UNIT-2

Introduction to Cassandra: Apache Cassandra – An Introduction, Features of Cassandra, CQL Data Types, CQLSH, Keyspaces, CRUD, Collections, Using a Counter, Time to Live, Alter Commands, Import and Export using CSV file.

Q) Explain features of Cassandra.

Apache Cassandra we 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.
It is built on Amazon's dynamo and Google's BigTable.
Cassandra has been immensely used in Twitter, Netflix, Cisco, Adobe, eBay and Rackspace.

Features:

1. **Open source:** Cassandra is an open source NoSQL database.
2. **Distributed:** Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware.
3. **No Single Point of Failure:** It does not have a master-slave architecture, so no question of single point of failure.
4. **Column Oriented:** It is a column oriented database designed to support peer-to-peer symmetric nodes instead of master-slave architecture. The data is stored in **tables** containing **rows** of **columns**.
5. **Availability:** It has adherence to the Availability and Partition Tolerance properties of CAP theorem.
6. **Scalability:** It is highly scalable, high performance distributed database. It distributes and manages gigantic amount of data across commodity servers.
7. **Peer-to-Peer Network:** Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware. A node in Cassandra is structurally identical to any other node.

Q) Explain how data writing takes place in Cassandra.

Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware. A node in Cassandra is structurally identical to any other node.

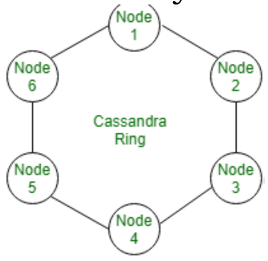


Fig. Sample Cluster

1. **Gossip and Failure Detection:** Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know

about. The gossip process runs every second and exchanges state messages with up to three other nodes in the cluster.
The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster whether a node is alive or offline.
A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

2. **Partitioner:** A partitioner takes a call on how to distribute data on the various nodes in a cluster. A partitioner is a hash function to compute the token of the partition key. The partition key helps to identify a row uniquely.

3. **Replication factor:** It determines number of copies of data(replicas) that will be stored across nodes in a cluster.

4. **Anti-Entropy and Read Repair:** when a client connects to any node in the cluster to read data, the key question is how many nodes will be read before responding to the client is based on the consistency level specified by the client.
   If the consistency is not met, the read operation blocks as few of the nodes ma respond with an out-of-date value.
   For repairing unread data, Cassandra uses an anti-entropy version of the gossip protocol, implies comparing all the replicas of each piece of data and updating each replica to the newest version.

5. **Writes in Cassandra:** Assume a client 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.
   When the number of objects stored in the Memtable reaches a defined threshold value, the contents of Memtable are flushed to the disk in a file called SSTable(Sorted String Table).
   It is possible to have multiple Memtables for a single column family. One of them is current and rest are waiting to be flushed.

6. **Hinted Handoffs:** Cassandra works on the philosophy that it will always be available for writes.
   Assume there are 3 nodes A,B,C in a cluster. Node C is down for some reason. We maintain a replication factor of 2. The client makes a write request to Node A. Node A is the coordinator and serves as proxy between the client and the nods on which the replica is to be places(B & C).

