Contents

[**Cassandra** 2](#_Toc54207200)

[Cassandra - Introduction 2](#_Toc54207201)

[**NoSQLDatabase** 2](#_Toc54207202)

[**NoSQL vs. Relational Database** 2](#_Toc54207203)

[**What is Apache Cassandra?** 3](#_Toc54207204)

[**Features of Cassandra** 3](#_Toc54207205)

[**History of Cassandra** 4](#_Toc54207206)

[Cassandra - Architecture 4](#_Toc54207207)

[**Data Replication in Cassandra** 4](#_Toc54207208)

[**Components of Cassandra** 5](#_Toc54207209)

[**Cassandra Query Language** 6](#_Toc54207210)

[**Write Operations** 6](#_Toc54207211)

[**Read Operations** 6](#_Toc54207212)

[Cassandra - Data Model 6](#_Toc54207213)

[**Cluster** 6](#_Toc54207214)

[**Keyspace** 6](#_Toc54207215)

[**Column Family** 7](#_Toc54207216)

[**Column** 8](#_Toc54207217)

[**SuperColumn** 8](#_Toc54207218)

[**Data Models of Cassandra and RDBMS** 8](#_Toc54207219)

[Cassandra – Installation 9](#_Toc54207220)

[**Pre-Installation Setup** 9](#_Toc54207221)

[**Create a User** 9](#_Toc54207222)

[**SSH Setup and Key Generation** 10](#_Toc54207223)

[**Installing Java** 10](#_Toc54207224)

[**Step 1** 10](#_Toc54207225)

[**Step 2** 10](#_Toc54207226)

[**Step 3** 11](#_Toc54207227)

[**Step 4** 11](#_Toc54207228)

[**Step 5** 11](#_Toc54207229)

[**Setting the Path** 11](#_Toc54207230)

[**Download Cassandra** 11](#_Toc54207231)

[**Configure Cassandra** 12](#_Toc54207232)

[**Create Directories** 12](#_Toc54207233)

[**Give Permissions to Folders** 12](#_Toc54207234)

[**Start Cassandra** 12](#_Toc54207235)

[**Programming Environment** 12](#_Toc54207236)

[**Eclipse Environment** 13](#_Toc54207237)

[**Maven Dependencies** 16](#_Toc54207238)

[Cassandra - Referenced Api 17](#_Toc54207239)

[**Cluster** 17](#_Toc54207240)

[**Methods** 18](#_Toc54207241)

[**Cluster.Builder** 18](#_Toc54207242)

[**Methods** 18](#_Toc54207243)

[**Session** 18](#_Toc54207244)

[**Methods** 18](#_Toc54207245)

[Cassandra – Cqlsh 19](#_Toc54207246)

[**Starting cqlsh** 19](#_Toc54207247)

[**Cqlsh Commands** 20](#_Toc54207248)

[**Documented Shell Commands** 20](#_Toc54207249)

[**CQL Data Definition Commands** 21](#_Toc54207250)

[**CQL Data Manipulation Commands** 21](#_Toc54207251)

[**CQL Clauses** 21](#_Toc54207252)

[Cassandra - Shell Commands 22](#_Toc54207253)

[**Help** 22](#_Toc54207254)

[**Capture** 22](#_Toc54207255)

[**Consistency** 23](#_Toc54207256)

[**Copy** 23](#_Toc54207257)

[**Describe** 24](#_Toc54207258)

[**Describe Type** 25](#_Toc54207259)

[**Describe Types** 25](#_Toc54207260)

[**Expand** 25](#_Toc54207261)

[**Exit** 26](#_Toc54207262)

[**Show** 26](#_Toc54207263)

[**Source** 27](#_Toc54207264)

[Cassandra - Create Keyspace 28](#_Toc54207265)

[**Creating a Keyspace using Cqlsh** 28](#_Toc54207266)

[**Syntax** 28](#_Toc54207267)

[**Replication** 28](#_Toc54207268)

[**Example** 28](#_Toc54207269)

[**Verification** 29](#_Toc54207270)

[**Durable\_writes** 29](#_Toc54207271)

[**Example** 29](#_Toc54207272)

[**Verification** 29](#_Toc54207273)

[**Using a Keyspace** 30](#_Toc54207274)

[**Example** 30](#_Toc54207275)

[**Creating a Keyspace using Java API** 30](#_Toc54207276)

[**Step1: Create a Cluster Object** 30](#_Toc54207277)

[**Step 2: Create a Session Object** 30](#_Toc54207278)

[**Step 3: Execute Query** 31](#_Toc54207279)

[**Step4 : Use the KeySpace** 31](#_Toc54207280)

[Cassandra - Alter Keyspace 32](#_Toc54207281)

[**Altering a KeySpace** 32](#_Toc54207282)

[**Syntax** 32](#_Toc54207283)

[**Replication** 32](#_Toc54207284)

[**Durable\_writes** 32](#_Toc54207285)

[**Example** 32](#_Toc54207286)

[**Altering Durable\_writes** 32](#_Toc54207287)

[**Altering a Keyspace using Java API** 33](#_Toc54207288)

[**Step1: Create a Cluster Object** 33](#_Toc54207289)

[**Step 2: Create a Session Object** 34](#_Toc54207290)

[**Step 3: Execute Query** 34](#_Toc54207291)

[Cassandra - Drop Keyspace 35](#_Toc54207292)

[**Dropping a Keyspace** 35](#_Toc54207293)

[**Syntax** 35](#_Toc54207294)

[**Example** 35](#_Toc54207295)

[**Verification** 35](#_Toc54207296)

[**Dropping a Keyspace using Java API** 36](#_Toc54207297)

[**Step1: Create a Cluster Object** 36](#_Toc54207298)

[**Step 2: Create a Session Object** 36](#_Toc54207299)

[**Step 3: Execute Query** 36](#_Toc54207300)

[Cassandra - Create Table 37](#_Toc54207301)

[**Creating a Table** 37](#_Toc54207302)

[**Syntax** 37](#_Toc54207303)

[**Defining a Column** 38](#_Toc54207304)

[**Primary Key** 38](#_Toc54207305)

[**Example** 38](#_Toc54207306)

[**Verification** 39](#_Toc54207307)

[**Creating a Table using Java API** 39](#_Toc54207308)

[**Step1: Create a Cluster Object** 39](#_Toc54207309)

[**Step 2: Create a Session Object** 39](#_Toc54207310)

[**Step 3: Execute Query** 40](#_Toc54207311)

[Cassandra - Alter Table 41](#_Toc54207312)

[**Altering a Table** 41](#_Toc54207313)

[**Syntax** 41](#_Toc54207314)

[**Adding a Column** 41](#_Toc54207315)

[**Dropping a Column** 42](#_Toc54207316)

[**Altering a Table using Java API** 42](#_Toc54207317)

[**Step1: Create a Cluster Object** 42](#_Toc54207318)

[**Step 2: Create a Session Object** 43](#_Toc54207319)

[**Step 3: Execute Query** 43](#_Toc54207320)

[**Deleting a Column** 44](#_Toc54207321)

[Cassandra - Drop Table 44](#_Toc54207322)

[**Dropping a Table** 45](#_Toc54207323)

[**Syntax** 45](#_Toc54207324)

[**Example** 45](#_Toc54207325)

[**Verification** 45](#_Toc54207326)

[**Deleting a Table using Java API** 45](#_Toc54207327)

[**Step1: Create a Cluster Object** 45](#_Toc54207328)

[**Step 2: Create a Session Object** 45](#_Toc54207329)

[**Step 3: Execute Query** 46](#_Toc54207330)

[Cassandra - Truncate Table 47](#_Toc54207331)

[**Truncating a Table** 47](#_Toc54207332)

[**Syntax** 47](#_Toc54207333)

[**Example** 47](#_Toc54207334)

[**Verification** 47](#_Toc54207335)

[**Truncating a Table using Java API** 48](#_Toc54207336)

[**Step1: Create a Cluster Object** 48](#_Toc54207337)

[**Step 2: Creating a Session Object** 48](#_Toc54207338)

[**Step 3: Execute Query** 48](#_Toc54207339)

[Cassandra - Create Index 49](#_Toc54207340)

[**Creating an Index using Cqlsh** 49](#_Toc54207341)

[**Creating an Index using Java API** 50](#_Toc54207342)

[**Step1: Create a Cluster Object** 50](#_Toc54207343)

[**Step 2: Create a Session Object** 50](#_Toc54207344)

[**Step 3: Execute Query** 50](#_Toc54207345)

[Cassandra - Drop Index 51](#_Toc54207346)

[**Dropping an Index** 51](#_Toc54207347)

[**Dropping an Index using Java API** 52](#_Toc54207348)

[**Step1: Create a Cluster Object** 52](#_Toc54207349)

[**Step 2: Create a Session Object** 52](#_Toc54207350)

[**Step 3: Execute Query** 52](#_Toc54207351)

[Cassandra - Batch Statements 53](#_Toc54207352)

[**Using Batch Statements** 53](#_Toc54207353)

[**Example** 54](#_Toc54207354)

[**Verification** 54](#_Toc54207355)

[**Batch Statements using Java API** 55](#_Toc54207356)

[**Step1: Create a Cluster Object** 55](#_Toc54207357)

[**Step 2: Create a Session Object** 55](#_Toc54207358)

[**Step 3: Execute Query** 55](#_Toc54207359)

[Cassandra - Create Data 57](#_Toc54207360)

[**Creating Data in a Table** 57](#_Toc54207361)

[**Example** 57](#_Toc54207362)

[**Verification** 57](#_Toc54207363)

[**Creating Data using Java API** 58](#_Toc54207364)

[**Step1: Create a Cluster Object** 58](#_Toc54207365)

[**Step 2: Create a Session Object** 58](#_Toc54207366)

[**Step 3: Execute Query** 59](#_Toc54207367)

[Cassandra - Update Data 60](#_Toc54207368)

[**Updating Data in a Table** 60](#_Toc54207369)

