eventual consistency? The choice is fairly obvious... when application performance matters the most. Example: A single server acknowledges the write and then begins propagating the data to other servers.

7.2.8.1 Read Consistency

Let us understand what the read consistency level means. It means how many replicas must respond before sending out the result to the client application. There are several read consistency levels as mentioned in Table 7.1.

7.2.8.2 Write Consistency

Let us understand what the write consistency level means. It means on how many replicas write must succeed before sending out an acknowledgement to the client application. There are several write consistency levels as mentioned in Table 7.2.

Table 7.1 Read consistency levels in Cassandra

ONE	Returns a response from the closest node (replica) holding the data.
QUORUM	Returns a result from a quorum of servers with the most recent timestamp for the data.
LOCAL_ QUORUM	Returns a result from a quorum of servers with the most recent timestamp for the data in the same data center as the coordinator node.
EACH_QUORUM	Returns a result from a quorum of servers with the most recent timestamp in all data centers.
ALL	This provides the highest level of consistency of all levels and the lowest level of availability of all levels. It responds to a read request from a client after all the replica nodes have responded.

Table 7.2 Write consistency levels in Cassandra

ALL	This is the highest level of consistency of all levels as it necessitates that a write must be written to the commit log and Memtable on all replica nodes in the cluster.
EACH_QUORUM	A write must be written to the commit log and Memtable on a quorum of replica nodes in *all* data centers.
QUORUM	A write must be written to the commit log and Memtable on a quorum of replica nodes.
LOCAL_QUORUM	A write must be written to the commit log and Memtable on a quorum of replica nodes in the same data center as the coordinator node. This is to avoid latency of inter-data center communication.
ONE	A write must be written to the commit log and Memtable of at least one replica node.
TWO	A write must be written to the commit log and Memtable of at least two replica nodes.
THREE	A write must be written to the commit log and Memtable of at least three replica nodes.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local data center.

7.3 CQL DATA TYPES

Refer Table 7.3 for built-in data types for columns in CQL.

Table 7.3 Built-in data types in Cassandra

Int	32 bit signed integer
Bigint	64 bit signed long
Double	64-bit IEEE-754 floating point
Float	32-bit IEEE-754 floating point
Boolean	True or false
Blob	Arbitrary bytes, expressed in hexadecimal
Counter	Distributed counter value
Decimal	Variable – precision integer
List	A collection of one or more ordered elements
Map	A JSON style array of elements
Set	A collection of one or more elements
Timestamp	Date plus time
Varchar	UTF 8 encoded string
Varint	Arbitrary-precision integers
Text	UTF 8 encoded string

7.4 CQLSH

7.4.1 Logging into cqlsh

The below screenshot depicts the cqlsh command prompt after logging in, using cqlsh succeeds.

```
C:\windows\system32\cmd.exe - Python cqlsh

d:\apache-cassandra-2.0.0\apache-cassandra-2.0.0\apache-cassandra-2.0.0\bin>Python cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.0.0 | Cassandra 2.0.0 | CQL spec 3.1.0 | Thrift protocol 19.37.0]
Use HELP for help.
cqlsh>
```

The upcoming sections have been designed as follows:

Objective:	What is it that we are trying to achieve here?
Input (optional):	What is the input that has been given to us to act upon?
Act:	The actual statement /command to accomplish the task at hand.
Outcome:	The result/output as a consequence of executing the statement.

Objective: To get help with CQL.

Act:

Help

Outcome:

```
C:\windows\system32\cmd.exe - Python cqlsh

d:\apache-cassandra-2.0.0\apache-cassandra-2.0.0\apache-cassandra-2.0.0\bin>Python cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.0.0 | Cassandra 2.0.0 | CQL spec 3.1.0 | Thrift protocol 19.37.0]
Use HELP for help.
cqlsh> help

Documented shell commands:
==========================
CAPTURE      COPY  DESCRIBE  EXPAND  SHOW    TRACING
CONSISTENCY  DESC  EXIT      HELP    SOURCE

CQL help topics:
================
ALTER                          CREATE_TABLE_OPTIONS  REVOKE
ALTER_ADD                      CREATE_TABLE_TYPES    SELECT
ALTER_ALTER                    CREATE_USER           SELECT_COLUMNFAMILY
ALTER_DROP                     DELETE                SELECT_EXPR
ALTER_RENAME                   DELETE_COLUMNS        SELECT_LIMIT
ALTER_USER                     DELETE_USING          SELECT_TABLE
ALTER_WITH                     DELETE_WHERE          SELECT_WHERE
APPLY                          DROP                  TEXT_OUTPUT
ASCII_OUTPUT                   DROP_COLUMNFAMILY     TIMESTAMP_INPUT
BEGIN                          DROP_INDEX            TIMESTAMP_OUTPUT
BLOB_INPUT                     DROP_KEYSPACE         TRUNCATE
BOOLEAN_INPUT                  DROP_TABLE            TYPES
CREATE                         DROP_USER             UPDATE
CREATE_COLUMNFAMILY            GRANT                 UPDATE_COUNTERS
CREATE_COLUMNFAMILY_OPTIONS    INSERT                UPDATE_SET
CREATE_COLUMNFAMILY_TYPES      LIST                  UPDATE_USING
CREATE_INDEX                   LIST_PERMISSIONS      UPDATE_WHERE
CREATE_KEYSPACE                LIST_USERS            USE
CREATE_TABLE                   PERMISSIONS           UUID_INPUT

cqlsh>
```

7.5 KEYSPACES

What is a keyspace? A keyspace is a container to hold application data. It is comparable to a relational database. It is used to group column families together. Typically, a cluster has one keyspace per application. Replication is controlled on a per keyspace basis. Therefore, data that has different replication requirements should reside on different keyspaces.

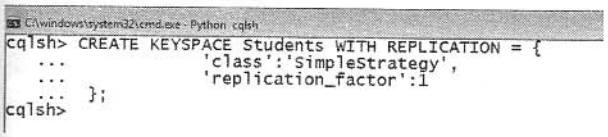
When one creates a keyspace, it is required to specify a strategy class. There are two choices available with us. Either we can specify a "SimpleStrategy" or a "NetworkTopologyStrategy" class. While using Cassandra for evaluation purpose, go with "SimpleStrategy" class and for production usage, work with the "NetworkTopologyStrategy" class.

Objective: To create a keyspace by the name "Students".

Act:

```
CREATE KEYSPACE Students WITH REPLICATION = {
        'class':'SimpleStrategy',
        'replication_factor':1
};
```

Outcome:



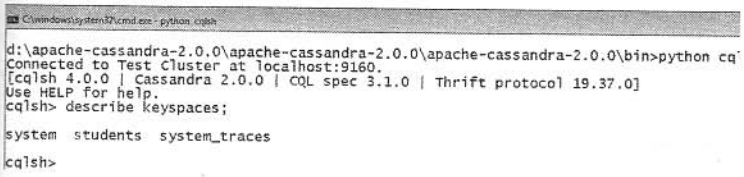
The replication factor stated above in the syntax for creating keyspace is related to the number of copies of keyspace data that is housed in a cluster.

Objective: To describe all the existing keyspaces.

Act:

```
DESCRIBE KEYSPACES;
```

Outcome:



Objective: To get more details on the existing keyspaces such as keyspace name, durable writes, strategy class, strategy options, etc.