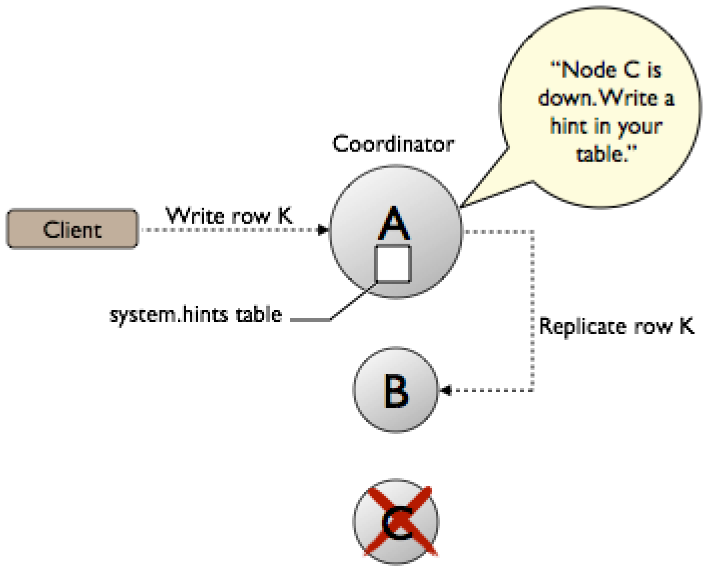
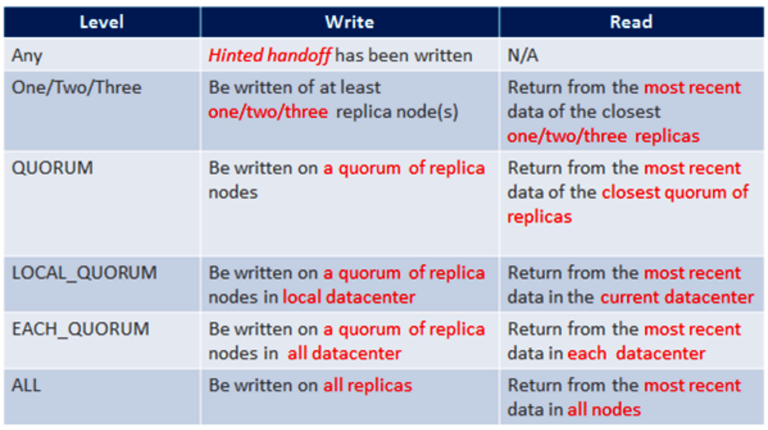


Fig. Depiction of hinted handoffs.
The client writes Row K to Node A. Node A then writes Row K to Node B and stores a hint for Node C. The hint will have the following information:

- Location of the node on which the replica is to be placed
- Version metadata
- 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. **Tunable Consistency:** The consistency level on Cassandra is tunable by the user.
   i. Strong Consistency: If we work with strong consistency it implies that each update propagates to all locations where that piece of data resides.
   ii. Eventual Consistency: It implies that the client is acknowledged with a success as soon as a part of the cluster acknowledges the write.



Level	Write	Read
Any	*Hinted handoff* has been written	N/A
One/Two/Three	Be written of at least one/two/three replica node(s)	Return from the most recent data of the closest one/two/three replicas
QUORUM	Be written on a quorum of replica nodes	Return from the most recent data of the closest quorum of replicas
LOCAL_QUORUM	Be written on a quorum of replica nodes in local datacenter	Return from the most recent data in the current datacenter
EACH_QUORUM	Be written on a quorum of replica nodes in all datacenter	Return from the most recent data in each datacenter
ALL	Be written on all replicas	Return from the most recent data in all nodes

```
quorum=( #replic / 2 ) + 1
```

Fig. Read and Write Consistency levels in Cassandra

This means that considering the default replication factor of three (3) defined for the tables of a keyspace and a consistency level of ALL, one write operation on Cassandra will wait for the data be written and confirmed by all 3 nodes before reply to the client.

Q) What is CQL? List data types in CQL.

The Cassandra Query Language (CQL) offers a model similar to SQL. The data is stored in **tables** containing **rows** of **columns**.

Built-in data types in Cassandra:

Data type	**Description**
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 of 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

Q) Define Keyspaces. Explain various operations on Keyspaces with suitable examples.

- **Keyspace** is a container to hold application data. It is comparable to a relational database.
- It is used to group column families(tables) together.
- Typically, a cluster has one keyspace per application.
- Replication is controlled on a per keyspace basis. So, data that has different replication requirements should reside on different keyspaces.
- When one creates a keyspace, it is required to specify a strategy class.

  SimpleStragegy class: Used when to evaluating Cassandra. Typically used when we work with single data centre.

  NetworkTopologyStrategy class: Used for production usage. Typically used when we work with more than one single data centre.

