# Cassandra

**About Cassandra**

Data is doubling every two years, and this exponential growth of data is exceeding the capacity of traditional computing. Organizations need an infrastructure to implement the following:

* Provide a rapidly scalable and flexible storage infrastructure
* Continue to provide consistent performance with varying workloads
* Support for multiple data centers across geographies
* Very quick reads and writes
* High availability 24X7, 365 days
* Reduce operational cost

Apache Cassandra is a free and open-source NoSQL column family store that provides an infrastructure to meet the above challenges. That is, it is highly scalable and stores related data in the same row of the table avoiding the need for joins, hence very fast reads and writes. In addition, Cassandra ensures high availability even in case of node failure.

It mainly focuses on designing data models for the database and querying it using Cassandra Query Language (CQL), optimizing the overall performance of the system and integration of Cassandra with other technologies.

**eCommerce logging scenario**

An eCommerce online shopping site maintains every interaction of its users in a log file. Data being logged includes a unique session id, customer id, access time, access type, the client IP address, the operation requested for and so on. This log data helps in finding the number of visitors to the shopping site, analyze times when the site is most popular, identify search terms people use the most, and other related information.

The shopping site will be shortly announcing a clearance sale offering huge discounts. During this period, the server is flooded with millions of user requests and the amount of data being logged reaches the peak. In such situations too, users expect the server to be available all the time 24X7 365 days without crashing.

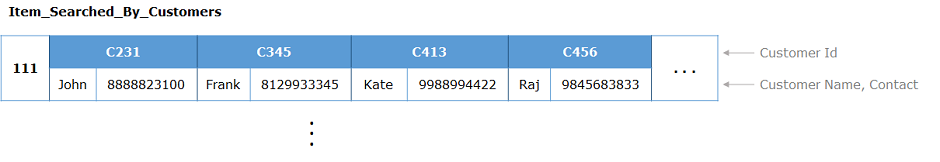
For the given scenario, you will see why Cassandra is chosen over RDBMS to meet the logging requirements of this eCommerce site.

| **Requirement** | **Challenges with RDBMS** | **With Cassandra** |
| --- | --- | --- |
| Millions of log entries every day | Cannot efficiently handle huge volumes of log data | Highly scalable, hence can deal with any volume of log data |
| Logging is write-heavy (writing to the database  more frequently than reading) | Becomes a bottleneck with continuous writes | Apt for write heavy workloads |
| Exponentially growing number of users | Difficult to serve users worldwide using the centralized single node model | can handle millions of user requests per day |
| Server with zero downtime | Server becomes unavailable during hardware failures | Can continue working even when nodes are down |
| Ease of use | Clustering is costly and complex to administer | Simple caching mechanism is used to store frequently accessed rows |

**Wide rows**

Added to these, unlike RDBMS, Cassandra supports wide rows with a very flexible schema wherein all rows need not have the same number of columns.

Consider the eCommerce scenario of customers searching for items. You may want to represent all customers searching for a particular item, as a single row, corresponding to the given item\_id as illustrated below. Also, if the item is in demand, there may be millions of customers searching for it. So wide rows would be required for the given scenario.



In general, other advantages of Cassandra over RDBMS are listed below:

| **Characteristic** | **RDBMS** | **Cassandra** |
| --- | --- | --- |
| Database model | Relational | Non-relational |
| Datatype supported | Only structured data | Both structured and unstructured data |
| Schema | Rigid | Flexible |
| Scalability | Vertical | Horizontal |
| Reliability | Single Point of Failure (SPOF) due to single node model | No SPOF due to its master-less architecture |

**MCQ**

An eCommerce website is using Cassandra to maintain a track of day-to-day activities of its customers. Which of the below characteristics are taken into consideration for choosing Cassandra?

A. Read heavy

B. Write heavy

C. Highly scalable

D. Consistency

**Solution**

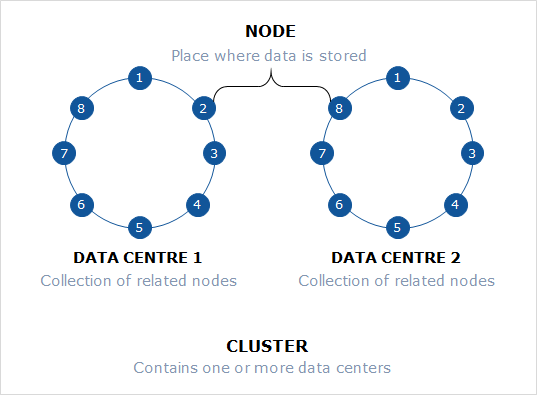
B and C are right

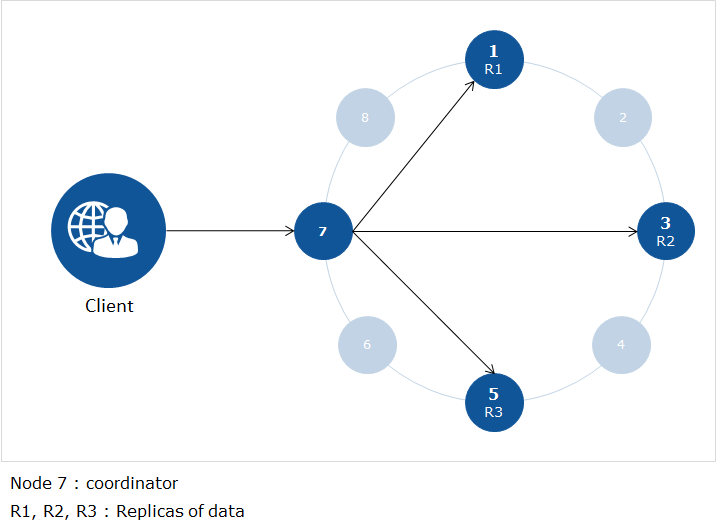
**Cassandra architecture**

Now that you have understood the use of Cassandra over RDBMS, here you will learn about its architecture.

* Cassandra follows a peer to peer master-less architecture. That is, all the nodes in the cluster are equal
* Data is replicated on multiple nodes so as to ensure fault tolerance (in case of node failure) and high availability
* Cassandra supports a read-write anywhere design policy, wherein, users can send their read/write requests to any available node regardless of which node actually stores the data
* The node that receives the client request for read/write is called the **coordinator**
* The coordinator forwards the request to the appropriate node responsible for the given row key

**Key Structure of Cassandra**





**Introduction to Apache Cassandra**

**Cassandra**

* is an open source column family NoSQL database
* is massively scalable, with high performance, and distributed in nature
* is highly available, therefore suitable for business-critical applications
* supports high speed writes on petabytes of data, with adequate read efficiency

**Applications of Cassandra**

* Suitable for high velocity data from sensors
* Useful for time series data
* Preferred by companies providing messaging services for managing massive amount of data
* Social media networking sites use Cassandra for analysis and recommendation of products to their customers
* and many more..

**Cassandra Query Language (CQL)**

* Users communicate with the Cassandra database using CQL
* Cassandra commands are executed at the CQL shell **cqlsh**
* The syntax of CQL is similar to SQL

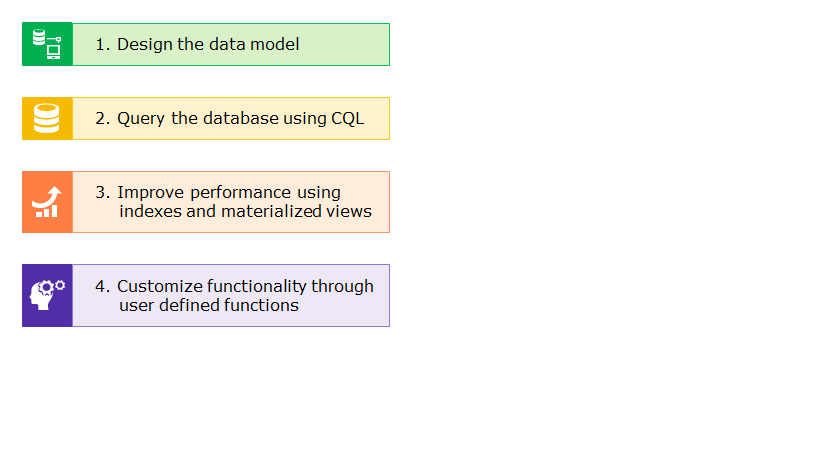
**eCommerce logging usecase – The Big Picture**

An eCommerce online shopping site has more than 200 million registered customers. Every second, thousands of its customers worldwide are accessing this application. All activities of the customers need to be logged into the database. These include unique session id, access time, client IP address from where the request originated, customer id, type of access, url of the page visited, operation performed, http method used (GET, SEARCH, ORDER, PROD, ADDBI ..) etc.

**Requirement specification for an eCommerce online shopping site**

1. Website records activities for millions of its customers in a log file
2. Need a keyspace to hold all the database objects (tables, indexes, views, user defined data types, user defined functions)
3. Bulk load log data into Cassandra tables
4. Query the log table to increment counters whenever a customer visits the site
5. Use indexes for faster search
6. Use materialized views to speed up customer queries to database and avoid update anomalies
7. Customize frequent customer operations through user defined functions

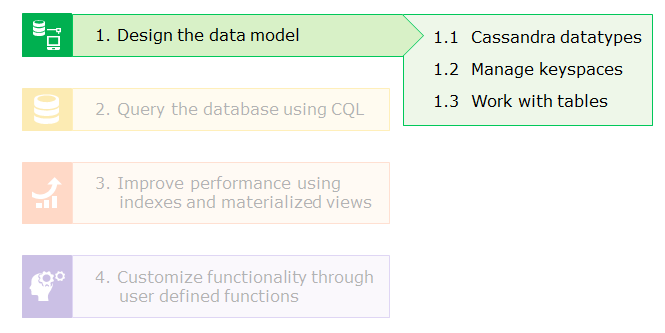
You will start working with Cassandra to meet the above requirements through the use of CQL (Cassandra Query Language).