[**Example** 60](#_Toc54207370)

[**Verification** 61](#_Toc54207371)

[**Updating Data using Java API** 61](#_Toc54207372)

[**Step1: Create a Cluster Object** 61](#_Toc54207373)

[**Step 2: Create a Session Object** 62](#_Toc54207374)

[**Step 3: Execute Query** 62](#_Toc54207375)

[Cassandra - Read Data 63](#_Toc54207376)

[**Reading Data using Select Clause** 63](#_Toc54207377)

[**Example** 63](#_Toc54207378)

[**Reading Required Columns** 64](#_Toc54207379)

[**Where Clause** 64](#_Toc54207380)

[**Reading Data using Java API** 64](#_Toc54207381)

[**Step1:Create a Cluster Object** 64](#_Toc54207382)

[**Step 2: Create a Session Object** 65](#_Toc54207383)

[**Step 3: Execute Query** 65](#_Toc54207384)

[**Step 4: Get the ResultSet Object** 65](#_Toc54207385)

[Cassandra - Delete Data 66](#_Toc54207386)

[**Deleting Datafrom a Table** 66](#_Toc54207387)

[**Example** 66](#_Toc54207388)

[**Verification** 67](#_Toc54207389)

[**Deleting an Entire Row** 67](#_Toc54207390)

[**Verification** 67](#_Toc54207391)

[**Deleting Data using Java API** 67](#_Toc54207392)

[**Step1: Create a Cluster Object** 67](#_Toc54207393)

[**Step 2: Create a Session Object** 68](#_Toc54207394)

[**Step 3: Execute Query** 68](#_Toc54207395)

[Cassandra - CQL Datatypes 69](#_Toc54207396)

[**Collection Types** 70](#_Toc54207397)

[**User-defined datatypes** 71](#_Toc54207398)

[Cassandra - CQL Collections 71](#_Toc54207399)

[**List** 71](#_Toc54207400)

[**Creating a Table with List** 71](#_Toc54207401)

[**Inserting Data into a List** 71](#_Toc54207402)

[**Updating a List** 72](#_Toc54207403)

[**Verification** 72](#_Toc54207404)

[**SET** 72](#_Toc54207405)

[**Creating a Table with Set** 72](#_Toc54207406)

[**Inserting Data into a Set** 72](#_Toc54207407)

[**Updating a Set** 72](#_Toc54207408)

[**Verification** 72](#_Toc54207409)

[**MAP** 73](#_Toc54207410)

[**Creating a Table with Map** 73](#_Toc54207411)

[**Inserting Data into a Map** 73](#_Toc54207412)

[**Updating a Set** 73](#_Toc54207413)

[**Verification** 73](#_Toc54207414)

[Cassandra - CQL User Defined Datatypes 73](#_Toc54207415)

[**Creating a User-defined Data Type** 74](#_Toc54207416)

[**Example** 74](#_Toc54207417)

[**Verification** 74](#_Toc54207418)

[**Altering a User-defined Data Type** 75](#_Toc54207419)

[**Adding a Field to a Type** 75](#_Toc54207420)

[**Verification** 75](#_Toc54207421)

[**Renaming a Field in a Type** 75](#_Toc54207422)

[**Verification** 75](#_Toc54207423)

[**Deleting a User-defined Data Type** 76](#_Toc54207424)

[**Example** 76](#_Toc54207425)

**Cassandra**

# Cassandra - Introduction

Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It is a type of NoSQL database. Let us first understand what a NoSQL database does.

## **NoSQLDatabase**

A NoSQL database (sometimes called as Not Only SQL) is a database that provides a mechanism to store and retrieve data other than the tabular relations used in relational databases. These databases are schema-free, support easy replication, have simple API, eventually consistent, and can handle huge amounts of data.

The primary objective of a NoSQL database is to have

* simplicity of design,
* horizontal scaling, and
* finer control over availability.

NoSql databases use different data structures compared to relational databases. It makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it must solve.

## **NoSQL vs. Relational Database**

The following table lists the points that differentiate a relational database from a NoSQL database.

|  |  |
| --- | --- |
| **Relational Database** | **NoSql Database** |
| Supports powerful query language. | Supports very simple query language. |
| It has a fixed schema. | No fixed schema. |
| Follows ACID (Atomicity, Consistency, Isolation, and Durability). | It is only “eventually consistent”. |
| Supports transactions. | Does not support transactions. |

Besides Cassandra, we have the following NoSQL databases that are quite popular −

* **Apache HBase** − HBase is an open source, non-relational, distributed database modeled after Google’s BigTable and is written in Java. It is developed as a part of Apache Hadoop project and runs on top of HDFS, providing BigTable-like capabilities for Hadoop.
* **MongoDB** − MongoDB is a cross-platform document-oriented database system that avoids using the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas making the integration of data in certain types of applications easier and faster.

## **What is Apache 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.

# Cassandra - Architecture

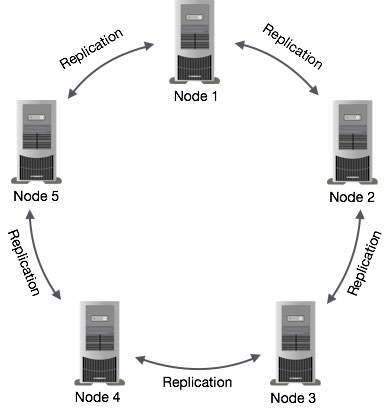
The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

* 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.

## **Cassandra Query Language**

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.

# Cassandra - Data Model

The data model of Cassandra is significantly different from what we normally see in an RDBMS. This chapter provides an overview of how Cassandra stores its data.

## **Cluster**

Cassandra database is distributed over several machines that operate together. The outermost container is known as the Cluster. For failure handling, every node contains a replica, and in case of a failure, the replica takes charge. Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.

### **Keyspace**

Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are −

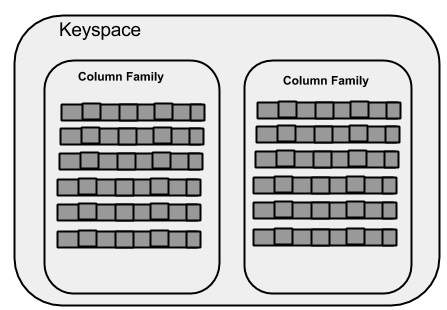
* **Replication factor** − It is the number of machines in the cluster that will receive copies of the same data.
* **Replica placement strategy** − It is nothing but the strategy to place replicas in the ring. We have strategies such as **simple strategy** (rack-aware strategy), **old network topology strategy** (rack-aware strategy), and **network topology strategy** (datacenter-shared strategy).
* **Column families** − Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

The syntax of creating a Keyspace is as follows −

CREATE KEYSPACE Keyspace name

WITH replication = {'class': 'SimpleStrategy', 'replication\_factor' : 3};

The following illustration shows a schematic view of a Keyspace.



### **Column Family**

A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

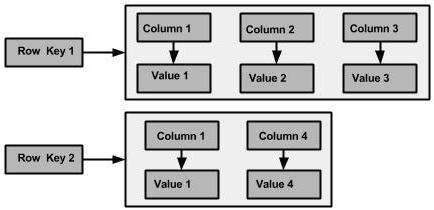
|  |  |
| --- | --- |
| **Relational Table** | **Cassandra column Family** |
| A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value. | In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time. |
| Relational tables define only columns and the user fills in the table with values. | In Cassandra, a table contains columns, or can be defined as a super column family. |

A Cassandra column family has the following attributes −

* **keys\_cached** − It represents the number of locations to keep cached per SSTable.
* **rows\_cached** − It represents the number of rows whose entire contents will be cached in memory.
* **preload\_row\_cache** − It specifies whether you want to pre-populate the row cache.

**Note −** Unlike relational tables where a column family’s schema is not fixed, Cassandra does not force individual rows to have all the columns.

The following figure shows an example of a Cassandra column family.



### **Column**

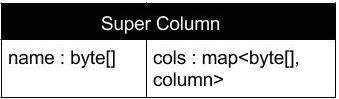
A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.



### **SuperColumn**

A super column is a special column, therefore, it is also a key-value pair. But a super column stores a map of sub-columns.

Generally column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that you are likely to query together in the same column family, and a super column can be helpful here.Given below is the structure of a super column.



## **Data Models of Cassandra and RDBMS**

The following table lists down the points that differentiate the data model of Cassandra from that of an RDBMS.

|  |  |
| --- | --- |
| **RDBMS** | **Cassandra** |
| RDBMS deals with structured data. | Cassandra deals with unstructured data. |
| It has a fixed schema. | Cassandra has a flexible schema. |
| In RDBMS, a table is an array of arrays. (ROW x COLUMN) | In Cassandra, a table is a list of “nested key-value pairs”. (ROW x COLUMN key x COLUMN value) |
| Database is the outermost container that contains data corresponding to an application. | Keyspace is the outermost container that contains data corresponding to an application. |
| Tables are the entities of a database. | Tables or column families are the entity of a keyspace. |
| Row is an individual record in RDBMS. | Row is a unit of replication in Cassandra. |
| Column represents the attributes of a relation. | Column is a unit of storage in Cassandra. |
| RDBMS supports the concepts of foreign keys, joins. | Relationships are represented using collections. |

# Cassandra – Installation

Cassandra can be accessed using cqlsh as well as drivers of different languages. This chapter explains how to set up both cqlsh and java environments to work with Cassandra.

## **Pre-Installation Setup**

Before installing Cassandra in Linux environment, we require to set up Linux using **ssh** (Secure Shell). Follow the steps given below for setting up Linux environment.

### **Create a User**

At the beginning, it is recommended to create a separate user for Hadoop to isolate Hadoop file system from Unix file system. Follow the steps given below to create a user.

* Open root using the command **“su”**.
* Create a user from the root account using the command **“useradd username”**.
* Now you can open an existing user account using the command **“su username”**.

Open the Linux terminal and type the following commands to create a user.