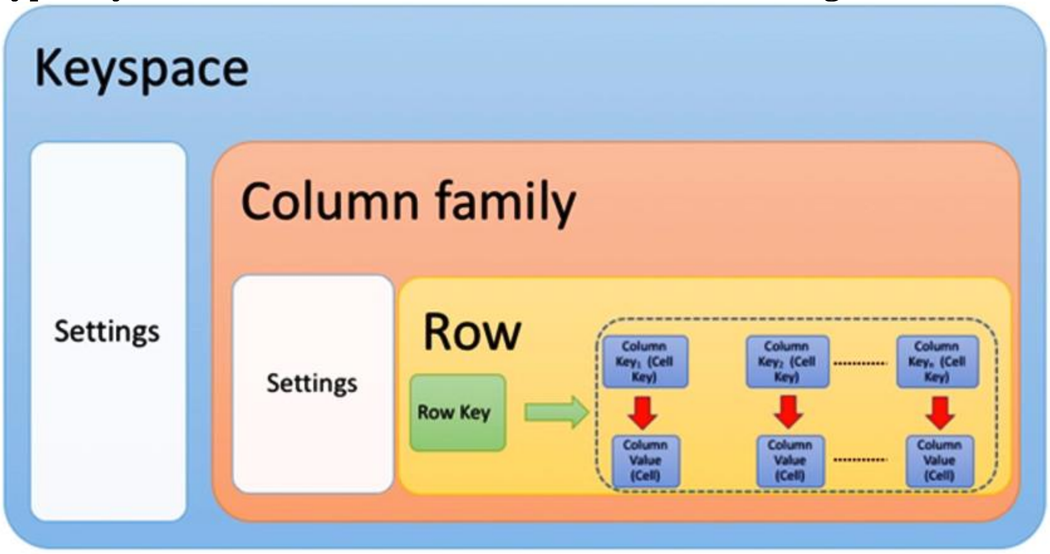


Fig. Representation of Keyspace and Column family.

1. **Create keyspace:** To create a new keyspace.

Syntax: CREATE KEYSPACE <keyspace_name> WITH <properties>
Eg.

i. Create a keyspace by the name "chp"

```
cqlsh> create keyspace chp with replication={'class' : 'SimpleStrategy',
'replication_factor' : 1};
```

ii. To describe all the existing keyspaces.

```
cqlsh> describe keyspaces;
system_schema  system  stud             system_traces
system_auth    chp     system_distributed  stud_b
```

2. **Use:** Use connects the client session to the specified keyspace.

Syntax: use keyspace_name;
Eg.

```
cqlsh> use chp;
cqlsh:chp>
```

3. **Alter keyspace:** To alter an existing keyspace. If keyspace doesn't exists it throws error.

Syntax: ALTER KEYSPACE <keyspace_name> WITH <properties>
Eg.

```
cqlsh:chp> alter keyspace chp with replication={'class' : 'SimpleStrategy',
'replication_factor' : 2};
```

4. **Drop keyspace:** To drop an existing keyspace. If keyspace doesn't exists it throws error.

Syntax: DROP keyspace KeyspaceName ;
Eg.

```
cqlsh:chp> drop keyspace chp;
```

Q) Explain CRUD operations with Suitable examples.

CRUD(Create, Read, Update and Delete) Operations:

1. **Create:** To creating a column family or table in a keyspace
   Syntax:

```
CREATE TABLE tablename(
        column1_name datatype PRIMARY KEY,
        column2_name data type,
        column3_name data type…
        ) ;
```

i. Connect to the keyspace "chp"

```
cqlsh> use chp;
```

ii. Create a table/column family "stud_info"

```
cqlsh:chp> create table stud_info(rno int Primary Key,sname text, doj
timestamp, percent double);
```

iii. Display the structure of the table stud_info

```
cqlsh:chp> describe stud_info;

CREATE TABLE chp.stud_info (
    rno int PRIMARY KEY,
    doj timestamp,
    percent double,
    sname text)
```

iv. Display all the tables in the keyspace chp

```
cqlsh:chp> describe tables;
stud_info
```

2. **Update/Insert:**

- An insert writes one or more columns to a record in Cassandra table automatically. An insert statement does not return an output.
- It is not required to place values in all 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 the disk.
- Internally update and insert operations are equal. However, insert does not support counters but update does.