**Designing the data model**

You will implement solutions for the requirements specified for the online eCommerce shopping site using four simple stages as shown below.

* 1. Cassandra Datatypes
  2. Managing Key-Spaces
  3. Working with tables



**Cassandra Datatypes**

Some of the commonly used datatypes in Cassandra are listed below:

* **Primitive types:** ascii, int, bigint, boolean, blob, decimal, text, timestamp, float, double, inet, timeuuid, uuid, varchar, varint
* **Special type\*:** counter
* **Collection types\*:**
  + list - Used to store ordered list of values
  + set - Used to store unordered list of unique values
  + map - Stores data as key value pairs with a value associated to each unique key

For example, consider the customer log data containing variety of data that can be represented as follows:

**Customer\_info**

| **Column name** | **Datatype** | **Description** |
| --- | --- | --- |
| session\_id | uuid | Universal unique ID |
| access\_time | timestamp | Date and time of access |
| access\_type | text | GET, SEARCH, ORDER, PROD and so on |
| client\_ipi | inet | Client IP address |
| cust\_id | int | Customer ID |
| http\_status\_code | int | 200, 404, 500 etc. |
| operation | text | /Product, /Basket, /SelectPaymentMethod etc. |

**\***More on counter columns and collection types will be discussed in later sections.

**SQL Vs CQL**

Databases deal with managing data in tables. CQL is used to query Cassandra just the same way that SQL is used to query relational databases. So, lets first understand the analogy between these and how they differ.

| **SQL** | **Cassandra** |
| --- | --- |
| Database | Keyspace |
| Table | Table (Column Family) |
| Primary key | Row key |
| Column | Column (key/value) |
| Support joins, foreign key and unique constraints | Does not support joins, foreign key or unique key |
| Supports AND, OR and NOT | Supports only AND operator |
| ORDER BY possible on any key | ORDER BY possible only on clustered columns |
| Support all relational operators such as <. <=. =, >. >= | The use of <. <=. = .>, >= are permitted only on clustered columns |
| WHERE clause is optional for all CRUD operations | WHERE clause is mandatory for UPDATE and DELETE operations |
| Cannot INSERT an already existing row  (corresponding to a given primary key value) | Can INSERT values for an already existing row wherein it performs an UPDATE |
| Cannot UPDATE values for a non-existing row (if the primary key is not found) | UPDATE of a non-existing row results in an INSERT operation |

Next, you will learn more about the CQL terminologies mentioned in the table above such as keyspaces, tables, row keys, clustered columns and so on.

**Managing keyspaces**

In Cassandra, keyspace contains data pertaining to your application. It includes tables, indexes, materialized views, user defined datatypes, user defined functions and so on.

**Creation of keyspace**

**Syntax:**

CREATE KEYSPACE  [IF NOT EXISTS] <keyspace\_name> WITH <properties>;

**Example:**

CREATE KEYSPACE IF NOT EXISTS onestop  
WITH replication={'class':'SimpleStrategy', 'replication\_factor':1};

**Note**:

* It is mandatory to specify the replica placement strategy and replication factor while creating a keyspace
* '**replication\_factor**' denotes the number of nodes, each on which, a copy of the same data is to be stored
* Use '**SimpleStrategy**'for a cluster setup with a single data center
* Use '**NetworkTopologyStrategy**' as the replication class**\***  for a cluster set up with multiple data centers

**\***More on replication strategy classes will be discussed later.

**Describing the keyspace**

* You can use the following command to view the structure of the keyspace and its contents

DESCRIBE KEYSPACE eCom;

**Altering the structure of the keyspace**

* If you want to change the replication factor, use the ALTER command as shown below

ALTER KEYSPACE eCom WITH replication={'class':'SimpleStrategy', 'replication\_factor':2};

**Using the keyspace**

* To use the created keyspace, run the below command

USE eCom;

**Describing all keyspaces**

* To view all available keyspaces, run the following command

DESCRIBE KEYSPACES;

**Dropping the keyspace**

* You can delete an unwanted keyspace using the following command

DROP KEYSPACE IF EXISTS eCom;

Next, you will see how to create, alter and drop tables in the **eCom**keyspace.

The log data generated by the online eCommerce shopping website needs to be stored in a table corresponding to the **eCom**keyspace. Here, you will learn to create tables for the same.

**Table creation**

**Syntax:**

CREATE TABLE [IF NOT EXISTS] <table\_name> (

<column\_name> datatype,

<column\_name> datatype,

<column\_name> datatype

.

.

.

PRIMARY KEY (<column\_name>, [<column\_name>…])

);

**Example:**

CREATE TABLE IF NOT EXISTS customer\_log (

session\_id uuid PRIMARY KEY,

access\_time timestamp,

client\_ip inet,

cust\_id int,

access\_type text,

operation text,

http\_status\_code int

);

**Note**:

* Data type **inet**is used to represent an IP address either in IPv4 or IPv6 format and **timestamp**indicates the date and time
* In Cassandra, it is mandatory for every table to have a primary key

**Partition key**

* A primary key that is defined using a single field is called the **partition key**
* Value of the partition key is used to determine which node stores the corresponding row in the distributed cluster
* In the above example, **session\_id**is the partition key

**Composite primary keys and clustered columns**

* Sometimes, a single field is not enough to uniquely identify a row
* CQL supports multiple fields to be a part of the primary key (composite primary key)

Consider the **search\_list** table where concurrent customers search for the same or different items at various timestamps.

| **customer\_id** | **searched\_at** | **item\_searched** |
| --- | --- | --- |
| C111 | 2017-10-18 14:16:27.432000+0000 | item-123-345 |
| C222 | |  | | --- | | 2017-10-18 14:16:27.562000+0000 | | item-123-345 |
| C111 | |  | | --- | | 2017-10-18 14:16:27.562000+0000 | | item-810-186 |
| C333 | |  | | --- | | 2017-10-18 14:16:27.561000+0000 | | item-101-318 |
| C222 | |  | | --- | | 2017-10-18 14:18:24.432000+0000 | | item-123-345 |
| C111 | 2017-10-18 14:17:33.218000+0000 | item-101-318 |

From the above sample data, the following assumptions need to be taken into consideration:

* The same customer (C111) searching for different items (item-123-345, item-810-186, item-101-318) at varied timestamps
* Concurrent customers (C111, C222) searching for the same item (item-123-345) at the same timestamp
* Concurrent customers (C111, C333) searching for the same item (item-101-318) at varied timestamps
* Concurrent customers (C111, C222) searching for different items (item-123-345, item-810-186) at the same timestamp

For the given assumptions, **customer\_id**alone does not serve as a primary key. Hence, you need to consider the composite primary key (**customer\_id, searched\_at**).

In a composite primary key,

* the first field is the **partition key**, and the remaining fields are called **clustered columns**
* data on disk is stored in sorted order of clustered columns for the corresponding set of  rows assigned to the respective nodes
* hence, **ORDER BY** can be performed only on clustered columns and that too in the same order as specified in the primary key

**With clustering order by**

By default, clustered columns are stored in ascending order. But you can use

**WITH CLUSTERING ORDER BY (<clustered\_column> DESC [,<clustered\_column> DESC...])**

to explicitly specify the ordering of clustered columns.

The search\_list example below shows that you want to store data in order of the most recent searches, hence ordering is in descending order.

**Creating the search\_list table using a composite primary key**

CREATE TABLE search\_list

( customer\_id text,

searched\_at timestamp,

item\_searched text,

PRIMARY KEY (customer\_id, searched\_at)

) WITH CLUSTERING ORDER BY (searched\_at DESC);

In the code above, **customer\_id** is the partition key and **searched\_at**is the clustered column.

**Describing the table**

You can view the structure of the table using the below command

DESCRIBE TABLE customer\_log;

**Describing all tables**

To view all tables within a given keyspace, run the following command

DESCRIBE TABLES;

**Altering table structure**

To add a new text column named **query**, use the command below

ALTER TABLE customer\_log ADD query text;

To add or modify properties corresponding to a table, use the WITH option. The example below adds a **comment**property

ALTER TABLE customer\_log WITH comment = 'This is logging based scenario';

To rename primary keys, use the following command

ALTER TABLE customer\_log RENAME session\_id TO instance\_id;

**Note**: Currently, non-primary keys cannot be renamed

To drop an existing column from the table, run the following command

ALTER TABLE customer\_log DROP query;

**Truncating a table**

To clear the contents of a table, use the below command

TRUNCATE TABLE customer\_log;

**Dropping the table**

Below command is used to drop the table when it is no longer required

DROP TABLE IF EXISTS customer\_log;

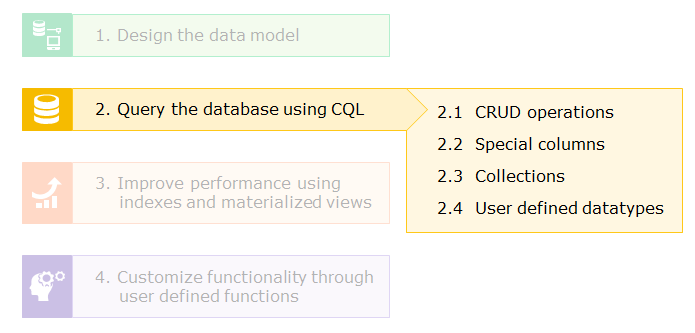
**ProblemStatement:**

The following case study is based on a company's application that monitors Web Service (eWS) resources which its customers run in real time. Within this application, the **Cloud\_Log**monitor collects data that is used to measure resources and their usage relating to eWS. It tracks data and activities such as timestamp, IP address, CPU usage, disk operations, etc. for managing resource utilization optimally.