$ su

password:

# useradd hadoop

# passwd hadoop

New passwd:

Retype new passwd

### **SSH Setup and Key Generation**

SSH setup is required to perform different operations on a cluster such as starting, stopping, and distributed daemon shell operations. To authenticate different users of Hadoop, it is required to provide public/private key pair for a Hadoop user and share it with different users.

The following commands are used for generating a key value pair using SSH −

* copy the public keys form id\_rsa.pub to authorized\_keys,
* and provide owner,
* read and write permissions to authorized\_keys file respectively.

$ ssh-keygen -t rsa

$ cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

$ chmod 0600 ~/.ssh/authorized\_keys

* Verify ssh:

ssh localhost

## **Installing Java**

Java is the main prerequisite for Cassandra. First of all, you should verify the existence of Java in your system using the following command −

$ java -version

If everything works fine it will give you the following output.

java version "1.7.0\_71"

Java(TM) SE Runtime Environment (build 1.7.0\_71-b13)

Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)

If you don’t have Java in your system, then follow the steps given below for installing Java.

### **Step 1**

Download java (JDK <latest version> - X64.tar.gz) from the following [link:](http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html)

**Then jdk-7u71-linux-x64.tar.gz will be downloaded onto your system.**

### **Step 2**

Generally you will find the downloaded java file in the Downloads folder. Verify it and extract the **jdk-7u71-linux-x64.gz** file using the following commands.

$ cd Downloads/

$ ls

jdk-7u71-linux-x64.gz

$ tar zxf jdk-7u71-linux-x64.gz

$ ls

jdk1.7.0\_71 jdk-7u71-linux-x64.gz

### **Step 3**

To make Java available to all users, you have to move it to the location “/usr/local/”. Open root, and type the following commands.

$ su

password:

# mv jdk1.7.0\_71 /usr/local/

# exit

### **Step 4**

For setting up **PATH** and **JAVA\_HOME** variables, add the following commands to **~/.bashrc** file.

export JAVA\_HOME = /usr/local/jdk1.7.0\_71

export PATH = $PATH:$JAVA\_HOME/bin

Now apply all the changes into the current running system.

$ source ~/.bashrc

### **Step 5**

Use the following commands to configure java alternatives.

# alternatives --install /usr/bin/java java usr/local/java/bin/java 2

# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2

# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2

# alternatives --set java usr/local/java/bin/java

# alternatives --set javac usr/local/java/bin/javac

# alternatives --set jar usr/local/java/bin/jar

Now use the **java -version** command from the terminal as explained above.

### **Setting the Path**

Set the path of Cassandra path in “/.bashrc” as shown below.

[hadoop@linux ~]$ gedit ~/.bashrc

export CASSANDRA\_HOME = ~/cassandra

export PATH = $PATH:$CASSANDRA\_HOME/bin

## **Download Cassandra**

Apache Cassandra is available at [Download Link](http://cassandra.apache.org/download/) Cassandra using the following command.

$ wget http://supergsego.com/apache/cassandra/2.1.2/apache-cassandra-2.1.2-bin.tar.gz

Unzip Cassandra using the command **zxvf** as shown below.

$ tar zxvf apache-cassandra-2.1.2-bin.tar.gz.

Create a new directory named cassandra and move the contents of the downloaded file to it as shown below.

$ mkdir Cassandra

$ mv apache-cassandra-2.1.2/\* cassandra.

## **Configure Cassandra**

Open the **cassandra.yaml:** file, which will be available in the **bin** directory of Cassandra.

$ gedit cassandra.yaml

**Note** − If you have installed Cassandra from a deb or rpm package, the configuration files will be located in **/etc/cassandra** directory of Cassandra.

The above command opens the **cassandra.yaml** file. Verify the following configurations. By default, these values will be set to the specified directories.

* data\_file\_directories **“/var/lib/cassandra/data”**
* commitlog\_directory **“/var/lib/cassandra/commitlog”**
* saved\_caches\_directory **“/var/lib/cassandra/saved\_caches”**

Make sure these directories exist and can be written to, as shown below.

### **Create Directories**

As super-user, create the two directories **/var/lib/cassandra** and **/var./log/cassandra** into which Cassandra writes its data.

[root@linux cassandra]# mkdir /var/lib/cassandra

[root@linux cassandra]# mkdir /var/log/cassandra

### **Give Permissions to Folders**

Give read-write permissions to the newly created folders as shown below.

[root@linux /]# chmod 777 /var/lib/cassandra

[root@linux /]# chmod 777 /var/log/cassandra

## **Start Cassandra**

To start Cassandra, open the terminal window, navigate to Cassandra home directory/home, where you unpacked Cassandra, and run the following command to start your Cassandra server.

$ cd $CASSANDRA\_HOME

$./bin/cassandra -f

Using the –f option tells Cassandra to stay in the foreground instead of running as a background process. If everything goes fine, you can see the Cassandra server starting.

## **Programming Environment**

To set up Cassandra programmatically, download the following jar files −

* slf4j-api-1.7.5.jar
* cassandra-driver-core-2.0.2.jar
* guava-16.0.1.jar
* metrics-core-3.0.2.jar
* netty-3.9.0.Final.jar

Place them in a separate folder. For example, we are downloading these jars to a folder named **“Cassandra\_jars”**.

Set the classpath for this folder in **“.bashrc”**file as shown below.

[hadoop@linux ~]$ gedit ~/.bashrc

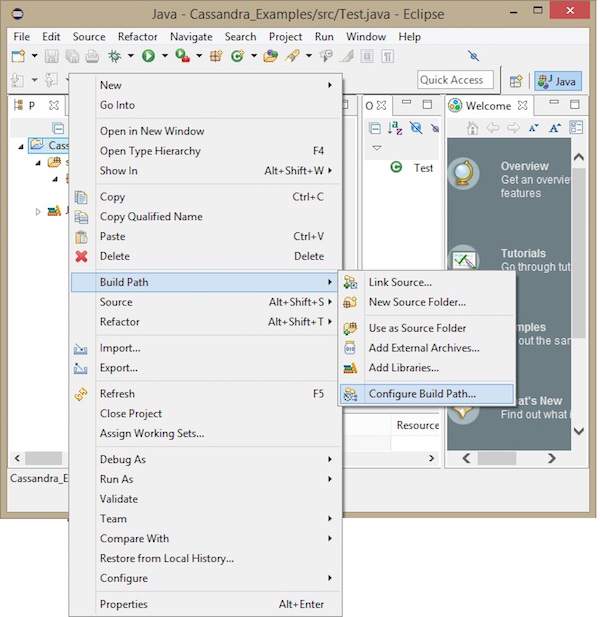
//Set the following class path in the .bashrc file.

export CLASSPATH = $CLASSPATH:/home/hadoop/Cassandra\_jars/\*

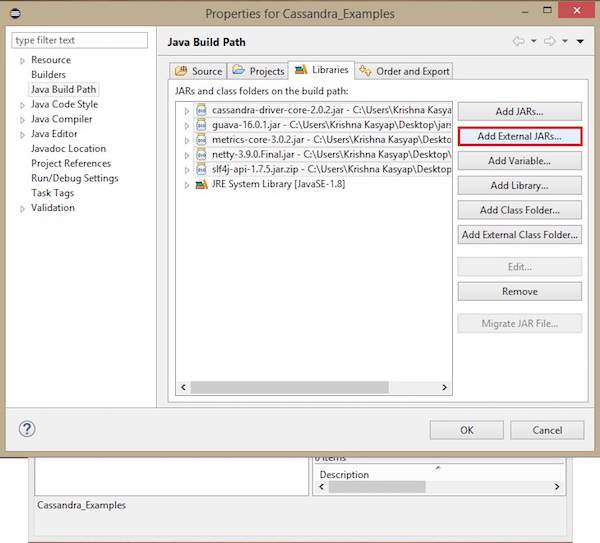
## **Eclipse Environment**

Open Eclipse and create a new project called Cassandra \_Examples.

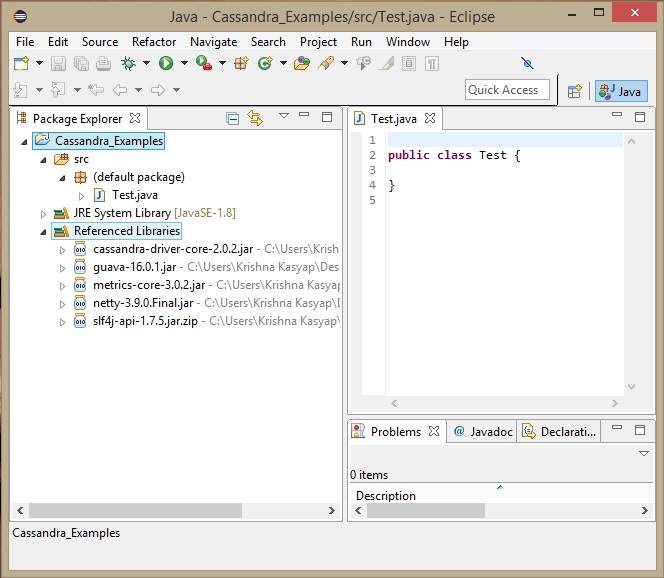
Right click on the project, select **Build Path→Configure Build Path** as shown below.



It will open the properties window. Under Libraries tab, select **Add External JARs**. Navigate to the directory where you saved your jar files. Select all the five jar files and click OK as shown below.



Under Referenced Libraries, you can see all the required jars added as shown below −



## **Maven Dependencies**

Given below is the pom.xml for building a Cassandra project using maven.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.5</version>

</dependency>

<dependency>

<groupId>com.datastax.cassandra</groupId>

<artifactId>cassandra-driver-core</artifactId>

<version>2.0.2</version>

</dependency>

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>16.0.1</version>

</dependency>

<dependency>

<groupId>com.codahale.metrics</groupId>

<artifactId>metrics-core</artifactId>

<version>3.0.2</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty</artifactId>

<version>3.9.0.Final</version>

</dependency>

</dependencies>

</project>

# Cassandra - Referenced Api

This chapter covers all the important classes in Cassandra.