Syntax for inserting data into a table:

```
INSERT INTO <tablename>
(<column1_name>, <column2_name>....)
VALUES (<value1>, <value2>....)
```

Syntax for updating a value:

```
UPDATE <tablename>
SET <column_name> = <new_value>
<column_name> = <value>....
WHERE <condition>
```

v. Insert data into the table stud_info

```
cqlsh:chp> insert into stud_info(rno,sname,doj,percent) values(1,'chp','2019-03-29',69.9);
cqlsh:chp> insert into stud_info(rno,sname,doj,percent) values(2,'vr','2018-05-15',70.1);
cqlsh:chp> insert into stud_info(rno,sname,doj,percent) values(3,'pnr','2019-03-30',42.3);
```

vi. Add new column branch in the table.

```
cqlsh:chp> alter table stud_info add branch text;
cqlsh:chp> select  * from stud_info;

 rno | branch | doj                             | percent | sname
-----+--------+---------------------------------+---------+-------
   1 |   null | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
   2 |   null | 2018-05-14 18:30:00.000000+0000 |    70.1 |    vr
   3 |   null | 2019-03-29 18:30:00.000000+0000 |    42.3 |   pnr
```

```
(3 rows)
```

vii. Update value into the column branch

```
cqlsh:chp> update stud_info set branch = 'it'  where rno = 1;
cqlsh:chp> update stud_info set branch = 'it' where rno = 3;
cqlsh:chp> update stud_info set branch = 'cse' where rno = 2;
cqlsh:chp> select * from stud_info;

 rno | branch | doj                             | percent | sname
-----+--------+---------------------------------+---------+-------
   1 |     it | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
   2 |    cse | 2018-05-14 18:30:00.000000+0000 |    70.1 |    vr
   3 |     it | 2019-03-29 18:30:00.000000+0000 |    42.3 |   pnr

(3 rows)
```

3. Read: To retrieve or fetch the data from a table.

Syntax:

select <column1_name>,<column2_name> .. from table_name where <condition>;

viii. Read data in the table stud_info.

```
cqlsh:chp> select * from stud_info;

 rno | doj                             | percent | sname
-----+---------------------------------+---------+-------
   1 | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
   2 | 2018-05-14 18:30:00.000000+0000 |    70.1 |    vr
   3 | 2019-03-29 18:30:00.000000+0000 |    42.3 |   pnr
```

ix. Display information of students whose rno are 1 and 2.

```
cqlsh:chp> select * from stud_info where rno in (1,2);

 rno | doj                             | percent | sname
-----+---------------------------------+---------+-------
   1 | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
   2 | 2018-05-14 18:30:00.000000+0000 |    70.1 |    vr

(2 rows)
```

x. Display information of the student whose name is chp.

```
cqlsh:chp> select * from stud_info where sname = 'chp';
```

InvalidRequest: Error from server: code=2200 [Invalid query] message="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**"

NOTE: when we try to run a query with sname in where clause, it lead to error as sname is neither primary key column nor a column in the primary key definition or does not have an index defined on it.

```
cqlsh:chp> create index on stud_info(sname);
```

```
cqlsh:chp> select * from stud_info where sname = 'chp';

 rno | doj                             | percent | sname
-----+---------------------------------+---------+-------
   1 | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
```

xi. Alias the column sname as student name and display student name and percent in the table.

```
cqlsh:chp> select sname as "student name",percent from stud_info;

 student name | percent
--------------+---------
          chp |    69.9
           vr |    70.1
          pnr |    42.3

(3 rows)
```

4. **Delete:** to delete a record or removing a value of a column in a record.

Syntax:
DELETE FROM <tablename> WHERE <condition>;

xii. Delete student record whose rno is 2.

```
cqlsh:chp> delete from stud_info where rno = 2;
cqlsh:chp> select * from stud_info;

 rno | branch | doj                             | percent | sname
-----+--------+---------------------------------+---------+-------
   1 |     it | 2019-03-28 18:30:00.000000+0000 |    69.9 |   chp
   3 |     it | 2019-03-29 18:30:00.000000+0000 |    42.3 |   pnr

(2 rows)
```

xiii. Make percent as null for the student whose rno is 1

```
cqlsh:chp> delete percent from stud_info where rno = 1;

cqlsh:chp> select * from stud_info;

 rno | branch | doj                             | percent | sname
-----+--------+---------------------------------+---------+-------
   1 |     it | 2019-03-28 18:30:00.000000+0000 |    null |   chp
   3 |     it | 2019-03-29 18:30:00.000000+0000 |    42.3 |   pnr
(2 rows)
```

xiv. Drop the column percent from the table

```
cqlsh:chp> alter table stud_info drop percent;
cqlsh:chp> select * from stud_info;

 rno | branch | doj                             | sname
-----+--------+---------------------------------+-------
   1 |     it | 2019-03-28 18:30:00.000000+0000 |   chp
   3 |     it  | 2019-03-29 18:30:00.000000+0000 |    pnr

(2 rows)
```

xv. Drop the table stud_info.

```
cqlsh:chp> drop table stud_info;
cqlsh:chp> select * from stud_info;
InvalidRequest: Error from server: code=2200 [Invalid query] message="unconfigured table stud_info"
```

Q) What is the need of Collections? Explain different collections in Cassandra.