It captures a list of performance metrics from eWS such as instanceID, timestamp, ipAddress, cpuUtilization, diskReadBytes, and diskWriteBytes.

As part of this exercise, you are required to perform the following tasks:

1. Create a keyspace **eWS**with the replication strategy as SimpleStrategy and replication factor of 2.
2. Within the keyspace created above in (1), create a table called **Cloud\_Log** with columns such as **instanceID, timestamp, ipAddress, cpuUtilization, diskReadBytes***,*and **diskWriteBytes***.* Make sure that **instanceID**is generated as a unique primary key.



**CRUD operations**

**The INSERT query**

**Syntax:**

INSERT INTO <table\_name> (<column\_name>, [<column\_name>…])  
VALUES (<column\_value>, [<column\_value>…]);

**Example:**

* Load log data into the '**customer\_log**' table using the following command

INSERT INTO customer\_log  
(session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation)  
VALUES ( uuid(), toTimestamp(now()), ‘get’, ‘192.168.1.179’, 1214, 200, ‘/Catalog’ );

**Note**:

* uuid()is used to generate a universally unique id and **toTimestamp(now())**is used to get the current date and time (timestamp)
* If the row being inserted is already available, the query would update the row with the values provided in the insert query. That is, insert would work as an update query for an already existing row

**The SELECT query**

You will see how to analyze log data using the SELECT command as shown below.

* Retrieve all rows. For e.g., analyze all customer logs

SELECT \* FROM customer\_log;

* Retrieve selective columns, for e.g., find the customer id and IP address for all rows

SELECT cust\_id, client\_ip FROM customer\_log;

* Filter rows based on primary key, for e.g., retrieve log data for a given session id

SELECT \* FROM customer\_log  
WHERE session\_id=60c63be5-c3d9-4125-9db8-3c4ac8cf040e;

* Filter rows based on non-primary key (Use ALLOW FILTERING). For e.g., retrieve log data for customer id 1214

SELECT \* FROM customer\_log  
WHERE cust\_id=1214 ALLOW FILTERING;

**Note**:**ALLOW FILTERING** has a performance hit as it needs to scan the entire table to fetch the required rows

* Using the IN operator to retrieve data corresponding to multiple sessions

SELECT \* FROM customer\_log  
WHERE session\_id IN (60c63be5-c3d9-4125-9db8-3c4ac8cf040e, ae369bfb-27ee-45fc-a18a-655c02bf7554);

**Note**: IN predicates on non-primary-key columns, for e.g., cust\_id, is not yet supported.

Next, you will see how to UPDATE and DELETE data in the Cassandra tables.

**The UPDATE query**

You can change the contents of a table using the UPDATE command as shown below.

* Query to update the **access\_type**for a given **session\_id**is shown below

UPDATE customer\_log SET access\_type=‘search’   
WHERE session\_id=60c63be5-c3d9-4125-9db8-3c4ac8cf040e;

**Note**:

* WHEREclause is mandatory in UPDATE and DELETE commands
* If the row does not exist for the given row key, an insert would be performed with the values being set in the update query. That is, update works as an upsert operation.

**The DELETE query**

Cassandra allows you delete either an entire row(s) or just the specific column(s).

* Deleting an entire row, given a session id

DELETE FROM customer\_log  
WHERE session\_id=60c63be5-c3d9-4125-9db8-3c4ac8cf040e;

* Deleting the **access\_type**column for a particular row

DELETE access\_type FROM customer\_log  
WHERE session\_id=60c63be5-c3d9-4125-9db8-3c4ac8cf040e;

**Batch Updates**

In Cassandra, BATCH is used to execute multiple DML statements (INSERT, UPDATE, DELETE) within a single request. Either all the statements are executed successfully or none of them are performed. That is, if one statement fails, all would fail.

BEGIN BATCH  
INSERT INTO customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, operation)  
   VALUES ( uuid(), toTimestamp(now()), ‘get’, ‘192.168.1.1’, 1321, ‘/SelectPaymentMethod’ );

UPDATE customer\_log SET access\_type='search' WHERE session\_id=ae369bfb-27ee-45fc-a18a-655c02bf7554;

INSERT INTO customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation)  
   VALUES (uuid(), toTimestamp(now()), ‘prod’, ‘192.168.1.179’, 1244, 200, ‘/Product 10800-1&French+Language+Courses&/Language+Courses’ );

DELETE FROM customer\_log WHERE session\_id=ae369bfb-27ee-45fc-a18a-655c02bf7554;

APPLY BATCH;

Now that you know how to query the database using CRUD operations, here you will learn to use counters and expiry columns (TTL, that is, Time To Live) that Cassandra supports.

**Counter columns**

Counters are used to increment or decrement the value of the counter column based on an event occurrence.

For example, in the log scenario, counters can be used to

* find the number of times a customer visits the website
* count orders placed by each of its customers
* analyze the number of times a customer searches for products and so on

**Example:**

* Creating a counter table **customer\_activity\_count**to represent the total visits, count of orders and the number of searches done by each customer id

CREATE TABLE customer\_activity\_count (   
cust\_id int PRIMARY KEY,   
total\_visits counter,  
no\_of\_orders counter,  
no\_of\_searches counter  
);

**Note**:

* Counter tables cannot have non-counter columns.That is, all non-primary key fields must be of counter type only
* INSERT operation cannot be performed on counter tables
* Counters can only be incremented or decremented but cannot be set to a value in the UPDATE query
* Updating the appropriate counters for each activity performed as discussed above

UPDATE customer\_activity\_count SET total\_visits = total\_visits + 1 WHERE cust\_id = 1234;

UPDATE customer\_activity\_count SET no\_of\_orders = no\_of\_orders +1 WHERE cust\_id = 1234;

UPDATE customer\_activity\_count SET no\_of\_searches = no\_of\_searches + 1 WHERE cust\_id = 1234;

**TTL (Time To Live)**

Sometimes data inserted into a table becomes obsolete after a given duration. Expiring such data is performed using TTL in Cassandra. It is specified in seconds.

**TTL Demo**

Assume, an OTP (One Time Password) is generated for a customer transaction during a particular session. This OTP is valid only for 5 mins after which the row is automatically deleted from the table as illustrated below.

* Creating the table **transaction\_otp** to store the required details

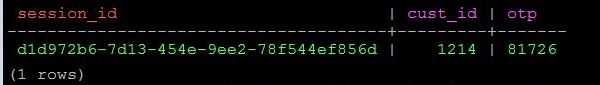
CREATE TABLE transaction\_otp\_log  
(session\_id uuid PRIMARY KEY,  
cust\_id int,  
otp int  
);

* Setting the OTP (say, 81726) to 5 mins (300 seconds) for the given customer transaction using INSERT query

INSERT INTO transaction\_otp\_log (session\_id, cust\_id, otp)  
VALUES (uuid(), 1214, 81726) USING TTL 300;

* Retrieving the row corresponding to OTP for the given customer transaction

SELECT \* FROM transaction\_otp\_log;



* Retrieving the remaining time left using the **ttl()** function before the details are deleted

SELECT TTL(otp) FROM transaction\_otp\_log;

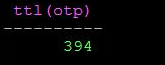


* Resetting the OTP (say, 54353) and the time to live (say, to 400 seconds) for the existing customer transaction using UPDATE query

UPDATE transaction\_otp\_log USING TTL 400  
SET cust\_id = 1214, otp = 54353  
WHERE session\_id = d1d972b6-7d13-454e-9ee2-78f544ef856d;

* Display the remaining time left after the previous update

SELECT TTL(otp) FROM transaction\_otp\_log;

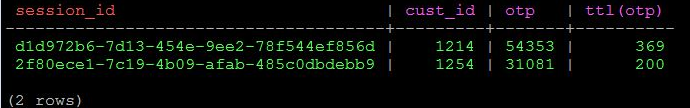


* Generating OTP for a new customer transaction using UPDATE query

UPDATE transaction\_otp\_log USING TTL 200  
SET cust\_id = 1254, otp = 31081  
WHERE session\_id = uuid();

* Displaying transaction OTP details

SELECT session\_id, cust\_id, otp, TTL(otp) FROM transaction\_otp\_log;



**Note**: TTL is not supported on tables with counter columns

Next you will learn about complex data types such as collections that CQL supports.

**Collections**

Sometimes you may want to store multiple values into a single column for a given row.

Cassandra supports the following collection types that can be used to represent a one-to-many relationship between entities.

**Set**

Used to store sorted order of unique elements. Consider a scenario illustrating a one-to-many relationship between **item**and available **payment**options:

* There may be multiple payment options for a given item
* Payment options can be stored in any order
* Each payment option is unique and duplicates will not be considered

For the given scenario set is suitable.