## **Cluster**

This class is the main entry point of the driver. It belongs to **com.datastax.driver.core** package.

### **Methods**

|  |  |
| --- | --- |
| **S. No.** | **Methods and Description** |
| **1** | **Session connect()**  It creates a new session on the current cluster and initializes it. |
| **2** | **void close()**  It is used to close the cluster instance. |
| **3** | **static Cluster.Builder builder()**  It is used to create a new Cluster.Builder instance. |

## **Cluster.Builder**

This class is used to instantiate the **Cluster.Builder** class.

### **Methods**

|  |  |
| --- | --- |
| **S. No** | **Methods and Description** |
| **1** | **Cluster.Builder addContactPoint(String address)**  This method adds a contact point to cluster. |
| **2** | **Cluster build()**  This method builds the cluster with the given contact points. |

## **Session**

This interface holds the connections to Cassandra cluster. Using this interface, you can execute **CQL** queries. It belongs to **com.datastax.driver.core** package.

### **Methods**

|  |  |
| --- | --- |
| **S. No.** | **Methods and Description** |
| **1** | **void close()**  This method is used to close the current session instance. |
| **2** | **ResultSet execute(Statement statement)**  This method is used to execute a query. It requires a statement object. |
| **3** | **ResultSet execute(String query)**  This method is used to execute a query. It requires a query in the form of a String object. |
| **4** | **PreparedStatement prepare(RegularStatement statement)**  This method prepares the provided query. The query is to be provided in the form of a Statement. |
| **5** | **PreparedStatement prepare(String query)**  This method prepares the provided query. The query is to be provided in the form of a String. |

# Cassandra – Cqlsh

This chapter introduces the Cassandra query language shell and explains how to use its commands.

By default, Cassandra provides a prompt Cassandra query language shell **(cqlsh)** that allows users to communicate with it. Using this shell, you can execute **Cassandra Query Language (CQL)**.

Using cqlsh, you can

* define a schema,
* insert data, and
* execute a query.

## **Starting cqlsh**

Start cqlsh using the command **cqlsh** as shown below. It gives the Cassandra cqlsh prompt as output.

[hadoop@linux bin]$ cqlsh

Connected to Test Cluster at 127.0.0.1:9042.

[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 | Native protocol v3]

Use HELP for help.

cqlsh>

**Cqlsh** − As discussed above, this command is used to start the cqlsh prompt. In addition, it supports a few more options as well. The following table explains all the options of **cqlsh** and their usage.

|  |  |
| --- | --- |
| **Options** | **Usage** |
| cqlsh --help | Shows help topics about the options of **cqlsh** commands. |
| cqlsh --version | Provides the version of the cqlsh you are using. |
| cqlsh --color | Directs the shell to use colored output. |
| cqlsh --debug | Shows additional debugging information. |
| cqlsh --execute  cql\_statement | Directs the shell to accept and execute a CQL command. |
| cqlsh --file= **“file name”** | If you use this option, Cassandra executes the command in the given file and exits. |
| cqlsh --no-color | Directs Cassandra not to use colored output. |
| cqlsh -u **“user name”** | Using this option, you can authenticate a user. The default user name is: cassandra. |
| cqlsh-p **“pass word”** | Using this option, you can authenticate a user with a password. The default password is: cassandra. |

## **Cqlsh Commands**

Cqlsh has a few commands that allow users to interact with it. The commands are listed below.

## **Documented Shell Commands**

Given below are the Cqlsh documented shell commands. These are the commands used to perform tasks such as displaying help topics, exit from cqlsh, describe,etc.

* **HELP** − Displays help topics for all cqlsh commands.
* **CAPTURE** − Captures the output of a command and adds it to a file.
* **CONSISTENCY** − Shows the current consistency level, or sets a new consistency level.
* **COPY** − Copies data to and from Cassandra.
* **DESCRIBE** − Describes the current cluster of Cassandra and its objects.
* **EXPAND** − Expands the output of a query vertically.
* **EXIT** − Using this command, you can terminate cqlsh.
* **PAGING** − Enables or disables query paging.
* **SHOW** − Displays the details of current cqlsh session such as Cassandra version, host, or data type assumptions.
* **SOURCE** − Executes a file that contains CQL statements.
* **TRACING** − Enables or disables request tracing.

## **CQL Data Definition Commands**

* **CREATE KEYSPACE** − Creates a KeySpace in Cassandra.
* **USE** − Connects to a created KeySpace.
* **ALTER KEYSPACE** − Changes the properties of a KeySpace.
* **DROP KEYSPACE** − Removes a KeySpace
* **CREATE TABLE** − Creates a table in a KeySpace.
* **ALTER TABLE** − Modifies the column properties of a table.
* **DROP TABLE** − Removes a table.
* **TRUNCATE** − Removes all the data from a table.
* **CREATE INDEX** − Defines a new index on a single column of a table.
* **DROP INDEX** − Deletes a named index.

## **CQL Data Manipulation Commands**

* **INSERT** − Adds columns for a row in a table.
* **UPDATE** − Updates a column of a row.
* **DELETE** − Deletes data from a table.
* **BATCH** − Executes multiple DML statements at once.

## **CQL Clauses**

* **SELECT** − This clause reads data from a table
* **WHERE** − The where clause is used along with select to read a specific data.
* **ORDERBY** − The orderby clause is used along with select to read a specific data in a specific order.

# Cassandra - Shell Commands

Cassandra provides documented shell commands in addition to CQL commands. Given below are the Cassandra documented shell commands.

## **Help**

The HELP command displays a synopsis and a brief description of all cqlsh commands. Given below is the usage of help command.

cqlsh> help

Documented shell commands:

===========================

CAPTURE COPY DESCRIBE EXPAND PAGING SOURCE

CONSISTENCY DESC EXIT HELP SHOW TRACING.

CQL help topics:

================

ALTER CREATE\_TABLE\_OPTIONS SELECT

ALTER\_ADD CREATE\_TABLE\_TYPES SELECT\_COLUMNFAMILY

ALTER\_ALTER CREATE\_USER SELECT\_EXPR

ALTER\_DROP DELETE SELECT\_LIMIT

ALTER\_RENAME DELETE\_COLUMNS SELECT\_TABLE

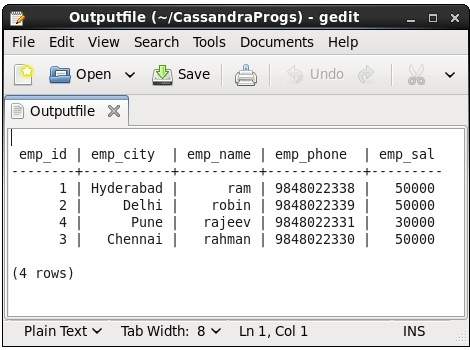
## **Capture**

This command captures the output of a command and adds it to a file. For example, take a look at the following code that captures the output to a file named **Outputfile**.

cqlsh> CAPTURE '/home/hadoop/CassandraProgs/Outputfile'

When we type any command in the terminal, the output will be captured by the file given. Given below is the command used and the snapshot of the output file.

cqlsh:tutorialspoint> select \* from emp;



You can turn capturing off using the following command.

cqlsh:tutorialspoint> capture off;

## **Consistency**

This command shows the current consistency level, or sets a new consistency level.

cqlsh:tutorialspoint> CONSISTENCY

Current consistency level is 1.

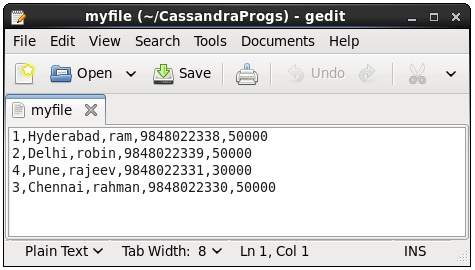
## **Copy**

This command copies data to and from Cassandra to a file. Given below is an example to copy the table named **emp** to the file **myfile**.

cqlsh:tutorialspoint> COPY emp (emp\_id, emp\_city, emp\_name, emp\_phone,emp\_sal) TO ‘myfile’;

4 rows exported in 0.034 seconds.

If you open and verify the file given, you can find the copied data as shown below.



## **Describe**

This command describes the current cluster of Cassandra and its objects. The variants of this command are explained below.

**Describe cluster** − This command provides information about the cluster.

cqlsh:tutorialspoint> describe cluster;

Cluster: Test Cluster

Partitioner: Murmur3Partitioner

Range ownership:

-658380912249644557 [127.0.0.1]

-2833890865268921414 [127.0.0.1]

-6792159006375935836 [127.0.0.1]

**Describe Keyspaces** − This command lists all the keyspaces in a cluster. Given below is the usage of this command.

cqlsh:tutorialspoint> describe keyspaces;

system\_traces system tp tutorialspoint

**Describe tables** − This command lists all the tables in a keyspace. Given below is the usage of this command.

cqlsh:tutorialspoint> describe tables;

emp

**Describe table** − This command provides the description of a table. Given below is the usage of this command.

cqlsh:tutorialspoint> describe table emp;

CREATE TABLE tutorialspoint.emp (

emp\_id int PRIMARY KEY,

emp\_city text,

emp\_name text,

emp\_phone varint,

emp\_sal varint

) WITH bloom\_filter\_fp\_chance = 0.01

AND caching = '{"keys":"ALL", "rows\_per\_partition":"NONE"}'

AND comment = ''

AND compaction = {'min\_threshold': '4', 'class':

'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',

'max\_threshold': '32'}

AND compression = {'sstable\_compression':

'org.apache.cassandra.io.compress.LZ4Compressor'}

AND dclocal\_read\_repair\_chance = 0.1

AND default\_time\_to\_live = 0

AND gc\_grace\_seconds = 864000

AND max\_index\_interval = 2048

AND memtable\_flush\_period\_in\_ms = 0

AND min\_index\_interval = 128

AND read\_repair\_chance = 0.0

AND speculative\_retry = '99.0PERCENTILE';

