UNIT- VI

Cassandra

Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world. It provides highly available service with no single point of failure.

Listed below are some of the notable points of Apache Cassandra:

- It is scalable, fault-tolerant, and consistent.
- It is a column-oriented database.
- Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.
- Created at Facebook, it differs sharply from relational database management systems.
- Cassandra implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.
- Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more.

Features of Cassandra

Cassandra has become so popular because of its outstanding technical features. Given below are some of the features of Cassandra:

- **Elastic scalability** - Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on architecture** - Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast linear-scale performance** - Cassandra is linearly scalable, i.e., it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- **Flexible data storage** - Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy data distribution** - Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction support** - Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).

- **Fast writes** - Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

History of Cassandra

- Cassandra was developed at Facebook for inbox search.
- It was open-sourced by Facebook in July 2008.
- Cassandra was accepted into Apache Incubator in March 2009.
- It was made an Apache top-level project since February 2010.

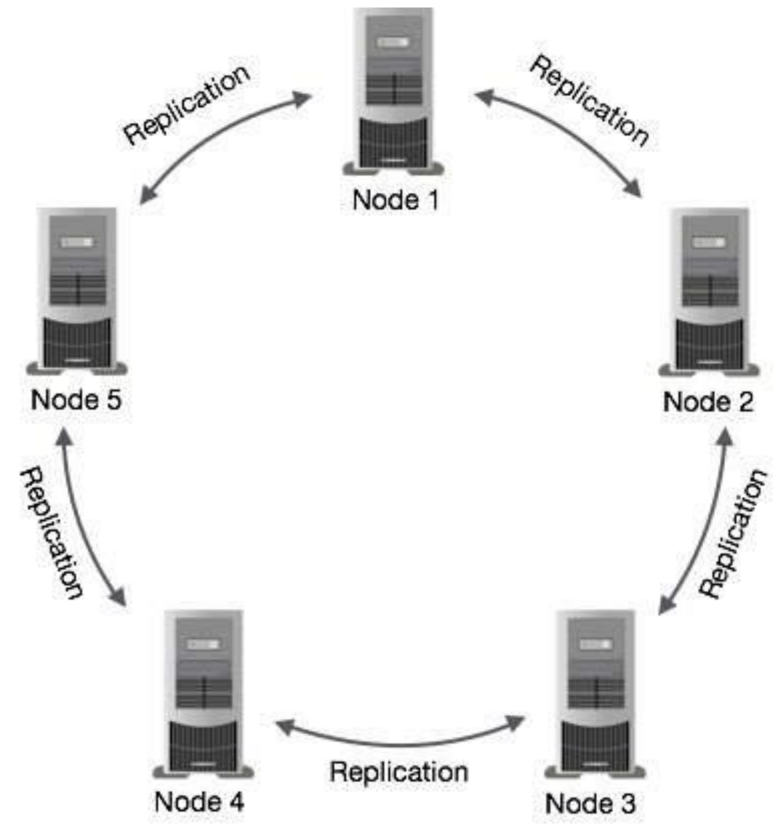
The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

- All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.
- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- When a node goes down, read/write requests can be served from other nodes in the network.

Data Replication in Cassandra

In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data. If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client. After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.

The following figure shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.



Note – Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

Components of Cassandra

The key components of Cassandra are as follows –

- **Node** – It is the place where data is stored.
- **Data center** – It is a collection of related nodes.
- **Cluster** – A cluster is a component that contains one or more data centers.
- **Commit log** – The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-table** – A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.

- **SSTable** − It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter** − These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

CQLSH

Users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers.

Clients approach any of the nodes for their read-write operations. That node (coordinator) plays a proxy between the client and the nodes holding the data.