**Set demo**

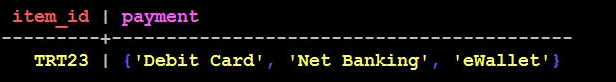
* Creating the **item\_payment\_options**table

CREATE TABLE item\_payment\_options  
( item\_id text PRIMARY KEY,  
  payment SET<text>  
);

* Inserting sample data into the **item\_payment\_options**table and retrieving the available payment options for a given item

INSERT INTO item\_payment\_options (item\_id, payment)  
VALUES( 'TRT23', {'Net Banking','eWallet', 'Debit Card'} );

SELECT \* FROM item\_payment\_options;



**Note**: The order in which items have been inserted into the set is different (stored in sorted order)

* Adding a new payment option for a given item and display the updated payment details

UPDATE item\_payment\_options  
SET payment = payment + {'CoD'}  
WHERE item\_id = 'TRT23';

SELECT \* FROM item\_payment\_options;



**List**

Useful when you want to retain the order in which elements are being added to the collection and also be able to add duplicated values. In the item search use case, the use of list is specified below:

* Customers search for any number of items
* The same item can be searched multiple times by the same customer
* Order of items searched is relevant

**List demo**

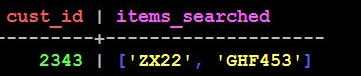
* Creating the **customer\_search**table

CREATE TABLE customer\_search  
(cust\_id int PRIMARY KEY,  
items\_searched list<text>  
);

* Appending the items searched to the list and displaying the same

UPDATE customer\_search  
SET items\_searched = items\_searched + ['ZX22', 'GHF453']  
WHERE cust\_id = 2343;

SELECT \* FROM customer\_search;

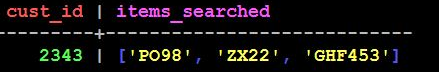


**Note**: The order in which items have been inserted into the list is retained

* Pre-pending items searched to the list and displaying the updated items searched

UPDATE customer\_search  
SET items\_searched = ['PO98'] + items\_searched  
WHERE cust\_id = 2343;

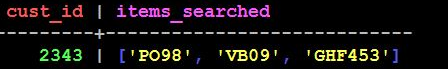
SELECT \* FROM customer\_search;



* Updating an item at a given index position in the list and displaying the updated list

UPDATE customer\_search  
SET items\_searched[1] = 'VB09'  
WHERE cust\_id = 2343;

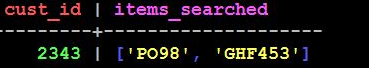
SELECT \* FROM customer\_search;



* Deleting an item at a given index position in the list and displaying the updated list

DELETE items\_searched[1] FROM customer\_search  
WHERE cust\_id = 2343;

SELECT \* FROM customer\_search;



**Map**

Used to represent data as key-value pairs. For example, representing the payment method used for various orders corresponding to a particular customer.

* Creating the **customer\_payment**table to represent payment mode used against each order

CREATE TABLE customer\_payment  
(cust\_id int PRIMARY KEY,  
payment map<text, text>  
);

* Insert payment mode used by a given customer for each order placed and display the corresponding details

INSERT INTO customer\_payment (cust\_id, payment)  
VALUES ( 1843, {'SDF23':'Net Banking', 'NJM56':'Credit Card'} );

SELECT \* FROM customer\_payment;

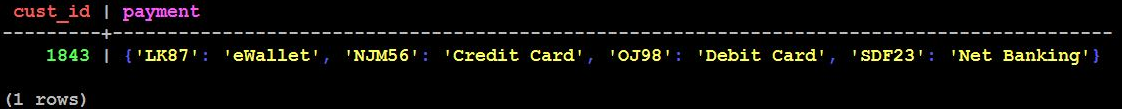


**Note**: The order in which items have been inserted into the map is changed (stored in sorted order of the key)

* Add new order-payment details to the existing one and display the updated details

UPDATE customer\_payment  
SET payment = payment + {'OJ98':'Debit Card', 'LK87':'eWallet'}  
WHERE cust\_id = 1843;

SELECT \* FROM customer\_payment;



Next, you will learn how to create user defined datatypes.

If you need to reuse a set of related fields among multiple tables, Cassandra allows you to create a user defined type (UDT).

For example, **address**field containing **door, area, city,**and **pincode**can be used to represent the customer's shipping address, delivery address or even the seller's address.

**Creating a UDT - Syntax**

CREATE TYPE <type\_name> (   
<column\_name> datatype,   
<column\_name> datatype,   
<column\_name> datatype   
  .  .  .    
);

**UDT Demo**

* Creating a user defined datatype to represent an **address**

CREATE TYPE address  
( door text,  
  street text,  
  area text,  
  city text,  
  pin int,  
  state text  
);

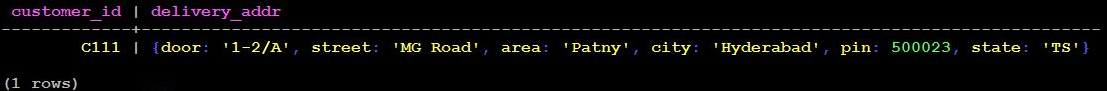
* Altering the existing **customer\_orders**table to add a new column for representing the customer's address

ALTER TABLE Customer\_Orders  
ADD delivery\_addr address;

* Updating the customer's delivery address and displaying the details

UPDATE Customer\_Orders  
SET delivery\_addr = { door:'1-2/A',  
        street:'MG Road',  
        area:'Patny',  
        city:'Hyderabad',  
        pin:500023,  
        state:'TS'  
      }  
WHERE order\_id = '210-109';

SELECT customer\_id,delivery\_addr  
FROM Customer\_Orders  
WHERE order\_id = '210-109';



This section concludes here. The next section discusses about how to use indexes and materialized views for faster retrieval of data.

**MCQ**

1. A table **emp\_data** is created with the following columns: **empId**as the primary key, **name**, **dept**, and **salary**
2. What is the output of executing the below query?

SELECT salary FROM emp\_data WHERE name='John';

("**John**"is a valid name corresponding to an employee in the **emp\_data**table)

Fetches the salary of the employee(s) whose name is "John"

Executes without an error but does not return any result

**Results in an error as filtering cannot be done based on non-primary column check**

Results in an error as it is incorrect to fetch only the salary column of an employee

**Querying the database using CQL - Exercise**

**ProblemStatement:**

The **eZon**applicationalso maintains a system which logs all the service requests from its customers. These service requests are logged when a customers raise a complaint. These complaints are assigned a priority level depending upon the severity of the issue. For every customer complaint received, a specific duration, based on its priority, is allotted before which the requests need to be serviced.

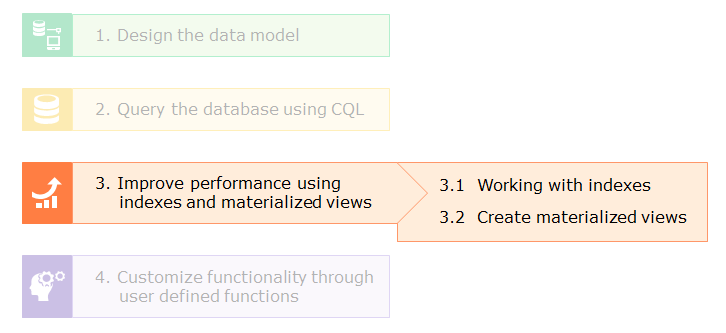
The **Service\_Log**table maintains some key attributes such as auto-generated request Id, customer Id, priority level and request description.

This Service\_Log system, apart from logging all the requests, also counts whether these requests are of high, medium or low priority.

For the above table, you need to perform the following tasks:

1. Within the **eWS**keyspace, insert the data from [metrics.txt](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_23563592206509100000/web-hosted/assets/metrics.txt) into the table **Cloud\_Log**.
2. Create a table **Service\_Log**containing an auto-generated request Id, customer Id, priority level and request description.
3. Insert data into the **Service\_Log**table to manage service requests using TTL wherein each service request has an expiry time associated with it before which the request is to be serviced.
4. Create another table **Service\_Count**used as a counter table to keep track of the number of requests based on priority.
5. Create a user defined datatype called **Customer**having columns such as **customerId, name, contact\_no,**and **emailId***.*
6. Alter the **Cloud\_Log**table to add a new column **cust\_info** of **Customer**type you created above in (5).

Being able to query the database alone is not sufficient. You should be able to retrieve data faster. Here you will learn how to improve read performance using indexes and materialized views.



**Indexes and Materialized Views**

Index needs to be created on columns that are frequently used in querying for faster access to the given rows. Primary keys are indexed by default.

**Creation of indexes**

**Syntax:**

CREATE INDEX <index\_name>

ON [<keyspace\_name>.]<table\_name> (<column\_name>);

**Example:**

* Create an index on **cust\_id**column of the **customer\_log**table

CREATE INDEX cust\_idx ON customer\_log (cust\_id);

**Note**:

* Cannot create an index on collection columns or counter fields
* Currently, CQL does not support compound indexes
* Retrieve data using the indexed key (need not use ALLOW FILTERING now)

SELECT \* FROM customer\_log WHERE cust\_id=1214;

**Dropping the index**

* Drop the index **cust\_idx**on the **customer\_log**table if it is no longer required

DROP INDEX IF EXISTS cust\_idx;

Next, you will learn about the use of materialized views.

To understand the use of materialized views, the **Customer\_Orders** table has been taken into consideration. Corresponding to the given scenario, challenges faced and appropriate solutions to overcome these challenges have been discussed here.

**Problem Statement**

The Customer\_Orders table has the following schema:

 ( order\_id text PRIMARY KEY,  
    customer\_id text,  
    year int,  
    month int,  
    day int,  
    amount int,  
    pay\_method text,  
    status text )

Assume, you want to perform the following queries:

1. Retrieve order details corresponding to each customer
2. Retrieve year-wise orders
3. Retrieve status-wise orders and so on..