Cassandra provides collection types as a way to group and store data together in a column i.e; to store multiple values in a column like storing multiple mobile number etc. They are used when to store or denormalize a small amount of data.

CQL makes use of the following collection types:

1. **Set:**
   A column of type set consists of unordered unique values.
   When the column is queried, it returns the values in sorted order.
   eg.: for text values, it sorts in alphabetical order

i. Create table users with uid as primary key and a column “emaiils” as set collection.

```
cqlsh:chp> create table users(uid text Primary Key, fname text, lname text, emails set<text>);
```

ii. Inserting data into the table.

```
cqlsh:chp> insert into users (uid,fname,lname,emails) values ('A','ch','praneeth',{'chp@pvpsit.in','praneeth@pvpsit.in'});
```

iii. Reading data from the table.

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

 uid | emails                                   | fname | lname
-----+------------------------------------------+-------+----------
   A | {'chp@pvpsit.in', 'praneeth@pvpsit.in'} |    ch | praneeth
```

iv. Adding new value to existing set column emails whose uid is ‘A’.

```
cqlsh:chp> update users set emails = emails + {'chpraneeth@pvpsit.in'}
where uid = 'A';
```

v. deleting a value from the column emails whose uid is ‘A’.

```
cqlsh:chp> update users set emails = emails - {'chp@pvpsit.in'} where uid =
'A';
```

vi. reading data from the table.

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

```
 uid | emails                                       | fname | lname
-----+----------------------------------------------+-------+----------
   A | {'chpraneeth@pvpsit.in', 'praneeth@pvpsit.in'} |    ch | praneeth
```

vii. making emails column empty whose uid is ‘A’.

```
cqlsh:chp> update users set emails = {} where uid = 'A';
```

viii. reading data from the table.

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

 uid | emails | fname | lname
-----+--------+-------+----------
   A |   null |    ch | praneeth
```

2. **List:**
   List is used when the order of elements matter.
   List allows to store the same value multiple times.
   Eg. When we want to store preferences of places to visit by a user, we would like to follow the preferences and retrieve the values in the same order rather than sorted order.

   i. Add top_places column as list collection.

```
cqlsh:chp> alter table users add top_places list<text>;
```

   ii. Update value of the column top_places whose uid is ‘A’

```
cqlsh:chp> update users set top_places = ['prakasam barrage', 'bhavani island'] where uid = 'A';
```

   iii. Update value of the column top_places whose uid is ‘A’.

```
cqlsh:chp> update users set top_places = ['killa'] + top_places where uid = 'A';
```

   iv. Reading data from the table users.

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

 uid | emails | fname | lname    | top_places
-----+--------+-------+----------+----------------------------------------------
   A |   null |    ch | praneeth | ['killa', 'prakasam barrage', 'bhavani island']
```

   v. Delete a value in the column top_places using index of the values whose uid is ‘A’.

```
cqlsh:chp> delete top_places[1] from users where uid = 'A';
```

   vi. Read data from the table.

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

 uid | emails | fname | lname    | top_places
-----+--------+-------+----------+----------------------------
   A |   null |    ch | praneeth | ['killa', 'bhavani island']

(1 rows)
```

vii. Remove the value killa from the top_places whose uid is ‘A’

```
cqlsh:chp> update users set top_places = top_places - ['killa']  where uid = 'A';
cqlsh:chp> select * from users;

 uid | emails | fname | lname     | top_places
-----+--------+-------+-----------+-------------------
   A |   null |    ch | praneeth | ['bhavani island']
```

3. **Map:**

   A map is a collection of <Key,Value> pairs.
   Each element of the map is stored as a Cassandra column.
   Each element can be individually queried, modified and deleted.
   Eg.

   i. Add a new column “todo_list” as map collection to the table users

```
cqlsh:chp> alter table users add todo_list map<timestamp, text>;
```

   ii. Add values to the column todo_list

```
cqlsh:chp> update users set todo_list = {'2021-10-22':'cassandra','2021-10-28':'hadoop'} where uid = 'A';
```

   iii. Read data from the table.