Write Operations

Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table.** Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

Read Operations

During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

CRUD

Creating Data using Cqlsh

You can insert data into the columns of a row in a table using the command**INSERT**. Given below is the syntax for creating data in a table.

```
INSERT INTO <tablename>
(<column1 name>, <column2 name>....)
VALUES (<value1>, <value2>....)
USING <option>
```

Example

Let us assume there is a table called **emp** with columns (emp_id, emp_name, emp_city, emp_phone, emp_sal) and you have to insert the following data into the **emp** table.

emp_id	emp_name	emp_city	emp_phone	emp_sal
1	ram	Hyderabad	9848022338	50000
2	robin	Hyderabad	9848022339	40000

3	rahman	Chennai	9848022330	45000

Use the commands given below to fill the table with required data.

```
cqlsh:tutorialspoint> INSERT INTO emp (emp_id, emp_name, emp_city,
   emp_phone, emp_sal) VALUES(1,'ram', 'Hyderabad', 9848022338, 50000);

cqlsh:tutorialspoint> INSERT INTO emp (emp_id, emp_name, emp_city,
   emp_phone, emp_sal) VALUES(2,'robin', 'Hyderabad', 9848022339, 40000);

cqlsh:tutorialspoint> INSERT INTO emp (emp_id, emp_name, emp_city,
   emp_phone, emp_sal) VALUES(3,'rahman', 'Chennai', 9848022330, 45000);
```

Verification

After inserting data, use SELECT statement to verify whether the data has been inserted or not. If you verify the emp table using SELECT statement, it will give you the following output.

```
cqlsh:tutorialspoint> SELECT * FROM emp;

 emp_id |  emp_city | emp_name |  emp_phone | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 | 50000
      2 | Hyderabad |    robin | 9848022339 | 40000
      3 |   Chennai |   rahman | 9848022330 | 45000

(3 rows)
```

Here you can observe the table has populated with the data we inserted.

Creating Data using Java API

You can create data in a table using the execute() method of Session class. Follow the steps given below to create data in a table using java API.

Step1: Create a Cluster Object

Create an instance of **Cluster.builder** class of **com.datastax.driver.core**package as shown below.

```
//Creating Cluster.Builder object
Cluster.Builder builder1 = Cluster.builder();
```

Add a contact point (IP address of the node) using the **addContactPoint()**method of **Cluster.Builder** object. This method returns **Cluster.Builder**.

```
//Adding contact point to the Cluster.Builder object
Cluster.Builder builder2 = build.addContactPoint("127.0.0.1");
```

Using the new builder object, create a cluster object. To do so, you have a method called **build()** in the **Cluster.Builder** class. The following code shows how to create a cluster object.

```
//Building a cluster
Cluster cluster = builder.build();
```

You can build a cluster object using a single line of code as shown below.

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
```

Step 2: Create a Session Object

Create an instance of Session object using the connect() method of Cluster class as shown below.

```
Session session = cluster.connect( );
```

This method creates a new session and initializes it. If you already have a keyspace, then you can set it to the existing one by passing the KeySpace name in string format to this method as shown below.

```
Session session = cluster.connect(“ Your keyspace name ” );
```

Here we are using the KeySpace called **tp**. Therefore, create the session object as shown below.

```
Session session = cluster.connect(“ tp” );
```

Step 3: Execute Query

You can execute CQL queries using the execute() method of Session class. Pass the query either in string format or as a **Statement** class object to the execute() method. Whatever you pass to this method in string format will be executed on the **cqlsh**.

In the following example, we are inserting data in a table called **emp**. You have to store the query in a string variable and pass it to the execute() method as shown below.

```
String query1 = “INSERT INTO emp (emp_id, emp_name, emp_city, emp_phone, 
emp_sal)
   VALUES(1,'ram', 'Hyderabad', 9848022338, 50000);” ;

String query2 = “INSERT INTO emp (emp_id, emp_name, emp_city, emp_phone, 
emp_sal)
   VALUES(2,'robin', 'Hyderabad', 9848022339, 40000);” ;

String query3 = “INSERT INTO emp (emp_id, emp_name, emp_city, emp_phone, 
emp_sal)
   VALUES(3,'rahman', 'Chennai', 9848022330, 45000);” ;

session.execute(query1);
session.execute(query2);
session.execute(query3);
```

Given below is the complete program to insert data into a table in Cassandra using Java API.