**Challenges**

* With order\_id as the primary key, the above queries require a full table scan accessing data across all the nodes which is time consuming
* Moreover a single table cannot have multiple primary keys defined

**Solution 1 - Global index tables**

Store the same data in three different tables as required, each with a different primary key. For example

* CREATE TABLE customer\_wise\_orders .. PRIMARY KEY (customer\_id, order\_id)
* CREATE TABLE year\_wise\_orders .. PRIMARY KEY (year, order\_id)
* CREATE TABLE status\_wise\_orders .. PRIMARY KEY (status,order\_id)

**Challenges with solution 1**

This model can lead to errors.

The changes to one table requires an update to be executed on all other tables to keep the database consistent.

If any of the other updates are missed, it will leave the database inconsistent.

**Solution 2 - Secondary indexes**

For the existing table **Customer\_Orders**, create indexes on each query field

* CREATE INDEX cust\_idx ON Customer\_Orders (customer\_id) ;
* CREATE INDEX year\_idx ON Customer\_Orders (year) ;
* CREATE INDEX status\_idx ON Customer\_Orders (status) ;

**Challenges with solution 2**

Querying and maintaining secondary indexes for each field is a great overhead for the following reasons:

* Whenever a new row is added, the corresponding indexes get updated. Hence, performance becomes worse with growing large scale applications
* Secondary indexes are managed local to each node.  To know which node contains the queried value, all the nodes have to be searched

**Solution 3 - Materialized views**

Create multiple materialized views from the base table.

Each materialized view will have a different query field as its partition key. For example

**customer\_id** as the partition key

CREATE MATERIALIZED VIEW customer\_wise\_orders AS

SELECT year, month, day, order\_id, customer\_id, amount, pay\_method,status

FROM customer\_orders WHERE order\_id IS NOT NULL AND customer\_id IS NOT NUL PRIMARY KEY (customer\_id, order\_id);

**year**as the partition key

CREATE MATERIALIZED VIEW year\_wise\_orders AS

SELECT year, month, day, order\_id, customer\_id, amount, pay\_method,status

FROM customer\_orders

WHERE order\_id IS NOT NULL AND year IS NOT NULL PRIMARY KEY (year, order\_id);

**status**as the partition key

CREATE MATERIALIZED VIEW status\_wise\_orders

AS

SELECT year, month, day, order\_id, customer\_id, amount, pay\_method,status

FROM customer\_orders

WHERE order\_id IS NOT NULL AND status IS NOT NULL

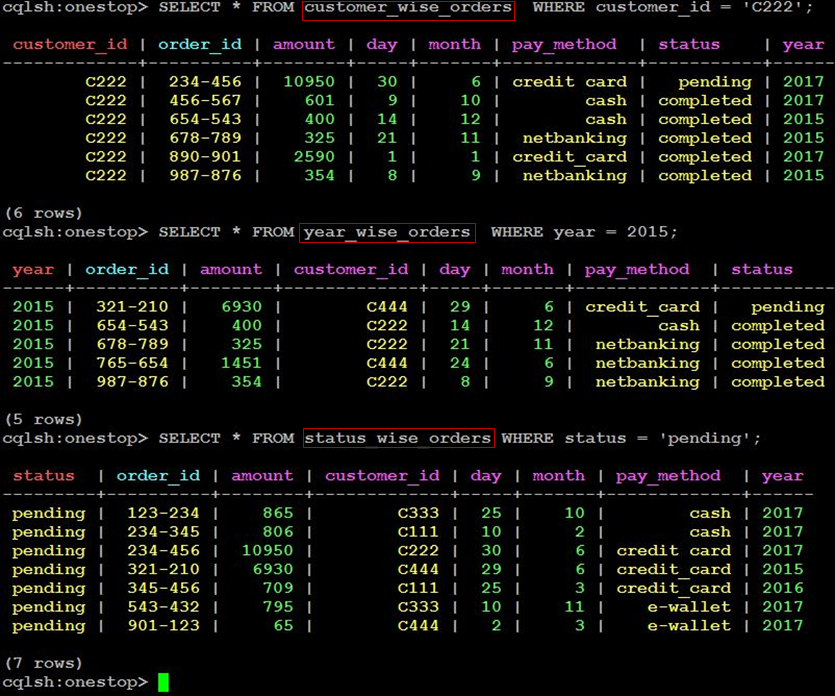
PRIMARY KEY (status, order\_id);

**Querying the materialized views**

SELECT \* FROM customer\_wise\_orders WHERE customer\_id='C222';

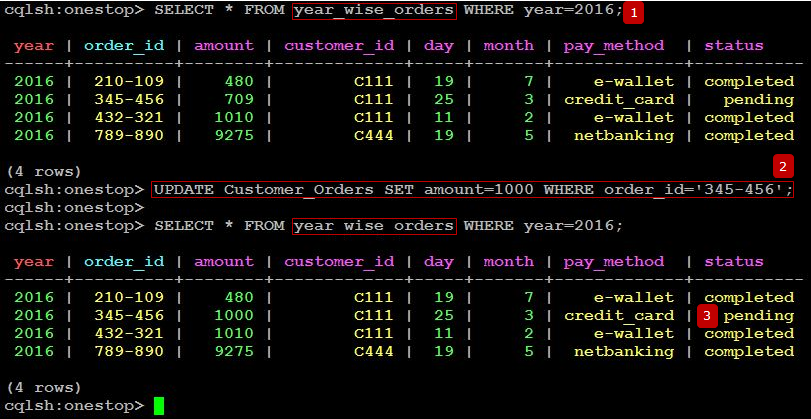
SELECT \* FROM year\_wise\_orders WHERE year = 2015;

SELECT \* FROM status\_wise\_orders WHERE status = 'pending';



**Updating the base table**

1. SELECT \* FROM year\_wise\_orders WHERE year=2016;
2. UPDATE Customer\_Orders SET amount=1000 WHERE order\_id='345-456';
3. SELECT \* FROM year\_wise\_orders WHERE year=2016



**Advantage of using materialized views over global index tables and secondary indexes:**

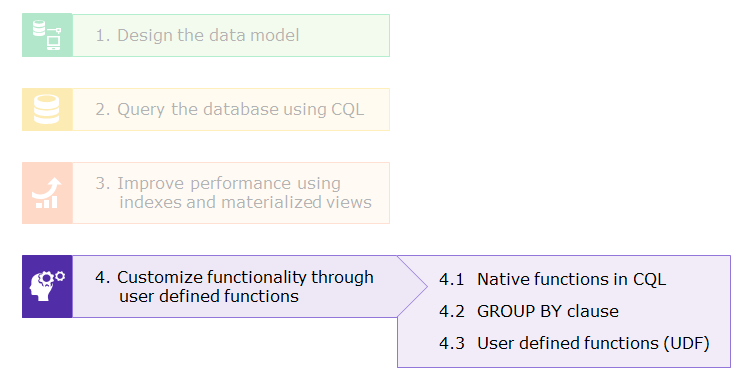
* Automatic updates reflected from the base table. Hence no inconsistency
* Query the appropriate materialized view based on the search field (partition key) to be used
* Each materialized view acts as a replica of the given base table
* Only the specific node containing the given partition key need to be reached to retrieve the required data
* Need not maintain separate index tables
* Can be queried just as any regular table
* Therefore, with materialized views there is better performance

**Points to remember**

* Need to include all primary key columns of the base table
* Primary key of the materialized view must contain exactly one key column that is not part of the base table’s primary key
* All primary key columns need to be filtered against IS NOT NULL
* Filtering against other values in WHERE clause is not possible
* Materialized views cannot be created on top of counter tables, or on other materialized views or on tables in other keyspaces

Before learning how to implement user defined functions, you will first see the built-in/native functions supported by CQL.

**Functions in Apache Cassandra**



There are two types of functions that CQL supports:

**Scalar Functions**

Executed on each row and returns a result for each row in the table. For example

* **writetime**() - Returns the write time (microseconds) of the column to the database

SELECT WRITETIME(cust\_id) FROM customer\_log LIMIT 1;



* **uuid**() - Returns a universally unique id used to identify a row uniquely

SELECT uuid() FROM customer\_log LIMIT 1;



* **now**() - Returns the current time at the time the function is invoked

SELECT toTimestamp(now()) FROM customer\_log LIMIT 1;



* **cast**() - Used to convert one native datatype to another

SELECT cast(now() as timestamp) FROM customer\_log LIMIT 1;



* **token**() - Computes the token value corresponding to a given partition key

SELECT token(cust\_id) FROM customer\_log LIMIT 1;



**Aggregate functions**

Executed on a set of rows as a group and returns a result for each group. Aggregation functions in CQL are similar to the ones provided by SQL

* Counting the number of customer orders, the average, sum, minimum and maximum order **amount**from the **customer\_orders** table

SELECT count(\*),avg(amount),sum(amount),min(amount),max(amount)

FROM customer\_orders;



Next, you will learn to aggregate data using the above functions with the GROUP BY clause.

The GROUP BY feature is supported from Cassandra 3.10 version. It's usage is similar to that of SQL databases, wherein, an aggregation operation can be performed on a group of rows yielding aggregated results for each group.

**Example**

The **Payment\_Details** table keeps track of daily payments made by customers as shown below.