CREATE INDEX emp\_emp\_sal\_idx ON tutorialspoint.emp (emp\_sal);

## **Describe Type**

This command is used to describe a user-defined data type. Given below is the usage of this command.

cqlsh:tutorialspoint> describe type card\_details;

CREATE TYPE tutorialspoint.card\_details (

num int,

pin int,

name text,

cvv int,

phone set<int>,

mail text

);

## **Describe Types**

This command lists all the user-defined data types. Given below is the usage of this command. Assume there are two user-defined data types: **card** and **card\_details**.

cqlsh:tutorialspoint> DESCRIBE TYPES;

card\_details card

## **Expand**

This command is used to expand the output. Before using this command, you have to turn the expand command on. Given below is the usage of this command.

cqlsh:tutorialspoint> expand on;

cqlsh:tutorialspoint> select \* from emp;

@ Row 1

-----------+------------

emp\_id | 1

emp\_city | Hyderabad

emp\_name | ram

emp\_phone | 9848022338

emp\_sal | 50000

@ Row 2

-----------+------------

emp\_id | 2

emp\_city | Delhi

emp\_name | robin

emp\_phone | 9848022339

emp\_sal | 50000

@ Row 3

-----------+------------

emp\_id | 4

emp\_city | Pune

emp\_name | rajeev

emp\_phone | 9848022331

emp\_sal | 30000

@ Row 4

-----------+------------

emp\_id | 3

emp\_city | Chennai

emp\_name | rahman

emp\_phone | 9848022330

emp\_sal | 50000

(4 rows)

**Note** − You can turn the expand option off using the following command.

cqlsh:tutorialspoint> expand off;

Disabled Expanded output.

## **Exit**

This command is used to terminate the cql shell.

## **Show**

This command displays the details of current cqlsh session such as Cassandra version, host, or data type assumptions. Given below is the usage of this command.

cqlsh:tutorialspoint> show host;

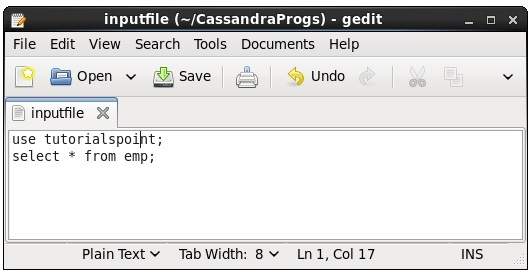
Connected to Test Cluster at 127.0.0.1:9042.

cqlsh:tutorialspoint> show version;

[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 | Native protocol v3]

## **Source**

Using this command, you can execute the commands in a file. Suppose our input file is as follows −



Then you can execute the file containing the commands as shown below.

cqlsh:tutorialspoint> source '/home/hadoop/CassandraProgs/inputfile';

emp\_id | emp\_city | emp\_name | emp\_phone | emp\_sal

--------+-----------+----------+------------+---------

1 | Hyderabad | ram | 9848022338 | 50000

2 | Delhi | robin | 9848022339 | 50000

3 | Pune | rajeev | 9848022331 | 30000

4 | Chennai | rahman | 9848022330 | 50000

(4 rows)

# Cassandra - Create Keyspace

## **Creating a Keyspace using Cqlsh**

A keyspace in Cassandra is a namespace that defines data replication on nodes. A cluster contains one keyspace per node. Given below is the syntax for creating a keyspace using the statement **CREATE KEYSPACE**.

### **Syntax**

CREATE KEYSPACE <identifier> WITH <properties>

i.e.

CREATE KEYSPACE “KeySpace Name”

WITH replication = {'class': ‘Strategy name’, 'replication\_factor' : ‘No.Of replicas’};

CREATE KEYSPACE “KeySpace Name”

WITH replication = {'class': ‘Strategy name’, 'replication\_factor' : ‘No.Of replicas’}

AND durable\_writes = ‘Boolean value’;

The CREATE KEYSPACE statement has two properties: **replication** and **durable\_writes**.

## **Replication**

The replication option is to specify the **Replica Placement strategy** and the number of replicas wanted. The following table lists all the replica placement strategies.

|  |  |
| --- | --- |
| **Strategy name** | **Description** |
| **Simple Strategy'** | Specifies a simple replication factor for the cluster. |
| **Network Topology Strategy** | Using this option, you can set the replication factor for each data-center independently. |
| **Old Network Topology Strategy** | This is a legacy replication strategy. |

Using this option, you can instruct Cassandra whether to use **commitlog** for updates on the current KeySpace. This option is not mandatory and by default, it is set to true.

### **Example**

Given below is an example of creating a KeySpace.

* Here we are creating a KeySpace named **TutorialsPoint**.
* We are using the first replica placement strategy, i.e.., **Simple Strategy**.
* And we are choosing the replication factor to **1 replica**.

cqlsh.> CREATE KEYSPACE tutorialspoint

WITH replication = {'class':'SimpleStrategy', 'replication\_factor' : 3};

### **Verification**

You can verify whether the table is created or not using the command **Describe**. If you use this command over keyspaces, it will display all the keyspaces created as shown below.

cqlsh> DESCRIBE keyspaces;

tutorialspoint system system\_traces

Here you can observe the newly created KeySpace **tutorialspoint.**

## **Durable\_writes**

By default, the durable\_writes properties of a table is set to **true,** however it can be set to false. You cannot set this property to **simplex strategy.**

### **Example**

Given below is the example demonstrating the usage of durable writes property.

cqlsh> CREATE KEYSPACE test

... WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 }

... AND DURABLE\_WRITES = false;

### **Verification**

You can verify whether the durable\_writes property of test KeySpace was set to false by querying the System Keyspace. This query gives you all the KeySpaces along with their properties.

cqlsh> SELECT \* FROM system\_schema.keyspaces;

keyspace\_name | durable\_writes | strategy\_class | strategy\_options

----------------+----------------+------------------------------------------------------+----------------------------

test | False | org.apache.cassandra.locator.NetworkTopologyStrategy | {"datacenter1" : "3"}

tutorialspoint | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor" : "4"}

system | True | org.apache.cassandra.locator.LocalStrategy | { }

system\_traces | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor" : "2"}

(4 rows)

Here you can observe the durable\_writes property of test KeySpace was set to false.

## **Using a Keyspace**

You can use a created KeySpace using the keyword **USE**. Its syntax is as follows −

Syntax:USE <identifier>

### **Example**

In the following example, we are using the KeySpace **tutorialspoint.**

cqlsh> USE tutorialspoint;

cqlsh:tutorialspoint>

## **Creating a Keyspace using Java API**

You can create a Keyspace using the **execute()** method of **Session** class. Follow the steps given below to create a keyspace using Java API.

### **Step1: Create a Cluster Object**

First of all, 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 **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 in 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, 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 ” );

### **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 this example, we are creating a KeySpace named **tp.** We are using the first replica placement strategy, i.e., Simple Strategy, and we are choosing the replication factor to 1 replica.

You have to store the query in a string variable and pass it to the execute() method as shown below.

String query = "CREATE KEYSPACE tp WITH replication "

+ "= {'class':'SimpleStrategy', 'replication\_factor':1}; ";

session.execute(query);

### **Step4 : Use the KeySpace**

You can use a created KeySpace using the execute() method as shown below.

execute(“ USE tp ” );

Given below is the complete program to create and use a keyspace in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Create\_KeySpace {

public static void main(String args[]){

//Query

String query = "CREATE KEYSPACE tp WITH replication "

+ "= {'class':'SimpleStrategy', 'replication\_factor':1};";

//creating Cluster object

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

//Creating Session object

Session session = cluster.connect();

//Executing the query

session.execute(query);

//using the KeySpace

session.execute("USE tp");

System.out.println("Keyspace 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\_KeySpace.java

$java Create\_KeySpace

Under normal conditions, it will produce the following output −

Keyspace created

# Cassandra - Alter Keyspace

## **Altering a KeySpace**

ALTER KEYSPACE can be used to alter properties such as the number of replicas and the durable\_writes of a KeySpace. Given below is the syntax of this command.

### **Syntax**

ALTER KEYSPACE <identifier> WITH <properties>

i.e.

ALTER KEYSPACE “KeySpace Name”

WITH replication = {'class': ‘Strategy name’, 'replication\_factor' : ‘No.Of replicas’};

The properties of **ALTER KEYSPACE** are same as CREATE KEYSPACE. It has two properties: **replication** and **durable\_writes**.

## **Replication**

The replication option specifies the replica placement strategy and the number of replicas wanted.

## **Durable\_writes**

Using this option, you can instruct Cassandra whether to use commitlog for updates on the current KeySpace. This option is not mandatory and by default, it is set to true.

### **Example**

Given below is an example of altering a KeySpace.

* Here we are altering a KeySpace named **TutorialsPoint**.
* We are changing the replication factor from 1 to 3.

cqlsh.> ALTER KEYSPACE tutorialspoint

WITH replication = {'class':'NetworkTopologyStrategy', 'replication\_factor' : 3};

### **Altering Durable\_writes**

You can also alter the durable\_writes property of a KeySpace. Given below is the durable\_writes property of the **test** KeySpace.

SELECT \* FROM system.schema\_keyspaces;

keyspace\_name | durable\_writes | strategy\_class | strategy\_options

----------------+----------------+------------------------------------------------------+----------------------------

test | False | org.apache.cassandra.locator.NetworkTopologyStrategy | {"datacenter1":"3"}

tutorialspoint | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor":"4"}

system | True | org.apache.cassandra.locator.LocalStrategy | { }

system\_traces | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor":"2"}

(4 rows)

ALTER KEYSPACE test

WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3}

AND DURABLE\_WRITES = true;

Once again, if you verify the properties of KeySpaces, it will produce the following output.