Act:

```
SELECT *
        FROM system.schema_keyspaces;
```

Outcome:

```
cqlsh> SELECT * FROM system.schema_keyspaces;

 keyspace_name | durable_writes | strategy_class                                      | strategy_options
---------------+----------------+-----------------------------------------------------+----------------------------
      demo_con |           True | org.apache.cassandra.locator.SimpleStrategy         | {"replication_factor":"3"}
        system |           True |  org.apache.cassandra.locator.LocalStrategy         | {}
system_traces |           True | org.apache.cassandra.locator.SimpleStrategy         | {"replication_factor":"1"}
      students |           True | org.apache.cassandra.locator.SimpleStrategy         | {"replication_factor":"1"}

(4 rows)
```

Note: Cassandra converted the Students keyspace to lowercase as quotation marks were not used.

Objective: To use the keyspace "Students", use the following command:

Use keyspace_name

Use connects the client session to the specified keyspace.

Act:

```
USE Students;
```

Outcome:

```
C:\windows\system32\cmd.exe - python  cqlsh
cqlsh> use Students;
cqlsh:students>
```

Objective: To create a column family or table by the name "student_info".

Act:

```
CREATE TABLE Student_Info (
   RollNo int PRIMARY KEY,
   StudName text,
   DateofJoining timestamp,
   LastExamPercent double
   );
```

Outcome:

```
cqlsh> use Students;
cqlsh:students> CREATE TABLE Student_Info (
            ...   RollNo int PRIMARY Key,
            ...   StudName text,
            ...   DateofJoining timestamp,
            ...   LastExamPercent double
            ...   );
```

The table "student_info" gets created in the keyspace "students".

Note: Tables can have either a single or compound primary key. Always ensure that there is exactly one primary key definition. The primary key, however, can be simple (consisting of a single attribute) or composite (comprising two or more attributes).

Explanation about the composite PRIMARY KEY:

Primary key (column_name1, *column_name2, column_name3* ...)
Primary key ((column_name4, *column_name5*), column_name6, *column_name7* ...)

In the above syntax,
column_name1 is the partition key
column_name2 and column_name3 are the clustering columns.
column_name4 and column_name5 are the partitioning keys
column_name6 and column_name7 are the clustering columns.

The partition key is used to distribute the data in the table across various nodes that constitute the cluster. The clustering columns are used to store data in ascending order on the disk.

Objective: To lookup the names of all tables in the current keyspace, or in all the keyspaces if there is no current keyspace.

Act:

```
DESCRIBE TABLES;
```

Outcome:

```
cqlsh:students> describe tables;

student_info

cqlsh:students>
```

Objective: To describe the table "student_info" use the below command.

Act:

```
DESCRIBE TABLE student_info;
```

Note: The output is a list of CQL commands with the help of which the table "student_info" can be recreated.

Outcome:

```
C:\windows\system32\cmd.exe - python cqlsh
cqlsh:students> describe table student_info;

CREATE TABLE student_info (
  rollno int,
  dateofjoining timestamp,
  lastexampercent double,
  studname text,
  PRIMARY KEY (rollno)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

7.6 CRUD (CREATE, READ, UPDATE, AND DELETE) OPERATIONS

Objective: To insert data into the column family "student_info".

An insert writes one or more columns to a record in Cassandra table atomically. An insert statement does not return an output. One is not required to place values in all the columns; however, it is mandatory to specify all the columns that make up the primary key. The columns that are missing do not occupy any space on disk.

Internally insert and update operations are equal. However, insert does not support counters but update does. Counters will be discussed later in the chapter.

Act:

```
BEGIN BATCH
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
VALUES (1,'Michael Storm','2012-03-29', 69.6)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
VALUES (2,'Stephen Fox','2013-02-27', 72.5)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
VALUES (3,'David Flemming','2014-04-12', 81.7)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
VALUES (4,'Ian String','2012-05-11', 73.4)
APPLY BATCH;
```

Outcome:

```
cqlsh:students> BEGIN BATCH
           ...   INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
           ... VALUES (1,'Michael Storm','2012-03-29',69.6)
           ...   INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
           ... VALUES (2,'Stephen Fox','2013-02-27',72.5)
           ...   INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
           ...  VALUES (3,'David Flemming ','2014-04-12',81.7)
           ...   INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
           ...  VALUES (4,'Ian String','2012-05-11',73.4)
           ... APPLY BATCH;
cqlsh:students>
```

Objective: To view the data from the table "student_info".

Act:

```
SELECT *
      FROM student_info;
```

The above select statement retrieves data from the "student_info" table.

Outcome:

```
cqlsh:students> select * from student_info;

 rollno | dateofjoining                                | lastexampercent | studname
--------+----------------------------------------------+-----------------+-----------------
      1 | 2012-03-29 00:00:00India Standard Time |            69.6 |   Michael Storm
      2 | 2013-02-27 00:00:00India Standard Time |            72.5 |     Stephen Fox
      4 | 2012-05-11 00:00:00India Standard Time |            73.4 |      Ian String
      3 | 2014-04-12 00:00:00India Standard Time |            81.7 | David Flemming

(4 rows)
cqlsh:students>
```

Objective: To view only those records where the RollNo column either has a value 1 or 2 or 3.

Act:

```
SELECT *
        FROM student_info
                WHERE RollNo IN(1,2,3);
```

Note: For the above statement to execute successfully, ensure that the following criteria are satisfied:

1. Either the partition key definition includes the column that is used in the where clause i.e. search criteria.
2. OR the column being used in the where clause, that is, search criteria, has an index defined on it using the CREATE INDEX statement.

Outcome:

```
cqlsh:students> Select * from student_info where RollNo IN(1,2,3);

 rollno | dateofjoining                             | lastexampercent | studname
--------+-------------------------------------------+-----------------+----------------
      1 | 2012-03-29 00:00:00India Standard Time |            69.6 |   Michael Storm
      2 | 2013-02-27 00:00:00India Standard Time |            72.5 |     Stephen Fox
      3 | 2014-04-12 00:00:00India Standard Time |            81.7 | David Flemming

(3 rows)

cqlsh:students>
```

Let us try running a query with "studname" in the where clause. Since "studname" is neither the primary key column nor a column in the primary key definition and also does not have an index defined on it, such a query will lead to error.

We set the stage to resolve the error by creating an index on the "studname" column of the "student_info" table and then subsequently executing the query.

To create an index on the "studname" column of the "student_info" column family use

```
CREATE INDEX ON student_info(studname)
```

To execute the query using the index defined on "studname" column use

```
SELECT *
        FROM student_info
                WHERE studname='Stephen Fox' ;
```

Outcome:

```
cqlsh:students> create index on student_info(studname);
cqlsh:students> select * from student_info where studname='Stephen Fox' ;

 rollno | dateofjoining                             | lastexampercent | studname
--------+-------------------------------------------+-----------------+-------------
      2 | 2013-02-27 00:00:00India Standard Time |            72.5 | Stephen Fox

(1 rows)

cqlsh:students>
```

Objective: Let us create another index on the "LastExamPercent" column of the "student_info" column family.

Act:

```
CREATE INDEX ON student_info(LastExamPercent);
```

Outcome:

```
cqlsh:students> create index on student_info(LastE
cqlsh:students> select * from student_info where LastExamPercent = 81.7;

 rollno | dateofjoining                             | lastexampercent | studname
--------+-------------------------------------------+-----------------+-----------------
      3 | 2014-04-12 00:00:00India Standard Time |            81.7 | David Flemming

(1 rows)
```

Objective: To specify the number of rows returned in the output using limit.

Act:

```
SELECT rollno, hobbies, language, lastexampercent
      FROM student_info LIMIT 2;
```

Outcome:

```
cqlsh:students> select rollno, hobbies, language, lastexampercent from student_info limit 2;

 rollno | hobbies                                | language          | lastexampercent
--------+----------------------------------------+-------------------+-----------------
      1 |                {'Chess, Table Tennis'} | ['Hindi, English'] |            69.6
      4 | {'Lawn Tennis, Table Tennis, Golf'} | ['Hindi, English'] |            73.4

(2 rows)
```

Objective: To use column alias for the column "language" in the "student_info" table. We would like the column heading to be "knows language".

Act:

```
SELECT rollno, language AS "knows language"
      FROM student_info;
```

Outcome:

```
cqlsh:students> select rollno, language as "knows language" from student_info;

 rollno | knows language
--------+-------------------------
      1 |       ['Hindi, English']
      4 |       ['Hindi, English']
      3 | ['Hindi,English,French']

(3 rows)
```

Objective: To update the value held in the "StudName" column of the "student_info" column family to "David Sheen" for the record where the RollNo column has value = 2.

Note: An update updates one or more column values for a given row to the Cassandra table. It does not return anything.

Act:

```
UPDATE student_info SET StudName = 'David Sheen' WHERE RollNo = 2;
```

Outcome:

```
cqlsh:students> UPDATE student_info  SET StudName = 'David Sheen'  WHERE RollNo = 2;
cqlsh:students> select * from student_info where rollno = 2;

 rollno | dateofjoining                                | lastexampercent | studname
--------+----------------------------------------------+-----------------+-------------
      2 | 2013-02-27 00:00:00India Standard Time |            72.5 | David Sheen

(1 rows)

cqlsh:students>
```

Objective: Let us try updating the value of a primary key column.

Act:

```
UPDATE student_info SET rollno=6 WHERE rollno=3;
```

Outcome:

```
cqlsh:students> update student_info set rollno=6 where rollno=3;
Bad Request: PRIMARY KEY part rollno found in SET part
cqlsh:students>
```

Note: It does not allow update to a primary key column.

Objective: Updating more than one column of a row of Cassandra table.

Act:

Step 1: Before the update

```
cqlsh:students> select rollno, studname, lastexampercent from student_info where rollno=3;

 rollno | studname        | lastexampercent
--------+-----------------+-----------------
      3 | David Flemming  |            81.7

(1 rows)
```

Step 2: Applying the update

```
cqlsh:students> select rollno, studname, lastexampercent from student_info where rollno=3;

 rollno | studname | lastexampercent
--------+----------+-----------------
      3 |  Samaira |              85

(1 rows)
```

Step 3: After the update

```
cqlsh:students> DELETE LastExamPercent FROM student_info where RollNo=2;
cqlsh:students> select * from student_info where rollno = 2;

 rollno | dateofjoining                                | lastexampercent | studname
--------+----------------------------------------------+-----------------+-------------
      2 | 2013-02-27 00:00:00India Standard Time |            null | David Sheen

(1 rows)

cqlsh:students>
```

Objective: To delete the column "LastExamPercent" from the "student_info" table for the record where the RollNo = 2.

Note: Delete statement removes one or more columns from one or more rows of a Cassandra table or removes entire rows if no columns are specified.

Act:

```
DELETE LastExamPercent FROM student_info WHERE RollNo=2;
```

Outcome:

```
cqlsh:students> DELETE LastExamPercent FROM student_info where RollNo=2;
cqlsh:students> select * from student_info where rollno = 2;

 rollno | dateofjoining                           | lastexampercent | studname
--------+-----------------------------------------+-----------------+-------------
      2 | 2013-02-27 00:00:00India Standard Time |            null | David Sheen

(1 rows)

cqlsh:students>
```

Objective: To delete a row (where RollNo = 2) from the table "student_info".

Act:

```
DELETE FROM student_info WHERE RollNo=2;
```

Outcome:

```
cqlsh:students> DELETE  FROM student_info where RollNo=2;
cqlsh:students> select * from student_info where rollno=2;

(0 rows)

cqlsh:students>
```

Objective: To create a table "project _details" with primary key as (project_id, project_name).

Act:

```
CREATE TABLE project_details (
        project_id int,
        project_name text,
        stud_name text,
        rating double,
        duration int,
        PRIMARY KEY (project_id, project_name));
```

Outcome:

```
cqlsh:students> CREATE TABLE project_details (
           ...   project_id int,
           ...   project_name text,
           ...   stud_name text,
           ...   rating double,
           ...   duration int,
           ...   PRIMARY KEY (project_id, project_name));
cqlsh:students>
```

Objective: To insert data into the column family "project_details".

Act:

```
BEGIN BATCH
INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
VALUES (1,'MS data migration','David Sheen',3.5,720)
INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
VALUES (1,'MS Data Warehouse','David Sheen',3.9,1440)
INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
VALUES (2,'SAP Reporting','Stephen Fox',4.2,3000)
INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
VALUES (2,'SAP BI DW','Stephen Fox',4,9000)
APPLY BATCH;
```

Outcome:

```
cqlsh:students> BEGIN BATCH
           ...   INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
           ... VALUES (1,'MS data Migration','David Sheen',3.5,720)
           ...   INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
           ... VALUES (1,'MS Data Warehouse','David Sheen',3.9,1440)
           ...   INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
           ... VALUES (2,'SAP Reporting','Stephen Fox',4.2,3000)
           ...   INSERT INTO project_details (project_id,project_name,stud_name,rating,duration)
           ... VALUES (2,'SAP BI DW','Stephen Fox',4,9000)
           ... APPLY BATCH;
```

Objective: To view all the rows of the "project_details" table.

Act:

```
SELECT *
    FROM project_details;
```

Outcome:

```
cqlsh:students> select * from project_details;

 project_id | project_name       | duration | rating | stud_name
------------+--------------------+----------+--------+-------------
          1 | MS Data Warehouse |     1440 |    3.9 | David Sheen
          1 | MS data Migration |      720 |    3.5 | David Sheen
          2 |          SAP BI DW |     9000 |      4 | Stephen Fox
          2 |      SAP Reporting |     3000 |    4.2 | Stephen Fox

(4 rows)
```

Objective: To view row/record from the "project_details" table wherein the project_id=1.

Act:

```
SELECT *
    FROM project_details
        WHERE project_id=1;
```

Outcome:

```
cqlsh:students> Select * from project_details where project_id=1;

 project_id | project_name       | duration | rating | stud_name
------------+--------------------+----------+--------+-------------
          1 | MS Data Warehouse |     1440 |    3.9 | David Sheen
          1 | MS data Migration |      720 |    3.5 | David Sheen

(2 rows)
```

Objective: To use "allow filtering" with the Select statement.

Note: When one attempts a potentially expensive query that might involve searching a range of rows, a prompt such as the one shown below appears:

> Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING.

Act:

```
SELECT *
    FROM project_details
        WHERE project_name='MS Data Warehouse' ALLOW FILTERING;
```

Outcome:

```
cqlsh:students> Select * from project_details where project_name='MS Data Warehouse' allow filtering;

 project_id | project_name       | duration | rating | stud_name
------------+--------------------+----------+--------+-------------
          1 | MS Data Warehouse |     1440 |    3.9 | David Sheen

(1 rows)
```

Objective: To sort or order the rows/records of the "project_details" column in ascending order of project_name.

Act:

```
SELECT *
    FROM project_details
        WHERE project_id IN (1,2)
            ORDER BY project_name DESC;
```

Outcome:

```
cqlsh:students> SELECT * FROM project_details WHERE project_id IN (1,2) ORDER BY project_name DESC;

 project_id | project_name       | duration | rating | stud_name
------------+--------------------+----------+--------+-------------
          2 |      SAP Reporting |     3000 |    4.2 | Stephen Fox
          2 |         SAP BI DW  |     9000 |      4 | Stephen Fox
          1 | MS data Migration  |      720 |    3.5 | David Sheen
          1 | MS Data Warehouse  |     1440 |    3.9 | David Sheen

(4 rows)
```

Note: By default sorting or ordering is done in ascending order. The user can specify the order by using the keyword "ASC" for ascending or "DESC" for descending.

ORDER BY clause can select a single column only. This column is the second column of the compound primary key. This applies even when the compound primary key has more than two columns.

When specifying the ORDER BY clause, use only the column name and not the column alias.

7.7 COLLECTIONS

7.7.1 Set Collection

A column of type set consists of unordered unique values. However, when the column is queried, it returns the values in sorted order. For example, for text values, it sorts in alphabetical order.

PICTURE THIS...

You are required to store details about users of service "xyz". The details of the user include: User_ID, User_Name, User_Contact_Nos, User_Email_Ids. A user may have n number of Contact Nos and also may have n number of Email IDs. How do we accomplish this task in RDBMS?

We would create a table, let us say "Users", to store details such as "User_ID", "User_Name" and another table "UsersContactDetails". The relationship between "UsersContactDetails" and "Users" is many-to-one. Likewise, we would create a table "UsersEmailIDs" and establish a many-to-one relationship between "UserEmailIDs" and the "Users" table.

However, the multiple Contact Nos and multiple Email IDs problem can be solved by defining a column as a collection. The usage of collection types for columns is not only convenient but intuitive as well.

CQL makes use of the following collection types:

- Set
- List
- Map

When to use collection?
Use collection when it is required to store or denormalize a small amount of data.

What is the limit on the values of items in a collection?
The values of items in a collection are limited to 64K.

Where to use collections?
Collections can be used when you need to store the following:

1. Phone numbers of users.
2. Email ids of users.

When should one refrain from using a collection?
One should refrain from using a collection when the data has unbound growth potential such as all the messages posted by a user or all the event data as captured by a sensor. When faced with such a situation, use a table with compound primary key with data being held in clustering columns.

7.7.2 List Collection

When the order of elements matter, one should go for a list collection. For example, when you store the preferences of places to visit by a user, you would like to respect his preferences and retrieve the values in the order in which he has entered rather than in sorted order. A list also allows one to store the same value multiple times.

7.7.3 Map Collection

As the name implies, a map is used to map one thing to another. A map is a pair of typed values. It is used to store timestamp related information. Each element of the map is stored as a Cassandra column. Each element can be individually queried, modified, and deleted.

Objective: To alter the schema for the table "student_info" to add a column "hobbies".

Act:

```
ALTER TABLE student_info ADD hobbies set<text>;
```

Outcome:

```
cqlsh:students> ALTER TABLE student_info ADD hobbies set<text>;
```

Confirm the structure of the table after the change has been made:

```
cqlsh:students> describe table student_info;

CREATE TABLE student_info (
  rollno int,
  dateofjoining timestamp,
  hobbies set<text>,
  lastexampercent double,
  studname text,
  PRIMARY KEY (rollno)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

CREATE INDEX student_info_lastexampercent_idx ON student_info (lastexampercent);

CREATE INDEX student_info_studname_idx ON student_info (studname);
```

Objective: To alter the schema of the table "student_info" to add a list column "language".

Act:

```
ALTER TABLE student_info ADD language list<text>;
```

Outcome:

```
cqlsh:students> ALTER TABLE student_info ADD language list<text>;
cqlsh:students>
```

Confirm the structure of the table after the change has been made:

```
cqlsh:students> describe table student_info;

CREATE TABLE student_info (
  rollno int,
  dateofjoining timestamp,
  hobbies set<text>,
  language list<text>,
  lastexampercent double,
  studname text,
  PRIMARY KEY (rollno)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

CREATE INDEX student_info_lastexampercent_idx ON student_info (lastexampercent);

CREATE INDEX student_info_studname_idx ON student_info (studname);

cqlsh:students>
```

Objective: To update the table "student_info" to provide the values for "hobbies" for the student with Rollno =1.

Act:

```
UPDATE student_info
    SET hobbies = hobbies + {'Chess, Table Tennis'}
        WHERE RollNo=1;
```

Outcome:

```
cqlsh:students> UPDATE student_info
           ...    SET hobbies = hobbies + {'Chess, Table Tennis'} WHERE RollNo=1;
```

To confirm the values in the hobbies column, use the below command:

```
SELECT *
    FROM student_info
        WHERE RollNo=1;
```

```
cqlsh:students> select * from student_info where RollNo=1;

 rollno | dateofjoining                           | hobbies                | language | lastexampercent | studname
--------+-----------------------------------------+------------------------+----------+-----------------+---------------
      1 | 2012-03-29 00:00:00India Standard Time | {'Chess, Table Tennis'} |     null |            69.6 | Michael Storm

(1 rows)
```

Likewise update a few more records:

```
cqlsh:students> UPDATE student_info
           ...    SET hobbies = hobbies + {'Chess, Badminton'} WHERE RollNo=3;
cqlsh:students>
cqlsh:students> UPDATE student_info
           ...    SET hobbies = hobbies + {'Lawn Tennis, Table Tennis, Golf'} WHERE RollNo=4;
cqlsh:students>
```

Records after the updation:

```
cqlsh:students> select * from student_info;

 rollno | dateofjoining                              | hobbies                                | language | lastexampercent | studna
--------+--------------------------------------------+----------------------------------------+----------+-----------------+-------
      1 | 2012-03-29 00:00:00India Standard Time     |                  {'Chess, Table Tennis'} |     null |            69.6 |   Mich
      4 | 2012-05-11 00:00:00India Standard Time     | {'Lawn Tennis, Table Tennis, Golf'}     |     null |            73.4 |      I
      3 | 2014-04-12 00:00:00India Standard Time     |                  {'Chess, Badminton'}   |     null |            81.7 | David

(3 rows)
```

Objective: To update values in the list column, "language" of the table "student_info".

Act:

```
UPDATE student_info
    SET language = language + ['Hindi, English']
        WHERE RollNo=1;
```

Outcome:

```
cqlsh:students> UPDATE student_info
           ...    SET language = language + ['Hindi, English'] WHERE RollNo=1;
cqlsh:students>
```

Likewise update the remaining records.