```
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;


public class Create_Data {


   public static void main(String args[]){


      //queries
      String query1 = "INSERT INTO emp (emp_id, emp_name, emp_city, 
emp_phone,  emp_sal)"
```

```
         + " VALUES(1,'ram', 'Hyderabad', 9848022338, 50000);" ;


      String query2 = "INSERT INTO emp (emp_id, emp_name, emp_city,
         emp_phone, emp_sal)"


         + " VALUES(2,'robin', 'Hyderabad', 9848022339, 40000);" ;


      String query3 = "INSERT INTO emp (emp_id, emp_name, emp_city,
emp_phone, emp_sal)"


         + " VALUES(3,'rahman', 'Chennai', 9848022330, 45000);" ;


      //Creating Cluster object
      Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();


      //Creating Session object
      Session session = cluster.connect("tp");


      //Executing the query
      session.execute(query1);


      session.execute(query2);


      session.execute(query3);


      System.out.println("Data created");
   }
}
```

Save the above program with the class name followed by .java, browse to the location where it is saved. Compile and execute the program as shown below.

```
$javac Create_Data.java
$java Create_Data
```

Under normal conditions, it should produce the following output:

Updating Data using Cqlsh

UPDATE is the command used to update data in a table. The following keywords are used while updating data in a table:

- **Where** - This clause is used to select the row to be updated.
- **Set** - Set the value using this keyword.
- **Must** - Includes all the columns composing the primary key.

While updating rows, if a given row is unavailable, then UPDATE creates a fresh row. Given below is the syntax of UPDATE command:

```
UPDATE <tablename>
SET <column name> = <new value>
<column name> = <value>....
WHERE <condition>
```

Example

Assume there is a table named **emp**. This table stores the details of employees of a certain company, and it has the following details:

emp_id	**emp_name**	**emp_city**	**emp_phone**	**emp_sal**
1	ram	Hyderabad	9848022338	50000
2	robin	Hyderabad	9848022339	40000
3	rahman	Chennai	9848022330	45000

Let us now update emp_city of robin to Delhi, and his salary to 50000. Given below is the query to perform the required updates.

```
cqlsh:tutorialspoint> UPDATE emp SET emp_city='Delhi',emp_sal=50000
   WHERE emp_id=2;
```

Verification

Use SELECT statement to verify whether the data has been updated or not. If you verify the emp table using SELECT statement, it will produce the following output.

```
cqlsh:tutorialspoint> select * from emp;

 emp_id |  emp_city | emp_name |  emp_phone | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 | 50000
      2 |     Delhi |    robin | 9848022339 | 50000
      3 |   Chennai |   rahman | 9848022330 | 45000

(3 rows)
```

Here you can observe the table data has got updated.

Updating Data using Java API

You can update data in a table using the execute() method of Session class. Follow the steps given below to update data in a table using Java API.

Step1: Create a Cluster Object

Create an instance of **Cluster.builder** class of **com.datastax.driver.core**package as shown below.

```
//Creating Cluster.Builder object
Cluster.Builder builder1 = Cluster.builder();
```

Add a contact point (IP address of the node) using the **addContactPoint()**method of **Cluster.Builder** object. This method returns **Cluster.Builder**.

```
//Adding contact point to the Cluster.Builder object
Cluster.Builder builder2 = build.addContactPoint("127.0.0.1");
```

Using the new builder object, create a cluster object. To do so, you have a method called **build()** in the **Cluster.Builder** class. Use the following code to create the cluster object.