CREATE TABLE payment\_details

(customer\_id text,

date text,

paid\_at timestamp,

amount float,

PRIMARY KEY (customer\_id,date,paid\_at));

* Inserting sample data into the above table and displaying the resultant details

INSERT INTO payment\_details (customer\_id,date,paid\_at,amount)

VALUES ('C111','2017-04-03','2017-04-03 07:01:00',7300);

INSERT INTO payment\_details (customer\_id,date,paid\_at,amount)

VALUES ('C234','2017-04-03','2017-04-03 07:02:00',2100);

INSERT INTO payment\_details (customer\_id,date,paid\_at,amount)

VALUES ('C234','2017-04-03','2017-04-03 08:01:00',1300);

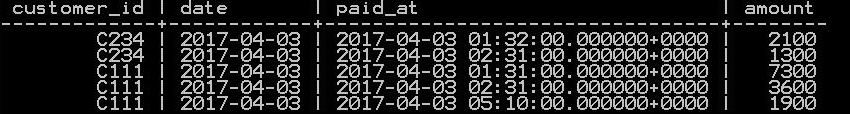
INSERT INTO payment\_details (customer\_id,date,paid\_at,amount)

VALUES ('C111','2017-04-03','2017-04-03 05:10:00',1900);

INSERT INTO payment\_details (customer\_id,date,paid\_at,amount)

VALUES ('C111','2017-04-03','2017-04-03 08:01:00',3600);

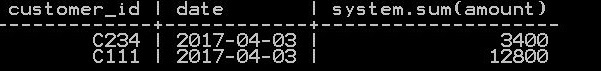
SELECT \* FROM payment\_details;



The following commands illustrate the GROUP BY operation in Cassandra 3.10 using data from the **Payment\_Details**table.

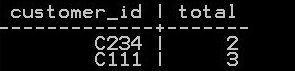
* Display the total payment **amount**per **date**corresponding to each **customer\_id**

SELECT customer\_id,date,sum(amount) FROM payment\_details GROUP BY customer\_id,date;



* Display the number of payments made by each customer

SELECT customer\_id,count(\*) as total FROM payment\_details GROUP BY customer\_id;



**Note**:

* GROUP BY is currently supported only on columns of the PRIMARY KEY, that too in the same order of the clustering columns declared
* GROUP BY is not supported on only part of the partition key.
* For example, on a table with **PRIMARY KEY( (partitionKey1, partitionKey2), clustering1, clustering2)**, the following query will NOT be supported:

SELECT partitionKey1, MAX(value) FROM myTable GROUP BY partitionKey;

Consider the **Customer\_Orders**table wherein the **amount**is specified in rupees.

**Problem Statement**

* You need the corresponding dollar value for the amount specified
* This functionality is not provided by CQL implicitly

**Solution**

* Write your own code for the currency conversion using a User Defined Function (UDF)
* Writing the code in java or javascript is directly supported by CQL

**Note**: For support on other compliant scripting languages (such as Python, Ruby, and Scala), add the appropriate JAR to the classpath.

**Syntax:**

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS] [keyspace.]functionName (param1 type1, param2 type2, …)

CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT

RETURN returnType

LANGUAGE language

AS '

// source code here

' ;

* By default, UDFs are disabled in CQL
* To enable UDFs, set **enable\_user\_defined\_functions: true**in cassandra.yaml (located at **$CASSANDRA\_HOME/conf/cassandra.yaml**)
* Also, to enable javascript based code, set **enable\_scripted\_user\_defined\_functions: true** in cassandra.yaml

**Demo for creating and using javascript based UDFs**

Creating the function **rupeeToDollar**that uses the **Math.round()**function in Javascript

CREATE OR REPLACE FUNCTION rupeeToDollar(rupee int)

CALLED ON NULL INPUT

RETURNS float

LANGUAGE javascript

AS '

if(rupee==null)

rupee=0;

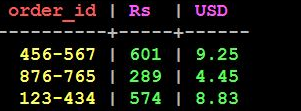
Math.round(rupee/65\*100)/100;

';

Using the function to display the order **amount**in dollars

SELECT order\_id, amount AS "Rs", rupeeToDollar(amount) AS "USD"

FROM customer\_orders LIMIT 3;



**Demo for creating and using java based UDFs**

The demo below displays the number of item searches done by each customer.

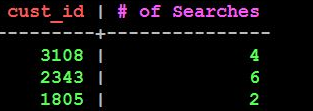
* Creating the function **numOfSearches**using the **size**() method in **java.util.List**

CREATE OR REPLACE FUNCTION numOfSearches(searched list<text>)

RETURNS NULL ON NULL INPUT RETURNS int LANGUAGE java AS ' return searched.size();';

* Calling the UDF to display the number of items searched

SELECT cust\_id, numOfSearches(items\_searched) AS '# of Searches" FROM customer\_search LIMIT 3;



**Arguments to the UDF and its return types can be any of the following:**

* Primitives (boolean, int, double, float, text, uuid, etc.)
* Collections (list, set, map)
* Tuple types and user defined data types too

**Note**:

* CALLED ON NULL INPUT ensures the function will always be executed irrespective of the value of input arguments
* RETURNS NULL ON NULL INPUT skips function execution and returns NULL if any input argument is NULL
* The scope of a UDF is **keyspace-wide**. Therefore you can add a keyspace prefix to an **UDF** name

**Drop the UDF**

* The function can be deleted when it is no longer needed using the command below:

DROP FUNCTION [IF EXISTS] <function\_name>;

**Example:**

DROP FUNCTION IF EXISTS rupeeToDollar;

Generally, there is a need to transfer data from relational databases to Cassandra server. This section focuses on migration of data from SQL table to Cassandra table.

**Problem Statement**

Assume, legacy customer log data is already available on an RDBMS server. Volumes of log data are increasing exponentially and managing Tera Bytes and Peta Bytes of this data efficiently has become a challenge for RDBMS.

**Solution**

Migrate data from SQL databases to Cassandra tables for improved performance and scalability using the following steps.

**Step 1:**Convert data from SQL tables either to **.csv** or **.tsv** files using relevant RDBMS tools

**Step 2:**Create a table (say, **customer\_log**), if not exists,whose schema corresponds to the structure of the file contents

**Step 3:** Bulk load the log file into the Cassandra table using any of the methods listed below:

* COPY command
* Using SSTable writers

**Note:**SSTables are the representation of Cassandra tables on disk.

          In this course, we will cover only the COPY command.

**The COPY command**

Use the COPY command to load data from a local file that is not too large into a Cassandra table.

COPY customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation) FROM '/root/sunitha/log.tsv' WITH DELIMITER='\t';

Few additional options that can be specified using the WITH option of the COPY command are listed below:

* **DELIMITER**- Used to set the delimiter if any other than comma (,) is used in the file. For e.g.,

COPY customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation)

FROM '/root/sunitha/log.tsv' WITH DELIMITER = '\t';

* **HEADER**- Used to indicate that the first row of the file is a header. For e.g.,

COPY customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id,

http\_status\_code, operation) FROM '/root/sunitha/log.tsv' WITH DELIMITER = '\t' AND

HEADER = true;

* **NULL**- Indicates a null value if using some other character representation other than an empty string in the source file. For e.g.,

COPY customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation) FROM '/root/sunitha/log.csv' WITH NULL = '\N';

* **DATETIMEFORMAT**- Used to set the timestamp format (say, 25/12/2015) if it varies from the default one that is '%Y-%m-%d %H:%M:%S%z'

COPY customer\_log (session\_id, access\_time, access\_type, client\_ip, cust\_id, http\_status\_code, operation) FROM '/root/sunitha/log.csv' WITH DATETIMEFORMAT = '%d/%m/%Y';

**Key components of Cassandra**

In this section, you will learn about the following key components of Cassandra:

* Gossip protocol
* Partitioners
* Replica placement strategy and
* cassandra.yaml*-*The configuration file

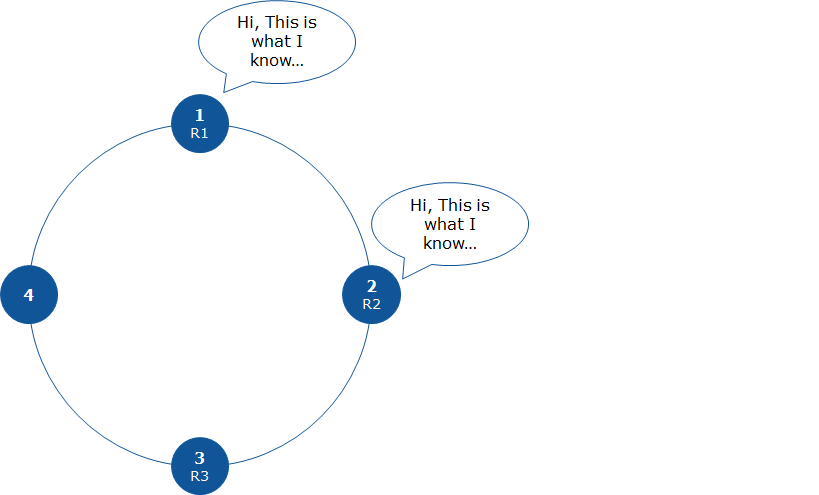
**Gossip protocol**

Cassandra

* has a peer-to-peer architecture.  In this architecture all the nodes are equal.
* uses the gossip protocol for inter-node communication.  Periodically nodes exchange their state information.