SELECT \* FROM system.schema\_keyspaces;

keyspace\_name | durable\_writes | strategy\_class | strategy\_options

----------------+----------------+------------------------------------------------------+----------------------------

test | True | org.apache.cassandra.locator.NetworkTopologyStrategy | {"datacenter1":"3"}

tutorialspoint | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor":"4"}

system | True | org.apache.cassandra.locator.LocalStrategy | { }

system\_traces | True | org.apache.cassandra.locator.SimpleStrategy | {"replication\_factor":"2"}

(4 rows)

## **Altering a Keyspace using Java API**

You can alter a keyspace using the **execute()** method of **Session** class. Follow the steps given below to alter a keyspace using Java API

### **Step1: Create a Cluster Object**

First of all, 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 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, 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 ” );

### **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 this example,

* We are altering a keyspace named **tp**. We are altering the replication option from Simple Strategy to Network Topology Strategy.
* We are altering the **durable\_writes** to false

You have to store the query in a string variable and pass it to the execute() method as shown below.

//Query

String query = "ALTER KEYSPACE tp WITH replication " + "= {'class':'NetworkTopologyStrategy', 'datacenter1':3}" +" AND DURABLE\_WRITES = false;";

session.execute(query);

Given below is the complete program to create and use a keyspace in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Alter\_KeySpace {

public static void main(String args[]){

//Query

String query = "ALTER KEYSPACE tp WITH replication " + "= {'class':'NetworkTopologyStrategy', 'datacenter1':3}"

+ "AND DURABLE\_WRITES = false;";

//Creating Cluster object

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

//Creating Session object

Session session = cluster.connect();

//Executing the query

session.execute(query);

System.out.println("Keyspace altered");

}

}

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 Alter\_KeySpace.java

$java Alter\_KeySpace

Under normal conditions, it produces the following output −

Keyspace Altered

# Cassandra - Drop Keyspace

## **Dropping a Keyspace**

You can drop a KeySpace using the command **DROP KEYSPACE**. Given below is the syntax for dropping a KeySpace.

### **Syntax**

DROP KEYSPACE <identifier>

i.e.

DROP KEYSPACE “KeySpace name”

### **Example**

The following code deletes the keyspace **tutorialspoint**.

cqlsh> DROP KEYSPACE tutorialspoint;

### **Verification**

Verify the keyspaces using the command **Describe** and check whether the table is dropped as shown below.

cqlsh> DESCRIBE keyspaces;

system system\_traces

Since we have deleted the keyspace tutorialspoint, you will not find it in the keyspaces list.

## **Dropping a Keyspace using Java API**

You can create a keyspace using the execute() method of Session class. Follow the steps given below to drop a keyspace using Java API.

### **Step1: Create a Cluster Object**

First of all, 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, 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”);

### **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 deleting a keyspace named **tp**. You have to store the query in a string variable and pass it to the execute() method as shown below.

String query = "DROP KEYSPACE tp; ";

session.execute(query);

Given below is the complete program to create and use a keyspace in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Drop\_KeySpace {

public static void main(String args[]){

//Query

String query = "Drop KEYSPACE tp";

//creating Cluster object

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

//Creating Session object

Session session = cluster.connect();

//Executing the query

session.execute(query);

System.out.println("Keyspace deleted");

}

}

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 Delete\_KeySpace.java

$java Delete\_KeySpace

Under normal conditions, it should produce the following output −

Keyspace deleted

# Cassandra - Create Table

## **Creating a Table**

You can create a table using the command **CREATE TABLE**. Given below is the syntax for creating a table.

### **Syntax**

CREATE (TABLE | COLUMNFAMILY) <tablename>

('<column-definition>' , '<column-definition>')

(WITH <option> AND <option>)

## **Defining a Column**

You can define a column as shown below.

column name1 data type,

column name2 data type,

example:

age int,

name text

## **Primary Key**

The primary key is a column that is used to uniquely identify a row. Therefore,defining a primary key is mandatory while creating a table. A primary key is made of one or more columns of a table. You can define a primary key of a table as shown below.

CREATE TABLE tablename(

column1 name datatype PRIMARYKEY,

column2 name data type,

column3 name data type.

)

**or**

CREATE TABLE tablename(

column1 name datatype PRIMARYKEY,

column2 name data type,

column3 name data type,

**PRIMARY KEY (column1)**

)

### **Example**

Given below is an example to create a table in Cassandra using cqlsh. Here we are −

* Using the keyspace tutorialspoint
* Creating a table named **emp**

It will have details such as employee name, id, city, salary, and phone number. Employee id is the primary key.

cqlsh> USE tutorialspoint;

cqlsh:tutorialspoint>; CREATE TABLE emp(

emp\_id int PRIMARY KEY,

emp\_name text,

emp\_city text,

emp\_sal varint,

emp\_phone varint

);

### **Verification**

The select statement will give you the schema. Verify the table using the select statement as shown below.

cqlsh:tutorialspoint> select \* from emp;

emp\_id | emp\_city | emp\_name | emp\_phone | emp\_sal

--------+----------+----------+-----------+---------

(0 rows)

Here you can observe the table created with the given columns. Since we have deleted the keyspace tutorialspoint, you will not find it in the keyspaces list.

## **Creating a Table using Java API**

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

### **Step1: Create a Cluster Object**

First of all, create an instance of the **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, 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 creating a table named **emp.** You have to store the query in a string variable and pass it to the execute() method as shown below.

//Query

String query = "CREATE TABLE emp(emp\_id int PRIMARY KEY, "

+ "emp\_name text, "

+ "emp\_city text, "

+ "emp\_sal varint, "

+ "emp\_phone varint );";

session.execute(query);

Given below is the complete program to create and use a keyspace in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Create\_Table {

public static void main(String args[]){

//Query

String query = "CREATE TABLE emp(emp\_id int PRIMARY KEY, "

+ "emp\_name text, "

+ "emp\_city text, "

+ "emp\_sal varint, "

+ "emp\_phone varint );";

//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("Table 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\_Table.java

$java Create\_Table

Under normal conditions, it should produce the following output −

Table created

# Cassandra - Alter Table

## **Altering a Table**

You can alter a table using the command **ALTER TABLE**. Given below is the syntax for creating a table.

### **Syntax**

ALTER (TABLE | COLUMNFAMILY) <tablename> <instruction>

Using ALTER command, you can perform the following operations −

* Add a column
* Drop a column

## **Adding a Column**

Using ALTER command, you can add a column to a table. While adding columns, you have to take care that the column name is not conflicting with the existing column names and that the table is not defined with compact storage option. Given below is the syntax to add a column to a table.

ALTER TABLE table name

ADD new column datatype;

**Example**

Given below is an example to add a column to an existing table. Here we are adding a column called **emp\_email** of text datatype to the table named **emp**.

cqlsh:tutorialspoint> ALTER TABLE emp

... ADD emp\_email text;

**Verification**

Use the SELECT statement to verify whether the column is added or not. Here you can observe the newly added column emp\_email.

cqlsh:tutorialspoint> select \* from emp;

emp\_id | emp\_city | emp\_email | emp\_name | emp\_phone | emp\_sal

--------+----------+-----------+----------+-----------+---------

## **Dropping a Column**

Using ALTER command, you can delete a column from a table. Before dropping a column from a table, check that the table is not defined with compact storage option. Given below is the syntax to delete a column from a table using ALTER command.

ALTER table name

DROP column name;

**Example**

Given below is an example to drop a column from a table. Here we are deleting the column named **emp\_email**.

cqlsh:tutorialspoint> ALTER TABLE emp DROP emp\_email;

**Verification**

Verify whether the column is deleted using the **select** statement, as shown below.

cqlsh:tutorialspoint> select \* from emp;

emp\_id | emp\_city | emp\_name | emp\_phone | emp\_sal

--------+----------+----------+-----------+---------

(0 rows)

Since **emp\_email** column has been deleted, you cannot find it anymore.

## **Altering a Table using Java API**

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

### **Step1: Create a Cluster Object**

First of all, 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, 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 ” );

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

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

### **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 adding a column to a table named **emp.** To do so, you have to store the query in a string variable and pass it to the execute() method as shown below.

//Query

String query1 = "ALTER TABLE emp ADD emp\_email text";

session.execute(query);

Given below is the complete program to add a column to an existing table.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Add\_column {

public static void main(String args[]){

//Query

String query = "ALTER TABLE emp ADD emp\_email text";

//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("Column added");

}

}

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 Add\_Column.java

$java Add\_Column

Under normal conditions, it should produce the following output −

Column added

## **Deleting a Column**

Given below is the complete program to delete a column from an existing table.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Delete\_Column {

public static void main(String args[]){

//Query

String query = "ALTER TABLE emp DROP emp\_email;";

//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("Column deleted");

}

}

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 Delete\_Column.java

$java Delete\_Column

Under normal conditions, it should produce the following output −

Column deleted

# Cassandra - Drop Table

## **Dropping a Table**

You can drop a table using the command **Drop Table**. Its syntax is as follows −

### **Syntax**

DROP TABLE <tablename>

### **Example**

The following code drops an existing table from a KeySpace.

cqlsh:tutorialspoint> DROP TABLE emp;

### **Verification**

Use the Describe command to verify whether the table is deleted or not. Since the emp table has been deleted, you will not find it in the column families list.

cqlsh:tutorialspoint> DESCRIBE COLUMNFAMILIES;

employee

## **Deleting a Table using Java API**

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

### **Step1: Create a Cluster Object**

First of all, 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 **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, 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 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 deleting a table named **emp**. You have to store the query in a string variable and pass it to the execute() method as shown below.

// Query

String query = "DROP TABLE emp1;”;

session.execute(query);

Given below is the complete program to drop a table in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Drop\_Table {

public static void main(String args[]){

//Query

String query = "DROP TABLE emp1;";

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("Table dropped");

}

}

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 Drop\_Table.java

$java Drop\_Table

Under normal conditions, it should produce the output : Table dropped

# Cassandra - Truncate Table

## **Truncating a Table**

You can truncate a table using the TRUNCATE command. When you truncate a table, all the rows of the table are deleted permanently. Given below is the syntax of this command.