```
cqlsh:students> UPDATE student_info
           ...    SET language = language + ['Hindi,English,French'] WHERE RollNo=3;
cqlsh:students>
cqlsh:students> UPDATE student_info
           ...    SET language = language + ['Hindi, English'] WHERE RollNo=4;
cqlsh:students>
```

To view the updates to the records, use the below statement:

```
cqlsh:students> select rollNo, studname, hobbies, language from student_info;

 rollno | studname        | hobbies                                | language
--------+-----------------+----------------------------------------+--------------------------
      1 |   Michael Storm |                  {'Chess, Table Tennis'} |       ['Hindi, English']
      4 |      Ian String | {'Lawn Tennis, Table Tennis, Golf'}     |       ['Hindi, English']
      3 | David Flemming  |                  {'Chess, Badminton'}   | ['Hindi,English,French']

(3 rows)
```

7.7.4 More Practice on Collections (SET and LIST)

Objective: To create a table "users" with an "emails" column. The type of this column "emails" is "set".

Act:

```
CREATE TABLE users (
        user_id text PRIMARY KEY,
        first_name text,
        last_name text,
        emails set<text>
);
```

Outcome:

```
cqlsh:students> CREATE TABLE users (
            ...   user_id text PRIMARY KEY,
            ...   first_name text,
            ...   last_name text,
            ...   emails set<text>
            ... );
cqlsh:students>
```

Objective: To insert values into the "emails" column of the "users" table.

Note: Set values must be unique.

Act:

```
INSERT INTO users
        (user_id, first_name, last_name, emails)
                VALUES('AB', 'Albert', 'Baggins', {'a@baggins.com', 'baggins@gmail.com'});
```

Outcome:

```
cqlsh:students> INSERT INTO users (user_id, first_name, last_name, emails)
            ...   VALUES('AB', 'Albert', 'Baggins', {'a@baggins.com', 'baggins@gmail.com'});
cqlsh:students>
```

Objective: Add an element to a set using the UPDATE command and the addition (+) operator.

Act:

```
UPDATE users
        SET emails = emails + {'ab@friendsofmordor.org'}
                WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> UPDATE users
            ...   SET emails = emails + {'ab@friendsofmordor.org'} WHERE user_id = 'AB';
cqlsh:students>
```

Objective: To retrieve email addresses for Albert from the set.

Act:

```
SELECT user_id, emails
    FROM users
        WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> SELECT user_id, emails FROM users WHERE user_id = 'AB';

 user_id | emails
---------+-----------------------------------------------------------------------
      AB | {'a@baggins.com', 'ab@friendsofmordor.org', 'baggins@gmail.com'}

(1 rows)
```

Objective: To remove an element from a set using the subtraction (–) operator.

Act:

```
UPDATE users
    SET emails = emails - {'ab@friendsofmordor.org'}
        WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> UPDATE users
           ...   SET emails = emails - {'ab@friendsofmordor.org'} WHERE user_id = 'AB';
```

To view the records from the "users" table:

```
cqlsh:students> select * from users;

 user_id | emails                                    | first_name | last_name
---------+-------------------------------------------+------------+-----------
      AB | {'a@baggins.com', 'baggins@gmail.com'} |     Albert |   Baggins

(1 rows)
```

Objective: To remove all elements from a set by using the UPDATE or DELETE statement.

Act:

```
UPDATE users
    SET emails = {}
        WHERE user_id = 'AB';
cqlsh:students> UPDATE users SET emails = {} WHERE user_id = 'AB';
```

Outcome: The above command removes the emails column. Here is the confirmation:

```
cqlsh:students> select * from users;

 user_id | emails | first_name | last_name
---------+--------+------------+-----------
      AB |   null |     Albert |   Baggins

(1 rows)
```

OR

```
DELETE emails
        FROM users
                WHERE user_id = 'AB';
```

```
cqlsh:students> DELETE emails FROM users WHERE user_id = 'AB';
```

The above command removes the emails column. Here is the confirmation:

```
cqlsh:students> select * from users;

 user_id | emails | first_name | last_name
---------+--------+------------+-----------
      AB |   null |     Albert |   Baggins

(1 rows)
```

Objective: To alter the "users" table to add a column, "top_places" of type list.

Act:

```
ALTER TABLE users ADD top_places list<text>;
```

```
cqlsh:students> ALTER TABLE users ADD top_places list<text>;
```

Outcome:

The above command alters the structure of the table, "users". Here is the confirmation.

```
cqlsh:students> describe table users;

CREATE TABLE users (
  user_id text,
  emails set<text>,
  first_name text,
  last_name text,
  top_places list<text>,
  PRIMARY KEY (user_id)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

Objective: To update the list column "top_places" in the "users" table for user_id = 'AB'.

Act:

```
UPDATE users
    SET top_places = [ 'Lonavla', 'Khandala' ]
        WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users
           ...   SET top_places = [ 'Lonavla', 'Khandala' ] WHERE user_id = 'AB';
cqlsh:students>
```

Outcome:

```
cqlsh:students> select * from users where user_id = 'AB';

 user_id | emails | first_name | last_name | top_places
---------+--------+------------+-----------+-------------------------
      AB |   null |     Albert |   Baggins | ['Lonavla', 'Khandala']

(1 rows)
```

Objective: Prepend an element to the list by enclosing it in square brackets and using the addition (+) operator.

Act:

```
UPDATE users
    SET top_places = [ 'Mahabaleshwar' ] + top_places
        WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users
           ...   SET top_places = [ 'Mahabaleshwar' ] + top_places WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> select * from users;

 user_id | emails | first_name | last_name | top_places
---------+--------+------------+-----------+-----------------------------------------
      AB |   null |     Albert |   Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala']

(1 rows)
```

Objective: To append an element to the list by switching the order of the new element data and the list name in the update command.

Act:

```
UPDATE users
    SET top_places = top_places + [ 'Tapola' ]
        WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users
           ...   SET top_places = top_places + [ 'Tapola' ] WHERE user_id = 'AB';
cqlsh:students>
```

Outcome:

```
cqlsh:students> select * from users;

 user_id | emails | first_name | last_name | top_places
---------+--------+------------+-----------+-----------------------------------------------------
      AB |   null |     Albert |   Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']

(1 rows)
```

Objective: To query the database for a list of top places.

Act:

```
SELECT user_id, top_places
        FROM users
                WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> SELECT user_id, top_places FROM users WHERE user_id = 'AB';

 user_id | top_places
---------+-----------------------------------------------------
      AB | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']

(1 rows)
```

Objective: To remove an element from a list using the DELETE command and the list index position in square brackets.

The record as it exists prior to deletion is

```
cqlsh:students> SELECT user_id, top_places FROM users WHERE user_id = 'AB';

 user_id | top_places
---------+-----------------------------------------------------
      AB | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']

(1 rows)
```

Act:

```
DELETE top_places[3]
        FROM users
                WHERE user_id = 'AB';
```

```
cqlsh:students> DELETE top_places[3] FROM users WHERE user_id = 'AB';
```

Outcome: The status after deletion is

```
cqlsh:students> select * from users;

 user_id | emails | first_name | last_name | top_places
---------+--------+------------+-----------+---------------------------------------
      AB |   null |     Albert |   Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala']

(1 rows)
```

7.7.5 Using Map: Key, Value Pair

Objective: To alter the "users" table to add a map column "todo".

Act:

```
ALTER TABLE users
        ADD todo map<timestamp, text>;
```

```
cqlsh:students> ALTER TABLE users ADD todo map<timestamp, text>;
```

Outcome:

```
cqlsh:students> describe table users;

CREATE TABLE users (
  user_id text,
  emails set<text>,
  first_name text,
  last_name text,
  todo map<timestamp, text>,
  top_places list<text>,
  PRIMARY KEY (user_id)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

cqlsh:students>
```

Objective: To update the record for user (user_id = 'AB') in the "users" table.

Act: The record from user_id = 'AB' as it exists in the "users" table is

```
cqlsh:students> select * from users where user_id='AB';

 user_id | emails | first_name | last_name | todo | top_places
---------+--------+------------+-----------+------+-----------------------------------------
      AB |   null |     Albert |   Baggins | null | ['Mahabaleshwar', 'Lonavla', 'Khandala']

(1 rows)
```

```
cqlsh:students> UPDATE users
            ...   SET todo =
            ...   {'2014-9-24':'Cassandra Session',
            ...    '2014-10-2 12:00' :'MongoDB Session'}
            ...   WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> select user_id, todo from users where user_id='AB';

r_id | todo
-----+-----------------------------------------------------------------------------------------------------------------------------
 AB | {'2014-09-24 00:00:00India Standard Time': 'Cassandra Session', '2014-10-02 12:00:00India Standard Time': 'MongoDB Se
```

Objective: To delete an element from the map using the DELETE command and enclosing the timestamp of the element in square brackets.