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

 uid | emails | fname | lname     | todo_list
| top_places
-----+--------+-------+-----------+--------------------------------------------------------------------------------------------+-------------------
   A |   null |    ch | praneeth | {'2021-10-21 18:30:00.000000+0000': 'cassandra', '2021-10-27 18:30:00.000000+0000': 'hadoop'} | ['bhavani island']

(1 rows)
```

   iv. Append a new value in the todo_list column whose uid is ‘A’

```
cqlsh:chp> update users set todo_list = todo_list + {'2021-11-10 09:00':'spark'} where uid = 'A';
```

   v. Retrieve todo_list column from the table.

```
cqlsh:chp> select todo_list from users;

 todo_list
--------------------------------------------------------------------------------------------------------------------------------
 {'2021-10-21 18:30:00.000000+0000': 'cassandra', '2021-10-27 18:30:00.000000+0000': 'hadoop', '2021-11-10 03:30:00.000000+0000': 'spark'}

(1 rows)
```

vi. Delete the element of todo_list with key '2021-11-10 09:00' whose uid is ‘A’

```
cqlsh:chp> delete todo_list ['2021-11-10 09:00'] from users where uid = 'A';
```

vii. Read data from the table.

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

 uid | emails | fname | lname    | todo_list
| top_places
-----+--------+-------+----------+----------------------------------------------------------------------------------------------+--------------------
   A |   null |    ch | praneeth | {'2021-10-21 18:30:00.000000+0000':
'cassandra', '2021-10-27 18:30:00.000000+0000': 'hadoop'} | ['bhavani
island']

(1 rows)
```

Q) What is a Counter variable? Explain its purpose with suitable example.

A **counter** is a special column that is changed in increments.
Eg. We use counter column to count the number of times a particular book is issued from the library to the student.

i. Create a table lib_book with a counter column

```
cqlsh:chp> create table lib_book(counter_val counter, bname varchar,
sname varchar, Primary Key(bname,sname));
```

ii. Updating counter variable

```
cqlsh:chp> update lib_book set counter_val = counter_val + 1 where
bname='FBDA' and sname = 'chp';
```

iii. Reading data from the table.

```
cqlsh:chp> select * from lib_book;

 bname | sname | counter_val
-------+-------+-------------
  FBDA |   chp |           1

(1 rows)
```

iv. Updating counter variable with new student and existing textbook

```
cqlsh:chp> update lib_book set counter_val = counter_val + 1 where
bname='FBDA' and sname = 'vr';
```

v. Updating counter variable with existing student and textbook.

```
cqlsh:chp> update lib_book set counter_val = counter_val + 1 where
bname='FBDA' and sname = 'chp';
```

vi. Reading data from the table.

```
cqlsh:chp> select * from lib_book;
```

```
 bname | sname | counter_val
-------+-------+-------------
  FBDA |   chp |           2
  FBDA |    vr |           1

(2 rows)
```

Q) What is TTL? Given an example to understand TTL purpose.

Data in a column, other than a counter column, can have an optional expiration period called **TTL**(Time To Live).
The TTL is specified in seconds.
Eg.

i. Create a new table userlogin with columns uid as primary key and pwd.

```
cqlsh:chp> create table userlogin(uid int Primary Key, pwd text);
```

ii. Insert data into the table using TTL

```
cqlsh:chp> insert into userlogin(uid,pwd) values (1,'pvpsit') using TTL 30;
```

iii. Read time remaining for the pwd

```
cqlsh:chp> select TTL(pwd) from userlogin where uid = 1;

 ttl(pwd)
----------
       28
```

iv. Read data from the table before time expires

```
cqlsh:chp> select * from userlogin;

 uid | pwd
-----+--------
   1  | pvpsit
```

v. Read time remaining for the pwd

```
cqlsh:chp> select TTL(pwd) from userlogin where uid = 1;

 ttl(pwd)
----------
```

vi. Read data from the table after TTL expires on pwd

```
cqlsh:chp> select * from userlogin;

 uid | pwd
-----+-----
(0 rows)
```

Q) Explain different alter commands in CQL with suitable examples.