In a Cassandra cluster,

* a node gossips with up to three other nodes each second, and the message has a version to maintain the node's current state



**Partitioning strategy**

In Cassandra,

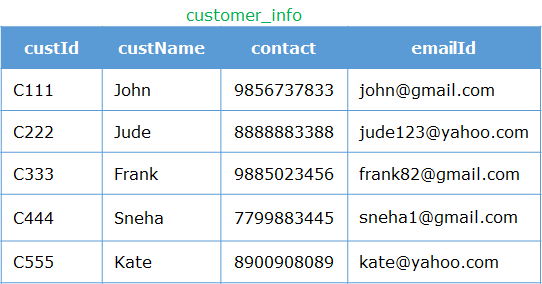
* data is organized into rows in tables and each row is identified by a primary key (row key)
* each node in the cluster is responsible for a set of data (tokens)
* partition key's value determines which node a row would be stored on

There are two strategies for the distribution of data across nodes in the cluster:

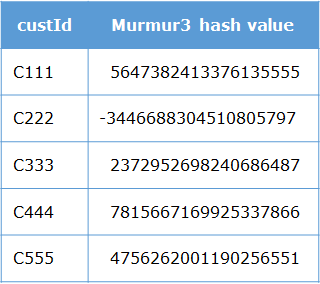
Random partitioning

* For each partition key, the Murmur3Partitioner class generates a corresponding hash value (token) which is used to identify the node that is responsible for the given token value
* Thus, rows are distributed randomly to the nodes in the cluster based on the hash values of their respective partition key
* This strategy minimizes the need to reorganize data whenever new nodes are added/removed from the cluster

For example, consider the following example



Cassandra assigns a hash value to each partition key as follows.



Ordered partitioning

* The ByteOrderedPartitioner class distributes rows to various nodes in sorted order of the partition key
* This partitioner class is useful when most of your queries are range based scans
* For example, if customer name is the partition key, range of data would be distributed to nodes in the alphabetical order of the names. Then you can scan for customers whose names fall between 'Priya' and 'Sneha' for example.

Reasons why ordered partitioner is not so useful as compared to random partitioner

* Data distribution may be uneven. For instance, the number of names beginning with letter 'S'  is much more than those beginning with 'X', so most of the requests would go to one node
* Adding new nodes or removing existing nodes may require data movement to maintain the sorted order
* An ordered partitioner requires administrators to manually calculate token ranges based on their estimates of the row key distribution

**Replica placement strategy.**

Replicas are

* copies of the same data stored on multiple nodes in the cluster
* used to ensure reliability and fault tolerance wherein if one node storing the data fails, the request can be processed from any other nodes holding the replica

Also,

* the total number of replicas is known as the **replication factor**
* a request for read/write can be processed at any available replica as all are equal and there is no priority of one copy over the other

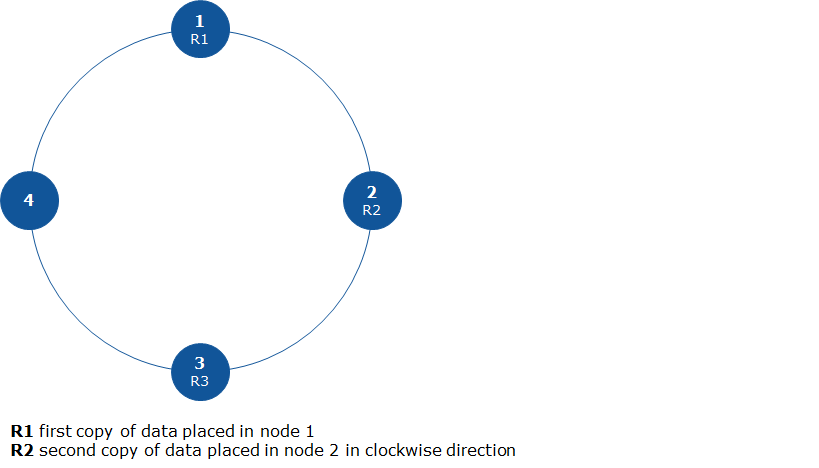
**Note**: Replication factor should not exceed the total number of nodes in the cluster.

**On which nodes to store replicas?**

Use one of the two replica placement strategy listed below to identify nodes where replicas may be placed

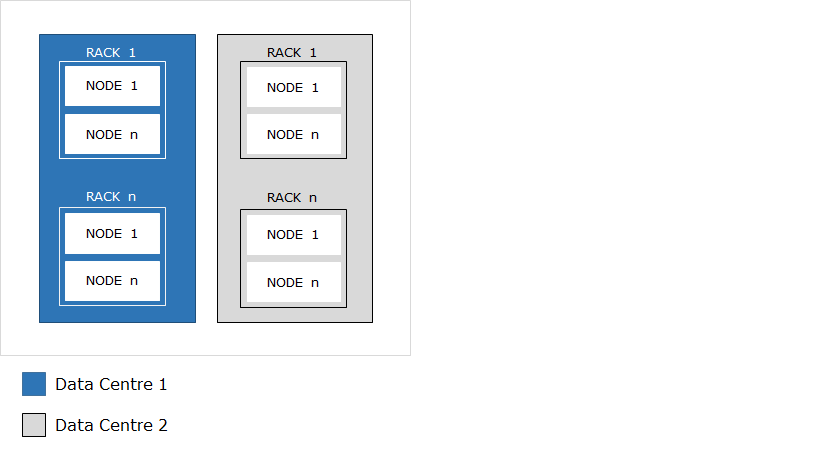
**SimpleStrategy**

* Used when the setup is within a single data center only
* First copy is placed on the node identified by the partitioner class. The remaining replicas are placed on next nodes in ring clock-wise direction depending on the replication factor

****

**NetworkTopologyStrategy**

* Used for multi-data center setup
* Highly recommended for production deployments as it is easier to scale the cluster size in future as required
* First copy is placed on the node identified by the partitioner class. Replicas are placed on next nodes belonging to a different rack within the same data center in ring clock-wise direction
* If the setup is not rack-aware, then **SimpleStrategy**is used to place the replicas
* Here, replication factor is maintained per data center



**Note**: If all copies are stored on a single rack, then rack failure makes the data unavailable to users.

Next, you will learn the use of the configuration file, cassandra.yaml

When Cassandra is installed, its default properties are stored in**cassandra.yaml** at the path $CASSANDRA\_HOME/conf.

Some of the common properties that you can alter in cassandra.yamlare cluster\_name, partitioner, data\_file\_directories, enable\_user\_defined\_functions, listen\_address, native\_transport\_port and so on.

**Data modeling in Cassandra**

**Architectural view of Cassandra**

Until now you have learnt Cassandra from a developer's point of view. In this section, you will know more about Cassandra from an architect's perspective. The following topics would be discussed here.

* Data modeling in Cassandra
* Performing tuning
* When to use and when not to use Cassandra
* Cassandra best practices
* Why to integrate with other technologies such as Hadoop and Spark

**Data modeling in Cassandra**

Data modeling refers to the way you design tables in the database. It greatly impacts the read/write performance. Therefore, model data based on your query access patterns. For instance, as the cost of storage has reduced greatly, it is okay for you to maintain a table per query if required (duplicate data if it helps reducing time to query in case speed is more important than storage cost).

**Consideration for data modeling**

* Identify query patterns that are expected to be performed on the given data set, such as, column(s) on which you would perform a group by, order by, filter data, and so on
* For write heavy workloads, ensure uniform distribution of data across nodes in the cluster. Choose the primary key carefully as the hash value of the partition key (first field in the primary key) is used to identify the node on which the row would be stored
* For faster reads, minimize the number of partitions (data pertaining to the row key) that need to be retrieved for the given query

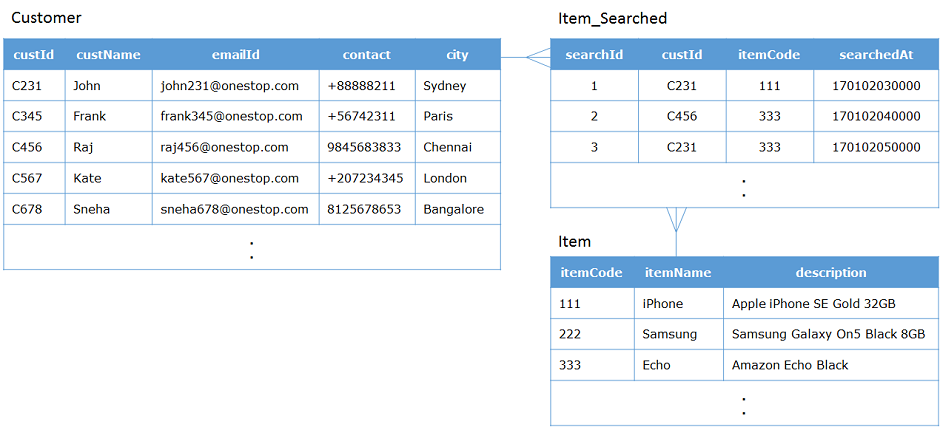
**Note:**Use appropriate materialized views based on the partition key to be used for the given query.

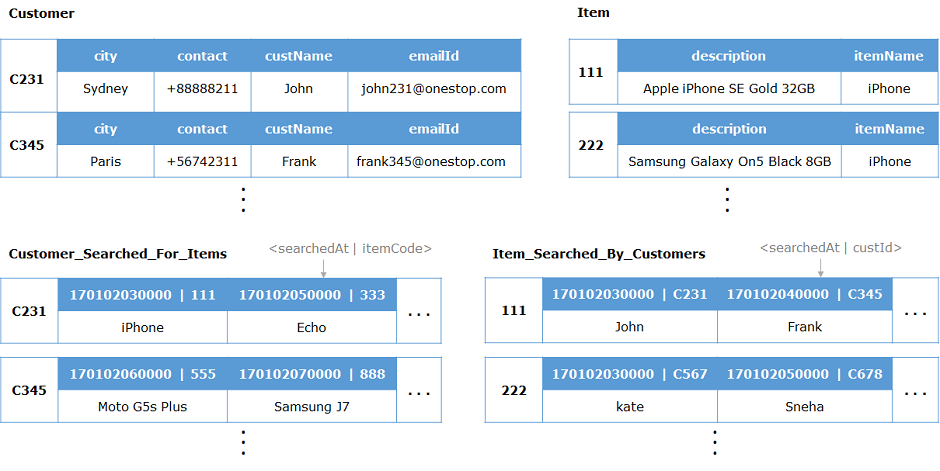
**Modeling relationships for an online eCommerce site**

Consider a scenario of customers searching for one or more items. Also, the same item may be searched by multiple customers. You need to model the many-to-many relationship **Item\_Searched** between customers and items.