Act:

```
DELETE todo['2014-9-24']
        FROM users
                WHERE user_id = 'AB';
```

```
cqlsh:students> DELETE todo['2014-9-24'] FROM users WHERE user_id = 'AB';
```

Outcome:

```
cqlsh:students> select user_id, todo from users where user_id='AB';

 user_id | todo
---------+-----------------------------------------------------------------
      AB | {'2014-10-02 12:00:00India Standard Time': 'MongoDB Session'}
(1 rows)
```

7.8 USING A COUNTER

A counter is a special column that is changed in increments. For example, we may need a counter column to count the number of times a particular book is issued from the library by the student.

Step 1:

```
CREATE TABLE library_book (
        counter_value counter,
        book_name varchar,
        stud_name varchar,
        PRIMARY KEY (book_name, stud_name)
    );
```

```
cqlsh:students> CREATE TABLE library_book
           ...   (     counter_value counter,
           ...         book_name varchar,
           ...         stud_name varchar,
           ...    PRIMARY KEY (book_name, stud_name)
           ... );
```

Step 2: Load data into the counter column.

```
UPDATE library_book
      SET counter_value = counter_value + 1
            WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

```
cqlsh:students> UPDATE library_book
           ...  SET counter_value = counter_value + 1
           ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

Step 3: Take a look at the counter value.

```
SELECT *
        FROM library_book;
```

Output is:

```
cqlsh:students> select * from library_book;

 book_name                          | stud_name | counter_value
------------------------------------+-----------+---------------
 Fundamentals of Business Analytics |      jeet |             1

(1 rows)
```

Step 4: Let us increase the value of the counter.

```
UPDATE library_book
        SET counter_value = counter_value + 1
                WHERE book_name='Fundamentals of Business Analytics' AND stud_name='shaan';
```

```
cqlsh:students> UPDATE library_book
           ...  SET counter_value = counter_value + 1
           ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='shaan';
```

Step 5: Again, take a look at the counter value.

```
cqlsh:students> select * from library_book;

 book_name                          | stud_name | counter_value
------------------------------------+-----------+---------------
 Fundamentals of Business Analytics |      jeet |             1
 Fundamentals of Business Analytics |     shaan |             1

(2 rows)
```

Step 6: Update another record for Stud_name "Jeet".

```
UPDATE library_book
        SET counter_value = counter_value + 1
                WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

```
cqlsh:students> UPDATE library_book
           ...  SET counter_value = counter_value + 1
           ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

Step 7: Let us take a look at the counter value, one last time.

```
cqlsh:students> select * from library_book;

 book_name                          | stud_name | counter_value
------------------------------------+-----------+---------------
 Fundamentals of Business Analytics |      jeet |             2
 Fundamentals of Business Analytics |     shaan |             1

(2 rows)
```

7.9 TIME TO LIVE (TTL)

Data in a column, other than a counter column, can have an optional expiration period called TTL (time to live). The client request may specify a TTL value for the data. The TTL is specified in seconds.

```
CREATE TABLE userlogin(
        userid int primary key, password text
);
```

```
cqlsh:students> CREATE TABLE  userlogin(
           ... userid int primary key, password text
           ... );
```

INSERT INTO userlogin (userid, password)
VALUES (1,'infy') USING TTL 30;

```
cqlsh:students> INSERT INTO userlogin (userid, password)
           ... VALUES (1,'infy') USING TTL 30;
```

SELECT TTL (password)
FROM userlogin
WHERE userid=1;

```
cqlsh:students> SELECT TTL (password)
           ... FROM userlogin
           ... WHERE userid=1;

 ttl(password)
---------------
            18

(1 rows)
```

7.10 ALTER COMMANDS

Let us look at a few Alter commands to bring about changes to the structure of the table/column family.

1. Create a table "sample" with columns "sample_id" and "sample_name".

CREATE TABLE sample(
sample_id text,
sample_name text,
PRIMARY KEY (sample_id)
);

```
cqlsh:students> Create table sample (sample_id text, sample_name text, primary key (sample_id));
cqlsh:students>
```

2. Insert a record into the table "sample".

INSERT INTO sample(
sample_id, sample_name)
VALUES ('S101', 'Big Data');

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 'S101', 'Big Data' );
```

3. View the records of the table "sample".

SELECT *
FROM sample;

```
cqlsh:students> select * from sample;

 sample_id | sample_name
-----------+-------------
      S101 |    Big Data

(1 rows)
```

7.10.1 Alter Table to Change the Data Type of a Column

1. Alter the schema of the table "sample". Change the data type of the column "sample_id" to integer from text.

ALTER TABLE sample
ALTER sample_id TYPE int;

```
cqlsh:students> alter table sample alter sample_id type int;
```

2. After the data type of the column "sample_id" is changed from text to integer, try inserting a record as follows and observe the error message:

INSERT INTO sample(sample_id, sample_name)
VALUES('S102', 'Big Data');

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 'S102', 'Big Data' );
Bad Request: Invalid STRING constant (S102) for sample_id of type int
cqlsh:students>
```

3. Try inserting a record as given below into the table "sample".

INSERT INTO sample(sample_id, sample_name)
VALUES(102, 'Big Data');

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 102, 'Big Data' );
cqlsh:students> select * from sample;

 sample_id  | sample_name
------------+-------------
 1395732529 |    Big Data
        102 |    Big Data

(2 rows)
```

4. Alter the data type of the "sample_id" column to varchar from integer.

ALTER TABLE sample
ALTER sample_id TYPE varchar;

```
cqlsh:students> alter table sample alter sample_id type varchar;
```

5. Check the records after the data type of "sample_id" has been changed to varchar from integer.

```
cqlsh:students> select * from sample;

 sample_id      | sample_name
----------------+-------------
          S101 |    Big Data
 \x00\x00\x00f |    Big Data

(2 rows)
```

7.10.2 Alter Table to Delete a Column

1. Drop the column "sample_id" from the table "sample".

ALTER TABLE sample
DROP sample_id;

```
cqlsh:students> alter table sample drop sample_id;
Bad Request: Cannot drop PRIMARY KEY part sample_id
```

Note: The request to drop the "sample_id" column from the table "sample" does not succeed as it is the primary key column.

2. Drop the column "sample_name" from the table "sample".

ALTER TABLE sample
DROP sample_name;

```
cqlsh:students> alter table sample drop sample_name;
```

Note: the above request to drop the column "sample_name" from table "sample" succeeds.

7.10.3 Drop a Table

1. Drop the column family/table "sample".

DROP columnfamily sample;

```
cqlsh:students> drop columnfamily sample;
```

The above request succeeds. The table/column family no longer exists in the keyspace.

2. Confirm the non-existence of the table "sample" in the keyspace by giving the following command:

```
cqlsh:students> describe table sample;

Column family 'sample' not found
```

7.10.4 Drop a Database

1. Drop the keyspace "students".

DROP keyspace students;

```
cqlsh:students> drop keyspace students;
```

2. Confirm the non-existence of the keyspace "students" by issuing the following command:

```
cqlsh:students> describe keyspace students;

Keyspace 'students' not found.
```

7.11 IMPORT AND EXPORT

7.11.1 Export to CSV

Objective: Export the contents of the table/column family "elearninglists" present in the "students" database to a CSV file (d:\elearninglists.csv).

Act:

Step 1: Check the records of the table "elearninglists" present in the "students" database.

SELECT *
FROM elearninglists;

```
cqlsh:students> select * from elearninglists;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra
 101 |            2 |      1002 |       Seema |   NoSQL MongoDB
 101 |            3 |      1003 |       Seema |    Hadoop Sqoop
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume

(4 rows)
```

Step 2: Execute the below command at the cqlsh prompt:

COPY elearninglists (id, course_order, course_id, courseowner, title) TO 'd:\elearninglists.csv';

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) to 'd:\elearninglists.csv';
4 rows exported in 0.000 seconds.
cqlsh:students>
```

Step 3: Check the existence of the "elearninglists.csv" file in "D:\". Given below is the content of the "d:\elearninglists.csv" file.