Alter commands are used to bring about changes to the structure of the table or column family.

Creating table in a keyspace 'chp':

```
cqlsh:stud> use chp;
cqlsh:chp> create table sample(sid text,sname text,Primary Key(sid));
cqlsh:chp> insert into sample(sid,sname) values('s101','Big Data');
cqlsh:chp> select * from sample;

 sid  | sname
------+----------
 s101 | Big Data
(1 rows)
```

1. **Alter table to change data type of a column.**
   i. Alter the schema of the table "sample". Change the data type of the column "sid" to int from text.

```
cqlsh:chp> alter table sample alter sid type int
```

NOTE: above query not supported in latest versions. InvalidRequest: Error from server: code=2200 [Invalid query] message="Altering of types is not allowed"

2. **Alter table to delete a column**
   ii. Delete the column "sname".

```
cqlsh:chp> alter table sample drop sname;
cqlsh:chp> select * from sample;

 sid
------
 s101
(1 rows)
```

NOTE: you can't delete a primary key column.

3. **Drop a table**
   iii. Drop the column family "sample"

```
cqlsh:chp> drop table sample;
cqlsh:chp> describe tables;

users  userlogin  proj_details  stud_info  lib_book
```

4. **Drop a keyspace.**
   iv. Drop the keyspace "chp"

```
cqlsh:chp> drop keyspace 'chp';
cqlsh:chp> describe keyspace 'chp';
keyspace 'chp' not found
```

Q) Illustrate working with CSV files.

1. **Copying data from a csv file into a table.**

   i. Create a table

```
cqlsh:chp> create table faculty (fid int primary key, fname text, dept text);
```

   ii. Viewing data in the table.

```
cqlsh:chp> select * from faculty;

 fid | dept | fname
-----+------+-------

(0 rows)
```

   iii. Copying data from the csv file faculty.csv into the table faculty.

```
cqlsh:chp> copy faculty(fid,fname,dept) from
'/home/hadoop/Desktop/faculty.csv';
```

   iv. Viewing data in the table after copying from csv file.

```
cqlsh:chp> select * from faculty;

 fid | dept | fname
-----+------+-------
   1 |   it |   chp
   2 |  cse |    vr
```

2. **Copying data from the table to a csv file.**

   i. Create a table
   ii. Inserting data into the table

```
cqlsh:chp> insert into faculty (fid,fname,dept) values (3,'pnr','it');
```

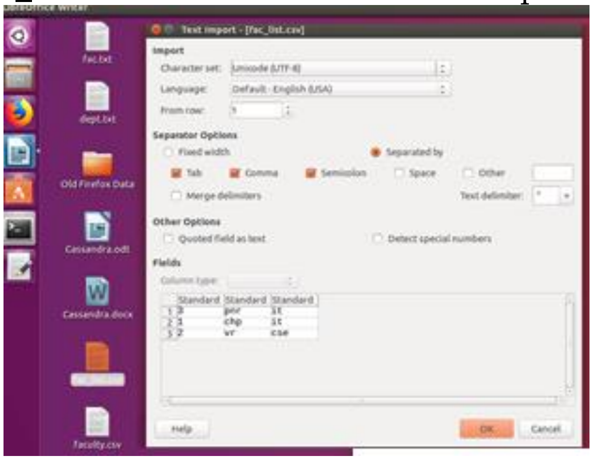
   iii. Viewing data in the table

```
cqlsh:chp> select * from faculty;

 fid | dept | fname
-----+------+-------
   1 |   it |   chp
   2 |  cse |    vr
   3 |   it |   pnr

(3 rows)
```

   iv. Copying data from the table faculty to fac_list.csv file

```
cqlsh:chp> copy faculty(fid,fname,dept) to
'/home/hadoop/Desktop/fac_list.csv';
```

v. Viewing fac_list.csv file that contains the copied data.