```
//Building a cluster
Cluster cluster = builder.build();
```

You can build the cluster object using a single line of code as shown below.

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
```

Step 2: Create a Session Object

Create an instance of Session object using the connect() method of Cluster class as shown below.

```
Session session = cluster.connect( );
```

This method creates a new session and initializes it. If you already have a keyspace, then you can set it to the existing one by passing the KeySpace name in string format to this method as shown below.

```
Session session = cluster.connect(“ Your keyspace name”);
```

Here we are using the KeySpace named **tp**. Therefore, create the session object as shown below.

```
Session session = cluster.connect(“tp”);
```

Step 3: Execute Query

You can execute CQL queries using the execute() method of Session class. Pass the query either in string format or as a Statement class object to the execute() method. Whatever you pass to this method in string format will be executed on the **cqlsh**.

In the following example, we are updating the emp table. You have to store the query in a string variable and pass it to the execute() method as shown below:

```
String query = “ UPDATE emp SET emp_city='Delhi',emp_sal=50000
WHERE emp_id = 2;” ;
```

Given below is the complete program to update data in a table using Java API.

```
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;

public class Update_Data {

   public static void main(String args[]){

      //query
      String query = " UPDATE emp SET emp_city='Delhi',emp_sal=50000"

      //Creating Cluster object
      Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();

      //Creating Session object
      Session session = cluster.connect("tp");

      //Executing the query
      session.execute(query);

      System.out.println("Data updated");
   }
}
```

Save the above program with the class name followed by .java, browse to the location where it is saved. Compile and execute the program as shown below.

```
$javac Update_Data.java
$java Update_Data
```

Under normal conditions, it should produce the following output:

```
Data updated
```

Counter:

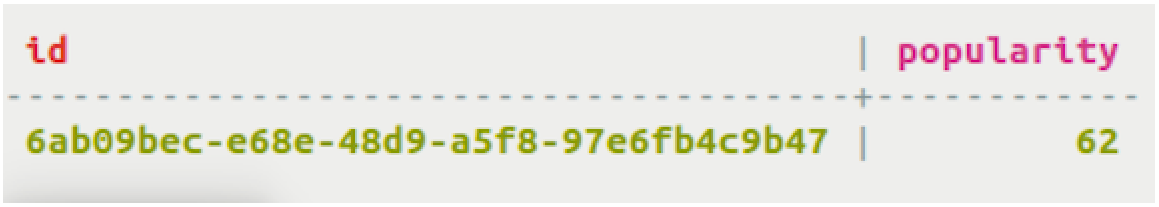
Creating a counter table

A counter is a special column used to store an integer that is changed in increments.

Counters are useful for many data models. Some examples:

- To keep track of the number of web page views received on a company website
- To keep track of the number of games played online or the number of players who have joined an online game

The table shown below uses **id** as the primary key and keeps track of the popularity of a cyclist based on thumbs up/thumbs down clicks in the **popularity** field of a counter table.



```
id                                   | popularity
-------------------------------------+------------
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |         62
```

Tracking count in a distributed database presents an interesting challenge. In Cassandra, at any given moment, the counter value may be stored in the Memtable, commit log, and/or one or more SSTables. Replication between nodes can cause consistency issues in certain edge cases. Cassandra counters were redesigned in Cassandra 2.1 to alleviate some of the difficulties.
Read "What's New in Cassandra 2.1: Better Implementation of Counters" to discover the improvements made in the counters.

Because counters are implemented differently from other columns, counter columns can only be created in dedicated tables. A counter column must have the datatype counter data type. This data type cannot be assigned to a column that serves as the primary key or partition key. To implement a counter column, create a table that only includes:

- The primary key (can be one or more columns)
- The counter column

Many counter-related settings can be set in the cassandra.yaml file.

A counter column cannot be indexed or deleted.. To load data into a counter column, or to increase or decrease the value of the counter, use the UPDATE command. Cassandra rejects USING TIMESTAMP orUSING TTL when updating a counter column.

To create a table having one or more counter columns, use this:

- Use CREATE TABLE to define the counter and non-counter columns. Use all non-counter columns as part of the PRIMARY KEY definition.