	A	B	C	D	E
1	101	1	1001	Subhashini	NoSQL Cassandra
2	101	2	1002	Seema	NoSQL MongoDB
3	101	3	1003	Seema	Hadoop Sqoop
4	101	4	1004	Subhashini	Hadoop Flume

7.11.2 Import from CSV

Objective: To import data from "D:\elearninglists.csv" into the table "elearninglists" present in the "students" database.

Step 1: Check for the table "elearninglists" in the "students" database. If the table is already present, truncate the table. This will remove all records from the table but retain the structure of the table. In our case, the table "elearninglists" is already present in the "students" database. Let us take a look at the records of the "elearninglists" before we run the truncate command on it.

```
cqlsh:students> select * from elearninglists;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra
 101 |            2 |      1002 |       Seema |   NoSQL MongoDB
 101 |            3 |      1003 |       Seema |    Hadoop Sqoop
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume

(4 rows)
```

Truncate the table using the below command:

TRUNCATE elearninglists;

```
cqlsh:students> Truncate  elearninglists;
cqlsh:students>
```

Note: No record is present in the table "elearninglists". The structure/schema is however preserved. We confirm it by executing the below command at the cqlsh prompt.

```
cqlsh:students> select * from elearninglists;

(0 rows)

cqlsh:students>
```

Step 2: Check for the content of the "D:\elearninglists.csv" file.

	A	B	C	D	E
1	101	1	1001	Subhashini	NoSQL Cassandra
2	101	2	1002	Seema	NoSQL MongoDB
3	101	3	1003	Seema	Hadoop Sqoop
4	101	4	1004	Subhashini	Hadoop Flume

Note: The content in the CSV agrees with the structure of the table "elearninglists" in the "students" database. The structure should be such that the content from the CSV can be housed within it without any issues.

Step 3: Execute the below command to import data from "d:\elearninglists.csv" into the table "elearninglists" in the database "students".

```
COPY elearninglists (id, course_order, course_id, courseowner, title) FROM 'd:\elearninglists.csv';
```

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) from 'd:\elearninglists.csv';
4 rows imported in 0.031 seconds.
cqlsh:students>
```

Step 4: Confirm that records have been imported into the table.

```
SELECT *
      FROM elearninglists;
```

```
cqlsh:students> select * from elearninglists;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra
 101 |            2 |      1002 |       Seema |   NoSQL MongoDB
 101 |            3 |      1003 |       Seema |    Hadoop Sqoop
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume

(4 rows)

cqlsh:students>
```

7.11.3 Import from STDIN

Objective: To import data into an existing table "persons" present in the "students" database. The data is to be provided by the user using the standard input device.

Step 1: Ensure that the table "persons" exists in the database "students".

```
DESCRIBE TABLE persons;
```

```
cqlsh:students> describe table persons;

CREATE TABLE persons (
  id int,
  fname text,
  lname text,
  PRIMARY KEY (id)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

cqlsh:students>
```

Step 2:

```
COPY persons (id, fname, lname) FROM STDIN;
```

```
cqlsh:students> COPY persons (id, fname, lname) FROM STDIN;
[Use \. on a line by itself to end input]
[copy] 1,"Samuel","Jones"
[copy] 2,"Virat","Kumar"
[copy] 3,"Andrew","Simon"
[copy] 4,"Raul","A Simpson"
[copy] \.

4 rows imported in 1 minute and 24.336 seconds.
cqlsh:students>
```

Step 3: Confirm that the records from the standard input device are loaded into the "persons" table existing in the "students" database.

```
SELECT *
      FROM persons;
```

```
cqlsh:students> select * from persons;

 id | fname  | lname
----+--------+-----------
  1 | Samuel |     Jones
  2 |  Virat |     Kumar
  4 |   Raul | A Simpson
  3 | Andrew |     Simon

(4 rows)

cqlsh:students>
```

7.11.4 Export to STDOUT

Objective: Export the contents of the table/column family "elearninglists" present in the "students" database to the standard output device (STDOUT).

Act:

Step 1: Check the records of the table "elearninglists" present in the "students" database.

```
SELECT *
      FROM elearninglists;
```

```
cqlsh:students> select * from elearninglists;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra
 101 |            2 |      1002 |       Seema |   NoSQL MongoDB
 101 |            3 |      1003 |       Seema |    Hadoop Sqoop
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume

(4 rows)
```

Step 2: Execute the below command at the cqlsh prompt.

COPY elearninglists (id, course_order, course_id, courseowner, title) TO STDOUT;

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) to STDOUT;
101,1,1001,Subhashini,NoSQL Cassandra
101,2,1002,Seema,NoSQL MongoDB
101,3,1003,Seema,Hadoop Sqoop
101,4,1004,Subhashini,Hadoop Flume
4 rows exported in 0.031 seconds.
cqlsh:students>
```

7.12 QUERYING SYSTEM TABLES

There are quite a few system tables such as schema_keyspaces, schema_columnfamilies, schema_columns, local, peers, etc. Let us look at what each of these system tables store in them.

SELECT *

FROM system.schema_keyspaces;

```
cqlsh:system> describe table system.schema_keyspaces;

CREATE TABLE schema_keyspaces (
  keyspace_name text,
  durable_writes boolean,
  strategy_class text,
  strategy_options text,
  PRIMARY KEY (keyspace_name)
) WITH COMPACT STORAGE AND
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='keyspace definitions' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=8640 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

SELECT *

FROM system.schema_columnfamilies;

```
CREATE TABLE schema_columnfamilies (
  keyspace_name text,
  columnfamily_name text,
  bloom_filter_fp_chance double,
  caching text,
  column_aliases text,
  comment text,
  compaction_strategy_class text,
  compaction_strategy_options text,
  comparator text,
  compression_parameters text,
  default_time_to_live int,
  default_validator text,
```

```
  dropped_columns map<text, bigint>,
  gc_grace_seconds int,
  index_interval int,
  key_aliases text,
  key_validator text,
  local_read_repair_chance double,
  max_compaction_threshold int,
  memtable_flush_period_in_ms int,
  min_compaction_threshold int,
  populate_io_cache_on_flush boolean,
  read_repair_chance double,
  replicate_on_write boolean,
  speculative_retry text,
  subcomparator text,
  type text,
  value_alias text,
  PRIMARY KEY (keyspace_name, columnfamily_name)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='ColumnFamily definitions' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=8640 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

SELECT *

FROM system.schema_columns;

```
cqlsh:system> describe table system.schema_columns;

CREATE TABLE schema_columns (
  keyspace_name text,
  columnfamily_name text,
  column_name text,
  component_index int,
  index_name text,
  index_options text,
  index_type text,
  type text,
  validator text,
  PRIMARY KEY (keyspace_name, columnfamily_name, column_name)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='ColumnFamily column attributes' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=8640 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

```
SELECT *
    FROM system.local;
CREATE TABLE local (
  key text,
  bootstrapped text,
  cluster_name text,
  cql_version text,
  data_center text,
  gossip_generation int,
  host_id uuid,
  native_protocol_version text,
  partitioner text,
  rack text,
  release_version text,
  schema_version uuid,
  thrift_version text,
  tokens set<text>,
  truncated_at map<uuid, blob>,
  PRIMARY KEY (key)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='information about the local node' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=0 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

SELECT *
    FROM system.peers;

CREATE TABLE peers (
  peer inet,
  data_center text,
  host_id uuid,
  preferred_ip inet,
  rack text,
  release_version text,
  rpc_address inet,
  schema_version uuid,
  tokens set<text>,
  PRIMARY KEY (peer)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='known peers in the cluster' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=0 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

7.13 PRACTICE EXAMPLES

Objective: To create table "elearninglist" with columns: id, course_order, course_id, title, courseowner.