### **Syntax**

TRUNCATE <tablename>

### **Example**

Let us assume there is a table called **student** with the following data.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **s\_name** | **s\_branch** | **s\_aggregate** |
| 1 | ram | IT | 70 |
| 2 | rahman | EEE | 75 |
| 3 | robbin | Mech | 72 |

When you execute the select statement to get the table **student,** it will give you the following output.

cqlsh:tp> select \* from student;

s\_id | s\_aggregate | s\_branch | s\_name

------+-------------+----------+--------

1 | 70 | IT | ram

2 | 75 | EEE | rahman

3 | 72 | MECH | robbin

(3 rows)

Now truncate the table using the TRUNCATE command.

cqlsh:tp> TRUNCATE student;

### **Verification**

Verify whether the table is truncated by executing the **select** statement. Given below is the output of the select statement on the student table after truncating.

cqlsh:tp> select \* from student;

s\_id | s\_aggregate | s\_branch | s\_name

------+-------------+----------+--------

(0 rows)

## **Truncating a Table using Java API**

You can truncate a table using the execute() method of Session class. Follow the steps given below to truncate a table.

### **Step1: Create a Cluster Object**

First of all, 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: Creating 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 ” );

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

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

### **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 truncating a table named **emp**.

//Query

String query = "TRUNCATE emp;;”;

session.execute(query);

Given below is the complete program to truncate a table in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Truncate\_Table {

public static void main(String args[]){

//Query

String query = "Truncate student;";

//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("Table truncated");

}

}

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 Truncate\_Table.java

$java Truncate\_Table

Under normal conditions, it should produce the following output −

Table truncated

# Cassandra - Create Index

## **Creating an Index using Cqlsh**

You can create an index in Cassandra using the command **CREATE INDEX.** Its syntax is as follows −

CREATE INDEX <identifier> ON <tablename>

Given below is an example to create an index to a column. Here we are creating an index to a column ‘emp\_name’ in a table named emp.

cqlsh:tutorialspoint> CREATE INDEX name ON emp1 (emp\_name);

## **Creating an Index using Java API**

You can create an index to a column of a table using the execute() method of Session class. Follow the steps given below to create an index to a column in a table.

### **Step1: Create a Cluster Object**

First of all, 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 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 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 creating an index to a column called emp\_name, in a table named **emp.** You have to store the query in a string variable and pass it to the execute() method as shown below.

//Query

String query = "CREATE INDEX name ON emp1 (emp\_name);";

session.execute(query);

Given below is the complete program to create an index of a column in a table in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Create\_Index {

public static void main(String args[]){

//Query

String query = "CREATE INDEX name ON emp1 (emp\_name);";

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("Index 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\_Index.java

$java Create\_Index

Under normal conditions, it should produce the following output −

Index created

# Cassandra - Drop Index

## **Dropping an Index**

You can drop an index using the command **DROP INDEX**. Its syntax is as follows −

DROP INDEX <identifier>

Given below is an example to drop an index of a column in a table. Here we are dropping the index of the column name in the table emp.

cqlsh:tp> drop index name;

## **Dropping an Index using Java API**

You can drop an index of a table using the execute() method of Session class. Follow the steps given below to drop an index from a table.

### **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 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 dropping an index “name” of **emp** table. You have to store the query in a string variable and pass it to the execute() method as shown below.

//Query

String query = "DROP INDEX user\_name;";

session.execute(query);

Given below is the complete program to drop an index in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Drop\_Index {

public static void main(String args[]){

//Query

String query = "DROP INDEX user\_name;";

//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("Index dropped");

}

}

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 Drop\_index.java

$java Drop\_index

Under normal conditions, it should produce the following output −

Index dropped

# Cassandra - Batch Statements

## **Using Batch Statements**

Using **BATCH,** you can execute multiple modification statements (insert, update, delete) simultaneiously. Its syntax is as follows −

BEGIN BATCH

<insert-stmt>/ <update-stmt>/ <delete-stmt>

APPLY BATCH

### **Example**

Assume there is a table in Cassandra called emp having the following data −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_city** | **emp\_phone** | **emp\_sal** |
| 1 | ram | Hyderabad | 9848022338 | 50000 |
| 2 | robin | Delhi | 9848022339 | 50000 |
| 3 | rahman | Chennai | 9848022330 | 45000 |

In this example, we will perform the following operations −

* Insert a new row with the following details (4, rajeev, pune, 9848022331, 30000).
* Update the salary of employee with row id 3 to 50000.
* Delete city of the employee with row id 2.

To perform the above operations in one go, use the following BATCH command −

cqlsh:tutorialspoint> BEGIN BATCH

... INSERT INTO emp (emp\_id, emp\_city, emp\_name, emp\_phone, emp\_sal) values( 4,'Pune','rajeev',9848022331, 30000);

... UPDATE emp SET emp\_sal = 50000 WHERE emp\_id =3;

... DELETE emp\_city FROM emp WHERE emp\_id = 2;

... APPLY BATCH;

### **Verification**

After making changes, verify the table using the SELECT statement. It should 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 | null | robin | 9848022339 | 50000

3 | Chennai | rahman | 9848022330 | 50000

4 | Pune | rajeev | 9848022331 | 30000

(4 rows)

Here you can observe the table with modified data.

## **Batch Statements using Java API**

Batch statements can be written programmatically in a table using the execute() method of Session class. Follow the steps given below to execute multiple statements using batch statement with the help of 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 this example, we will perform the following operations −

* Insert a new row with the following details (4, rajeev, pune, 9848022331, 30000).
* Update the salary of employee with row id 3 to 50000.
* Delete the city of the employee with row id 2.

You have to store the query in a string variable and pass it to the execute() method as shown below.

String query1 = ” BEGIN BATCH INSERT INTO emp (emp\_id, emp\_city, emp\_name, emp\_phone, emp\_sal) values( 4,'Pune','rajeev',9848022331, 30000);

UPDATE emp SET emp\_sal = 50000 WHERE emp\_id =3;

DELETE emp\_city FROM emp WHERE emp\_id = 2;

APPLY BATCH;”;

Given below is the complete program to execute multiple statements simultaneously on a table in Cassandra using Java API.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Batch {

public static void main(String args[]){

//query

String query =" BEGIN BATCH INSERT INTO emp (emp\_id, emp\_city,

emp\_name, emp\_phone, emp\_sal) values( 4,'Pune','rajeev',9848022331, 30000);"

+ "UPDATE emp SET emp\_sal = 50000 WHERE emp\_id =3;"

+ "DELETE emp\_city FROM emp WHERE emp\_id = 2;"

+ "APPLY BATCH;";

//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("Changes done");

}

}

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 Batch.java

$java Batch

Under normal conditions, it should produce the following output −

Changes done

# Cassandra - Create Data

## **Creating Data in a Table**

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 −

Data created

# Cassandra - Update Data

## **Updating Data in a Table**

**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

# Cassandra - Read Data

## **Reading Data using Select Clause**

SELECT clause is used to read data from a table in Cassandra. Using this clause, you can read a whole table, a single column, or a particular cell. Given below is the syntax of SELECT clause.

SELECT FROM <tablename>

### **Example**

Assume there is a table in the keyspace named **emp** with the following details −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **emp\_id** | **emp\_name** | **emp\_city** | **emp\_phone** | **emp\_sal** |
| 1 | ram | Hyderabad | 9848022338 | 50000 |
| 2 | robin | null | 9848022339 | 50000 |
| 3 | rahman | Chennai | 9848022330 | 50000 |
| 4 | rajeev | Pune | 9848022331 | 30000 |

The following example shows how to read a whole table using SELECT clause. Here we are reading a table called **emp**.

cqlsh:tutorialspoint> select \* from emp;

emp\_id | emp\_city | emp\_name | emp\_phone | emp\_sal

--------+-----------+----------+------------+---------

1 | Hyderabad | ram | 9848022338 | 50000

2 | null | robin | 9848022339 | 50000

3 | Chennai | rahman | 9848022330 | 50000

4 | Pune | rajeev | 9848022331 | 30000

(4 rows)

## **Reading Required Columns**

The following example shows how to read a particular column in a table.

cqlsh:tutorialspoint> SELECT emp\_name, emp\_sal from emp;

emp\_name | emp\_sal

----------+---------

ram | 50000

robin | 50000

rajeev | 30000

rahman | 50000

## **Where Clause**

Using WHERE clause, you can put a constraint on the required columns. Its syntax is as follows −

SELECT FROM <table name> WHERE <condition>;

**Note** − A WHERE clause can be used only on the columns that are a part of primary key or have a secondary index on them.

In the following example, we are reading the details of an employee whose salary is 50000. First of all, set secondary index to the column emp\_sal.

cqlsh:tutorialspoint> CREATE INDEX ON emp(emp\_sal);

cqlsh:tutorialspoint> SELECT \* FROM emp WHERE emp\_sal=50000;

emp\_id | emp\_city | emp\_name | emp\_phone | emp\_sal

--------+-----------+----------+------------+---------

1 | Hyderabad | ram | 9848022338 | 50000

2 | null | robin | 9848022339 | 50000

3 | Chennai | rahman | 9848022330 | 50000

## **Reading Data using Java API**

You can read data from a table using the execute() method of Session class. Follow the steps given below to execute multiple statements using batch statement with the help of 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 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 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 this example, we are retrieving the data from **emp** table. Store the query in a string and pass it to the execute() method of session class as shown below.

String query = ”SELECT 8 FROM emp”;

session.execute(query);

Execute the query using the execute() method of Session class.

### **Step 4: Get the ResultSet Object**

The select queries will return the result in the form of a **ResultSet** object, therefore store the result in the object of **RESULTSET** class as shown below.

ResultSet result = session.execute( );

Given below is the complete program to read data from a table.