- Using a counter
  A counter is a special column for storing a number that is changed in increments.

List, Set, Map, Tracing, Hands on Practice :

CQL provides the facility of using Collection data types. Using these Collection types, you can store multiple values in a single variable. This chapter explains how to use Collections in Cassandra.

List

List is used in the cases where

- the order of the elements is to be maintained, and
- a value is to be stored multiple times.

You can get the values of a list data type using the index of the elements in the list.

Creating a Table with List

Given below is an example to create a sample table with two columns, name and email. To store multiple emails, we are using list.

```
cqlsh:tutorialspoint> CREATE TABLE data(name text PRIMARY KEY, email
list<text>);
```

Inserting Data into a List

While inserting data into the elements in a list, enter all the values separated by comma within square braces [ ] as shown below.

```
cqlsh:tutorialspoint> INSERT INTO data(name, email) VALUES ('ramu',
['abc@gmail.com','cba@yahoo.com'])
```

Updating a List

Given below is an example to update the list data type in a table called **data**. Here we are adding another email to the list.

```
cqlsh:tutorialspoint> UPDATE data
... SET email = email +['xyz@tutorialspoint.com']
... where name = 'ramu';
```

Verification

If you verify the table using SELECT statement, you will get the following result:

```
cqlsh:tutorialspoint> SELECT * FROM data;

 name | email
------+--------------------------------------------------------------
 ramu | ['abc@gmail.com', 'cba@yahoo.com', 'xyz@tutorialspoint.com']

(1 rows)
```

SET

Set is a data type that is used to store a group of elements. The elements of a set will be returned in a sorted order.

Creating a Table with Set

The following example creates a sample table with two columns, name and phone. For storing multiple phone numbers, we are using set.

```
cqlsh:tutorialspoint> CREATE TABLE data2 (name text PRIMARY KEY, phone
set<varint>);
```

Inserting Data into a Set

While inserting data into the elements in a set, enter all the values separated by comma within curly braces { } as shown below.

```
cqlsh:tutorialspoint> INSERT INTO data2(name, phone)VALUES ('rahman',
{9848022338,9848022339});
```

Updating a Set

The following code shows how to update a set in a table named data2. Here we are adding another phone number to the set.

```
cqlsh:tutorialspoint> UPDATE data2
   ... SET phone = phone + {9848022330}
   ... where name = 'rahman';
```

Verification

If you verify the table using SELECT statement, you will get the following result:

```
cqlsh:tutorialspoint> SELECT * FROM data2;

   name | phone
--------+--------------------------------------
 rahman | {9848022330, 9848022338, 9848022339}

(1 rows)
```

MAP

Map is a data type that is used to store a key-value pair of elements.

Creating a Table with Map

The following example shows how to create a sample table with two columns, name and address. For storing multiple address values, we are using map.

```
cqlsh:tutorialspoint> CREATE TABLE data3 (name text PRIMARY KEY, address
map<timestamp, text>);
```

Inserting Data into a Map

While inserting data into the elements in a map, enter all the **key : value** pairs separated by comma within curly braces { } as shown below.

```
cqlsh:tutorialspoint> INSERT INTO data3 (name, address)
   VALUES ('robin', {'home' : 'hyderabad' , 'office' : 'Delhi' } );
```

Updating a Set

The following code shows how to update the map data type in a table named data3. Here we are changing the value of the key office, that is, we are changing the office address of a person named robin.

```
cqlsh:tutorialspoint> UPDATE data3
   ... SET address = address+{'office':'mumbai'}
   ... WHERE name = 'robin';
```

Verification

If you verify the table using SELECT statement, you will get the following result:

```
cqlsh:tutorialspoint> select * from data3;

  name | address
-------+-------------------------------------------
 robin | {'home': 'hyderabad', 'office': 'mumbai'}

(1 rows)
```