Act:

```
CREATE TABLE elearninglists (
        id int,
        course_order int,
        course_id int,
        title text,
        courseowner text,
        PRIMARY KEY (id, course_order )
        );
```

Here, id ==> Partition Key, course_order ==> Clustering Column. The combination of the id and course_order in the elearninglists table uniquely identifies a row in the elearninglists table. You can have more than one row with the same id as long as the rows contain different course_order values.

Outcome:

```
cqlsh:students> CREATE TABLE elearninglists (
           ...   id int,
           ...   course_order int,
           ...   course_id int,
           ...   title text,
           ...   courseowner text,
           ...   PRIMARY KEY  (id, course_order ) );
cqlsh:students>
```

Objective: To insert rows into the table "elearninglists".

Act:

```
INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
VALUES (101, 1, 1001,'NoSQL Cassandra','Subhashini');

INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
VALUES (101, 2, 1002,'NoSQL MongoDB','Seema');

INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
VALUES (101, 3, 1003,'Hadoop Sqoop','Seema');

INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
VALUES (101, 4, 1004,'Hadoop Flume', 'Subhashini');
```

Outcome:

```
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
           ...   VALUES (101,1,1001,'NoSQL Cassandra','Subhashini');
cqlsh:students>
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
           ...   VALUES (101,2,1002,'NoSQL MongoDB','Seema');
cqlsh:students>
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
           ...   VALUES (101,3,1003,'Hadoop Sqoop','Seema',);
Bad Request: line 2:44 no viable alternative at input ')'
cqlsh:students>
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
           ...   VALUES (101, 4,1004,'Hadoop Flume', 'Subhashini');
cqlsh:students>
```

Objective: To query the table "elearninglists".

Act:

```
SELECT *
    FROM elearninglists;
```

Outcome:

```
cqlsh:students> SELECT * FROM elearninglists;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra
 101 |            2 |      1002 |       Seema |   NoSQL MongoDB
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume

(3 rows)
```

Objective: To query the "elearninglist" table on "courseowner" as a filter.

Act:

```
SELECT *
    FROM elearninglists
        WHERE courseowner = 'Seema';
```

Outcome: The query returns an error stating "No indexed columns present in by-columns clause with Equal operator".

```
cqlsh:students> SELECT * FROM elearninglists WHERE courseowner = 'Seema';
Bad Request: No indexed columns present in by-columns clause with Equal operator
cqlsh:students>
```

Solution: Create an index on courseowner column.

```
CREATE INDEX ON
    Elearninglists(courseowner);
```

```
cqlsh:students> CREATE INDEX ON elearninglists(courseowner);
cqlsh:students>
```

Executing the same query now shows the record:

```
cqlsh:students> SELECT * FROM elearninglists WHERE courseowner = 'Seema';

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+---------------
 101 |            2 |      1002 |       Seema | NoSQL MongoDB

(1 rows)
cqlsh:students>
```

Objective: To order all the rows of elearninglists with id = 101 in descending order of "course_order". The maximum number of records to retrieve is 50.

Act:

```
SELECT *
    FROM elearninglists
        WHERE id = 101
            ORDER BY course_order DESC LIMIT 50;
```

Outcome:

```
cqlsh:students> SELECT * FROM elearninglists WHERE id = 101 ORDER BY course_order DESC LIMIT 50;

 id  | course_order | course_id | courseowner | title
-----+--------------+-----------+-------------+-----------------
 101 |            4 |      1004 |  Subhashini |    Hadoop Flume
 101 |            2 |      1002 |       Seema |  NoSQL MongoDB
 101 |            1 |      1001 |  Subhashini | NoSQL Cassandra

(3 rows)
```

REMIND ME

- Apache Cassandra was born at Facebook. After Facebook open sourced the code in 2008, Cassandra became an Apache Incubator project in 2009 and subsequently became a top-level Apache project in 2010.
- Cassandra does NOT compromise on availability. Since it does not have a master-slave architecture, there is no question of single point of failure. This proves beneficial for business critical applications that need to be up and running always and cannot afford to go down ever.
- A replication strategy is employed to determine which nodes to place the data on. Two replication strategies are available:
  - SimpleStrategy.
  - NetworkToplogyStrategy.
- One of the features of Cassandra that has made it immensely popular is its ability to utilize tunable consistency. The database systems can go for either strong consistency or eventual consistency.
- Read consistency means how many replicas must respond before sending out the result to the client application.
- Write consistency means on how many replicas write must succeed before sending out an acknowledgement to the client application.

POINT ME (BOOK)

- ***Cassandra:*** The Definitive Guide By Eben Hewitt, Publisher: O'Reilly Media.

CONNECT ME (INTERNET RESOURCES)

- http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html
- http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf
- http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html
- http://www.datastax.com/docs/1.0/cluster_architecture/about_client_requests
- http://www.datastax.com/docs/datastax_enterprise3.1/solutions/about_pig

TEST ME

A. Unsolved Exercises

1. What is Cassandra?
2. Comment on Cassandra writes.
3. What is your understanding of tunable consistency?
4. What are collections in CQLSH? Where are they used?
5. Explain hinted handoffs.
6. What is Cassandra – cli?
7. Explain Cassandra's data model.
8. Explain the replication strategy in Cassandra.
9. Cassandra adheres to the Availability and Partition tolerant traits as stated by the CAP theorem. Explain.

ASSIGNMENTS FOR HANDS-ON PRACTICE

ASSIGNMENT 1: COLLECTIONS

Objective: To learn about the various collection types: Set, List and Map.
Problem Description: Design a table/column family to support the following requirements.

- Store the basic information about students such as Student Roll No, Student Name, Student Date of Birth, and Student Address.
- Store the subject preferences of each student. There should be a minimum of two subject preferences and a maximum of four. The order of preferences as given by the student should be preserved.
- Store the hobbies of each student. There should be a minimum of two hobbies and a maximum of four. The hobbies as given by the student should be arranged in alphabetical order.

ASSIGNMENT 2: TIME TO LIVE

Objective: To learn about the TTL type (Time To Live).

Problem Description: Design a table/column family to support the following requirements.

Store the login details of the user such as UserID and Password. The information stored should expire in a day's time.

ASSIGNMENT 3: IMPORT FROM CSV

Objective: To learn about the import from CSV to Cassandra table/column family.

Problem Description: Read a public dataset from the site www.kdnuggets.com. If not already in CSV format, first convert to CSV format and then import into a Cassandra table/column family by the name "PublicDataSet" in the "Sample" database.

Confirm the presence of data in the table "PublicDataSet" in the "Sample" database.