Text import - [fac_list.csv]

Import
Character set: Unicode (UTF-8)
Language: Default - English (USA)
From row: 1

Separator Options
Fixed width [✓ Separated by]
[✓ Tab] [✓ Comma] [✓ Semicolon] Space Other
Merge delimiters
Text delimiter: "

Other Options
Quoted field as text
Detect special numbers

Fields

	Standard	Standard	Standard
1	3	pnr	it
2	1	chp	it
3	2	vr	cse

Q) Illustrate how to read records from standard input and vice versa with suitable examples.

1. **Reading records from standard input**

i. Create table
ii. Reading records from standard input

```
cqlsh:chp> copy faculty(fid,fname,dept) from stdin;

Using 1 child processes
Starting copy of chp.faculty with columns [fid, fname, dept].
[Use . on a line by itself to end input]
[copy] 4,'vijay','it'
[copy] 5,'ravi','cse'
[copy] .

Processed: 2 rows; Rate:       0 rows/s; Avg. rate:       0 rows/s
2 rows imported from 1 files in 57.864 seconds (0 skipped).
```

iii. Viewing data from the table

```
cqlsh:chp> select * from faculty;

 fid | dept  | fname
-----+-------+---------
   5 | 'cse' |  'ravi'
   1 |    it |     chp
   2 |   cse |      vr
   4 |  'it' | 'vijay'
   3 |    it |     pnr

(5 rows)
```

2. **Displaying data on to the standard input**

```
cqlsh:chp> copy faculty(fid,fname,dept) to stdout;

3,pnr,it
1,chp,it
2,vr,cse
4,'vijay','it'
5,'ravi','cse'
```

Q) Answer for the following queries

i. Create a table "earnings" with columns:sid, cid, corder, title, coordinator with primary key as id and corder

```
cqlsh> use chp;
cqlsh:chp> create table earnings(sid int, cid int,corder int, title text,coordinator text, Primary Key(sid,corder));
```

ii. Insert data into the table

```
cqlsh:chp> insert into earnings(sid,cid,corder,title,coordinator) values(101,1,1001,'Cassandra','chp');
cqlsh:chp> insert into earnings(sid,cid,corder,title,coordinator) values(101,2,1002,'Mongo DB','vr');
cqlsh:chp> insert into earnings(sid,cid,corder,title,coordinator) values(101,3,1003,'Hadoop','chp');
```

iii. Display all the records in the column family.

```
cqlsh:chp> select * from earnings;

sid | corder | cid | coordinator | title
-----+--------+-----+-------------+-----------
 101 |   1001 |   1 |         chp | Cassandra
 101 |   1002 |   2 |          vr |  Mongo DB
 101 |   1003 |   3 |         chp |    Hadoop
```

iv. Retrieve data from the table with coordinator as chp.

```
cqlsh:chp> select * from earnings where coordinator = 'chp' allow filtering;

sid | corder | cid | coordinator | title
-----+--------+-----+-------------+-----------
 101 |   1001 |   1 |         chp | Cassandra
 101 |   1003 |   3 |         chp |    Hadoop
```

v. Display only the 1st record from the table.

```
cqlsh:chp> select * from earnings limit 1;

 sid | corder | cid | coordinator | title
-----+--------+-----+-------------+-----------
 101 |   1001 |   1 |         chp | Cassandra
```

vi. Display data from the table with sid = 101 and sort by corder in descending order.

```
cqlsh:chp> select * from earnings where sid = 101 order by corder desc limit 5;

 sid | corder | cid | coordinator | title
-----+--------+-----+-------------+-----------
 101 |   1003 |   3 |         chp |     Hadoop
 101 |   1002 |   2 |          vr |  Mongo DB
 101 |   1001 |   1 |         chp | Cassandra

(3 rows)
```

vii. Delete all records in the table.

```
cqlsh:chp> truncate table earnings;
cqlsh:chp> select * from earnings;

 sid | corder | cid | coordinator | title
-----+--------+-----+-------------+-------

(0 rows)
```

Q) Differentiate SQL and CQL.

Characteristic	**SQL**	**CQL**
Consistency	Strong	Eventual
Data Reference(Foreign Key)	Yes	Denormalized data
Join	Yes	We have to use 3rd party tools
Where clause	Yes	Yes, but performance hits can apply for non-key columns
Order by cluase	Yes	Can only be applied to a clustering column.