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.ResultSet;

import com.datastax.driver.core.Session;

public class Read\_Data {

public static void main(String args[])throws Exception{

//queries

String query = "SELECT \* FROM emp";

//Creating Cluster object

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

//Creating Session object

Session session = cluster.connect("tutorialspoint");

//Getting the ResultSet

ResultSet result = session.execute(query);

System.out.println(result.all());

}

}

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 Read\_Data.java

$java Read\_Data

Under normal conditions, it should produce the following output −

[Row[1, Hyderabad, ram, 9848022338, 50000], Row[2, Delhi, robin,

9848022339, 50000], Row[4, Pune, rajeev, 9848022331, 30000], Row[3,

Chennai, rahman, 9848022330, 50000]]

# Cassandra - Delete Data

## **Deleting Datafrom a Table**

You can delete data from a table using the command **DELETE**. Its syntax is as follows −

DELETE FROM <identifier> WHERE <condition>;

### **Example**

Let us assume there is a table in Cassandra called **emp** having the following data −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

The following statement deletes the emp\_sal column of last row −

cqlsh:tutorialspoint> DELETE emp\_sal FROM emp WHERE emp\_id=3;

### **Verification**

Use SELECT statement to verify whether the data has been deleted or not. If you verify the emp table using SELECT, 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 | null

(3 rows)

Since we have deleted the salary of Rahman, you will observe a null value in place of salary.

## **Deleting an Entire Row**

The following command deletes an entire row from a table.

cqlsh:tutorialspoint> DELETE FROM emp WHERE emp\_id=3;

### **Verification**

Use SELECT statement to verify whether the data has been deleted or not. If you verify the emp table using SELECT, 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

(2 rows)

Since we have deleted the last row, there are only two rows left in the table.

## **Deleting Data using Java API**

You can delete data in a table using the execute() method of Session class. Follow the steps given below to delete data from 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 a 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 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 deleting data from a table named **emp.** You have to store the query in a string variable and pass it to the **execute()** method as shown below.

String query1 = ”DELETE FROM emp WHERE emp\_id=3; ”;

session.execute(query);

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

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.Session;

public class Delete\_Data {

public static void main(String args[]){

//query

String query = "DELETE FROM emp WHERE emp\_id=3;";

//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 deleted");

}

}

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 Delete\_Data.java

$java Delete\_Data

Under normal conditions, it should produce the following output −

Data deleted

# Cassandra - CQL Datatypes

CQL provides a rich set of built-in data types, including collection types. Along with these data types, users can also create their own custom data types. The following table provides a list of built-in data types available in CQL.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Constants** | **Description** |
| ascii | strings | Represents ASCII character string |
| bigint | bigint | Represents 64-bit signed long |
| **blob** | blobs | Represents arbitrary bytes |
| Boolean | booleans | Represents true or false |
| **counter** | integers | Represents counter column |
| decimal | integers, floats | Represents variable-precision decimal |
| double | integers | Represents 64-bit IEEE-754 floating point |
| float | integers, floats | Represents 32-bit IEEE-754 floating point |
| inet | strings | Represents an IP address, IPv4 or IPv6 |
| int | integers | Represents 32-bit signed int |
| text | strings | Represents UTF8 encoded string |
| **timestamp** | integers, strings | Represents a timestamp |
| **timeuuid** | uuids | Represents type 1 UUID |
| **uuid** | uuids | Represents type 1 or type 4 |
|  |  | UUID |
| varchar | strings | Represents uTF8 encoded string |
| varint | integers | Represents arbitrary-precision integer |

## **Collection Types**

Cassandra Query Language also provides a collection data types. The following table provides a list of Collections available in CQL.

|  |  |
| --- | --- |
| **Collection** | **Description** |
| list | A list is a collection of one or more ordered elements. |
| map | A map is a collection of key-value pairs. |
| set | A set is a collection of one or more elements. |

## **User-defined datatypes**

Cqlsh provides users a facility of creating their own data types. Given below are the commands used while dealing with user defined datatypes.

* **CREATE TYPE** − Creates a user-defined datatype.
* **ALTER TYPE** − Modifies a user-defined datatype.
* **DROP TYPE** − Drops a user-defined datatype.
* **DESCRIBE TYPE** − Describes a user-defined datatype.
* **DESCRIBE TYPES** − Describes user-defined datatypes.

# Cassandra - CQL Collections

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)

# Cassandra - CQL User Defined Datatypes

CQL provides the facility of creating and using user-defined data types. You can create a data type to handle multiple fields. This chapter explains how to create, alter, and delete a user-defined data type.

## **Creating a User-defined Data Type**

The command **CREATE TYPE** is used to create a user-defined data type. Its syntax is as follows −

CREATE TYPE <keyspace name>. <data typename>

( variable1, variable2).

### **Example**

Given below is an example for creating a user-defined data type. In this example, we are creating a **card\_details** data type containing the following details.

|  |  |  |
| --- | --- | --- |
| **Field** | **Field name** | **Data type** |
| credit card no | num | int |
| credit card pin | pin | int |
| name on credit card | name | text |
| cvv | cvv | int |
| Contact details of card holder | phone | set |

cqlsh:tutorialspoint> CREATE TYPE card\_details (

... num int,

... pin int,

... name text,

... cvv int,

... phone set<int>

... );

**Note** − The name used for user-defined data type should not coincide with reserved type names.

### **Verification**

Use the **DESCRIBE** command to verify whether the type created has been created or not.

CREATE TYPE tutorialspoint.card\_details (

num int,

pin int,

name text,

cvv int,

phone set<int>

);

## **Altering a User-defined Data Type**

**ALTER TYPE** − command is used to alter an existing data type. Using ALTER, you can add a new field or rename an existing field.

### **Adding a Field to a Type**

Use the following syntax to add a new field to an existing user-defined data type.

ALTER TYPE typename

ADD field\_name field\_type;

The following code adds a new field to the **Card\_details** data type. Here we are adding a new field called email.

cqlsh:tutorialspoint> ALTER TYPE card\_details ADD email text;

### **Verification**

Use the **DESCRIBE** command to verify whether the new field is added or not.

cqlsh:tutorialspoint> describe type card\_details;

CREATE TYPE tutorialspoint.card\_details (

num int,

pin int,

name text,

cvv int,

phone set<int>,

);

### **Renaming a Field in a Type**

Use the following syntax to rename an existing user-defined data type.

ALTER TYPE typename

RENAME existing\_name TO new\_name;

The following code changes the name of the field in a type. Here we are renaming the field email to mail.

cqlsh:tutorialspoint> ALTER TYPE card\_details RENAME email TO mail;

### **Verification**

Use the **DESCRIBE** command to verify whether the type name changed or not.

cqlsh:tutorialspoint> describe type card\_details;

CREATE TYPE tutorialspoint.card\_details (

num int,

pin int,

name text,

cvv int,

phone set<int>,

mail text

);

## **Deleting a User-defined Data Type**

**DROP TYPE** is the command used to delete a user-defined data type. Given below is an example to delete a user-defined data type.

### **Example**

Before deleting, verify the list of all user-defined data types using **DESCRIBE\_TYPES** command as shown below.

cqlsh:tutorialspoint> DESCRIBE TYPES;

card\_details card

From the two types, delete the type named **card** as shown below.

cqlsh:tutorialspoint> drop type card;

Use the **DESCRIBE** command to verify whether the data type dropped or not.

cqlsh:tutorialspoint> describe types;

card\_details

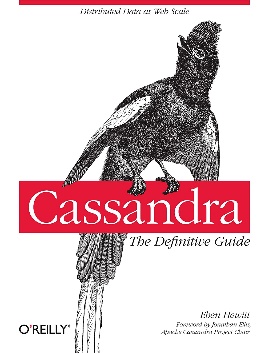
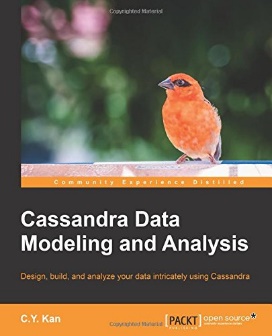
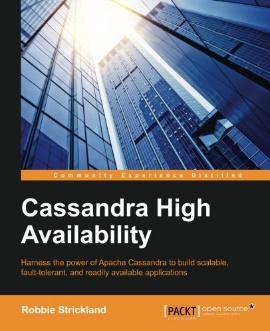
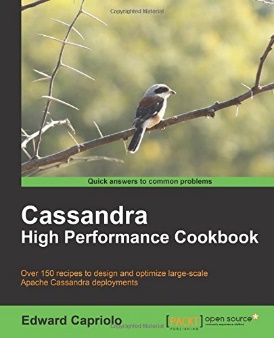
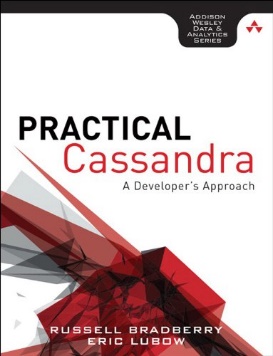
# Cassandra - Useful Resources

The following resources contain additional information on Cassandra. Please use them to get more in-depth knowledge on this topic.

## **Useful Links on Cassandra**

* [Cassandra](http://cassandra.apache.org/) - Official site of Cassandra.
* [Cassandra Wikipedia](http://en.wikipedia.org/wiki/Apache_Cassandra) - Wikipedia reference for Cassandra

## **Useful Books on Cassandra**

* [](http://www.amazon.com/Apache-Cassandra-Hands--Training-Level/dp/1500548391/httpwwwtuto0a-20) [](http://www.amazon.com/Cassandra-Definitive-Guide-Eben-Hewitt-ebook/dp/B004FGMTZY/httpwwwtuto0a-20) [](http://www.amazon.com/Cassandra-Data-Modeling-Analysis-C-Y/dp/1783988886/httpwwwtuto0a-20) [](http://www.amazon.com/Cassandra-High-Availability-Robbie-Strickland/dp/1783989122/httpwwwtuto0a-20)
* [](http://www.amazon.com/Cassandra-Performance-Cookbook-Answers-Problems/dp/1849515123/httpwwwtuto0a-20) [](http://www.amazon.com/Practical-Cassandra-Developers-Addison-Wesley-Analytics-ebook/dp/B00HDFOUOM/httpwwwtuto0a-20)