Find customers by custId

For e.g., querying for customer '**C231**' should return the row ('**C231**','**John**','**john231@onestop.com**',+**88888211**,'**Sydney**')





1. Find items by itemCode

For e.g., querying for item **111**should return the row (**111**,'**iPhone**','**Apple iPhone SE Gold 32GB**')

1. Find all customers searching for a particular item

For e.g., retrieving names of customers searching for item **333** should return '**Raj**' and '**John**'

1. Find all items being searched by a particular customer

For e.g., retrieving items searched by customer '**C231**' should return the following rows

(**111**,'**iPhone**','**Apple iPhone SE Gold 32GB**') and (**333**,'**Echo**','**Amazon Echo Black**')

Using this model, you can retrieve item names searched by a particular customer and also names of all customers who searched for a given item.

Also, storing data based on timeuuid (searchedAt) makes range queries on time slots very efficient. For example, you can efficiently query for the most recent customers who searched a given item and the most recent items searched by a given customer, without the need to read all columns of a row.

In case you need to retrieve other details such as city, contact or emailId of customers who searched for a particular item, then two lookups would be performed. First, retrieve the custIds for the given item from the **Item\_Searched\_By\_Customers** table and then lookup the **Customer** table for the required details.

Using this design, even if the number of customers searching for a particular item increases, you would require not more than two lookups to be performed.

**Using CQL to represent this model**

* Creating the **Customer**table

CREATE TABLE customer  
(custId text PRIMARY KEY,  
custName text,  
emailId text,  
contact bigint,  
city text  
);

* Creating the **Item** table

CREATE TABLE item  
(itemCode int PRIMARY KEY,  
itemName text,  
description text  
);

* Creating the **Customer\_Searched\_For\_Items** table

CREATE TABLE customer\_searched\_for\_items  
( custId text,  
  searchedAt timeuuid,  
  itemCode int,  
  itemName text,  
  PRIMARY KEY (custId, searchedAt, itemCode)  
) WITH CLUSTERING ORDER BY (searchedat DESC);

* Creating the **Item\_Searched\_By\_Customers** table

CREATE TABLE item\_searched\_by\_customers  
( itemCode int,  
  searchedAt timeuuid,  
  custId text,  
  custName text,  
  PRIMARY KEY (itemCode, searchedAt, custId)  
) WITH CLUSTERING ORDER BY (searchedAt DESC);

**Performance tuning**

Performance tuning is done to improve the system performance. In this section, you will learn to tune the following resources

* caches
* compaction

Here, you will see the use of memory caches for better performance.

**Caches**

Consider the items being searched by customers. Some items that are currently in demand, for example an iPhone, may be searched more frequently than the other items. Therefore, for faster access, these items are stored in cache (cache reads are approx. 100X times faster than main memory reads). Here, you will learn to tune data caches.

**Key cache**

* Stores the exact location of the partition in the SSTable on disk (hence reducing the number of read seeks per SSTable)
* Key caches are stored on heap
* Size is configurable on a per-SSTable basis during table creation
* Key cache performance is monitored through 'key cache hit rate' as a result of executing the command **nodetool cfstats**
* To set the optimal size of the key cache required, use the formula,

**Σ (key cache size for each table) X (average size of keys for the table)**

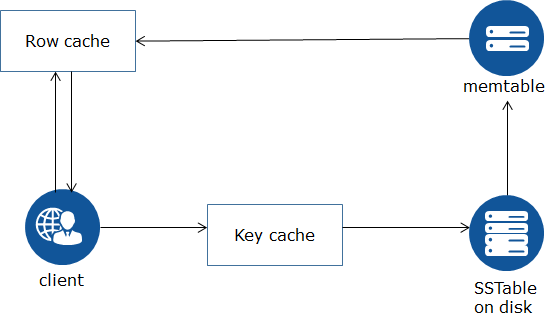
Key cache becomes too expensive if keys within a table are accessed randomly without being repeated, as you need to keep majority of the keys cached. Therefore, monitoring the key access patterns may help you determine if a key cache is appropriate and what size is optimal.

**Row cache**

* Unlike the key cache, the row cache holds the entire contents of frequently accessed row keys in memory
* Very useful if the size of data is small enough to load the complete row in cache and also when almost all columns of the given row are requested in the query
* In case of a row cache miss (wherein the row requested for is not already loaded in cache), the key cache might be accessed for the given row key that eliminates one seek per SSTable making a disk read efficient

**Workflow with caching enabled**

* Consider the scenario of items searched by customers wherein some items are more frequently searched than other items and hence caching has been enabled for the given table.



1. The row is searched in the row cache first
2. If the row is available in the row cache (hit), it is immediately returned to the client
3. Else, in case of a row cache miss, check whether the given row key is cached in the key cache
4. Use the partition key cache to
   1. directly locate the corresponding row on disk
   2. load the row into the Memtable
   3. cache it in the row cache for processing further requests

**Points to remember**

* Disable caching for low-demand data or extremely long rows in the table
* Logically separate heavily-read data into discrete tables
* Each key cache hit saves 1 seek and each row cache hit saves 2 seeks at the minimum, sometimes more
* The row cache saves even more time than key cache, but must contain the entire row, so it is extremely space-intensive. It's best to only use the row cache if you have hot rows (frequently accessed and also small sized partitions)

**Using CQL to configure caching**

* Caching is either enabled or disabled by setting the table caching property in the **WITH**clause of the table definition as shown below:

CREATE TABLE <table name> (column definition)  
WITH caching = {'keys': '.. ', 'rows\_per\_partition' : '…. '};

**Example**:

* Enabling caching for the existing table **Items\_Searched\_By\_Customers**

ALTER TABLE item\_searched\_by\_customers  
WITH caching = {'keys': 'ALL', 'rows\_per\_partition':'120'};

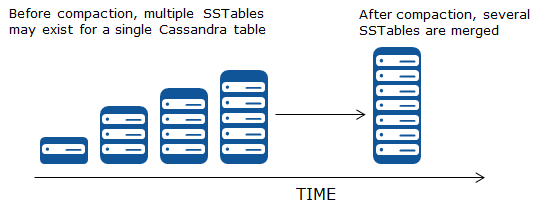
* Disabling caching for the table

ALTER TABLE item\_searched\_by\_customers  
WITH caching = {'keys': 'NONE'};

**Tune compaction.**

As write operations are continuously being performed, the number of SSTables stored on disk for a given Cassandra table keep increasing. This occupies lot of disk space. Also, read operations may require many seeks to return a result leading to performance deterioration as SSTables accumulate. This can be solved using compaction.

**Compaction**is used to merge several SSTables accumulated over time to free memory resources and improve performance. This is done by merging keys, combining columns, evicting tombstones (deleted rows/columns), and creating a new index in the merged SSTable.



**Using CQL to configure compaction**

* For non-existing tables, use the compaction option while creating the table

CREATE TABLE <table name> (column definition)  
WITH **compaction**= {'class': '**<compaction strategy>**'…….};

* For existing tables, use the ALTER TABLE command with compaction option

ALTER TABLE <table name>  
WITH **compaction**= {'class': '**<compaction strategy>**'…….};

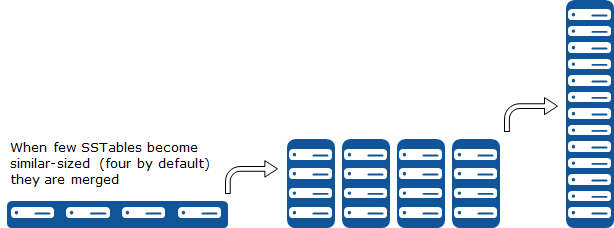
**Types of compaction strategies**

Compaction strategies are used to decide when compaction needs to be performed. It can be configured based on the characteristics of your data.

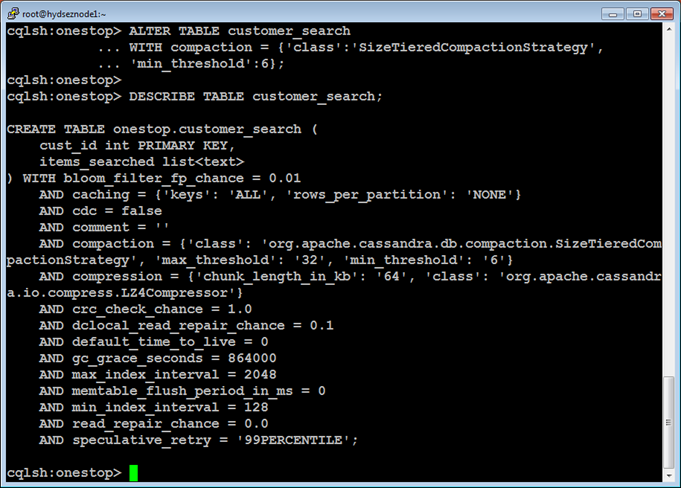
* SizeTieredCompactionStrategy (STCS)
* DateTieredCompactionStrategy (DTCS)
* LeveledCompactionStrategy (LCS)

**SizeTieredCompactionStrategy (STCS)**

In this strategy, size is the deciding factor. That is, when enough similar-sized SSTables are present (four by default), Cassandra will merge them.



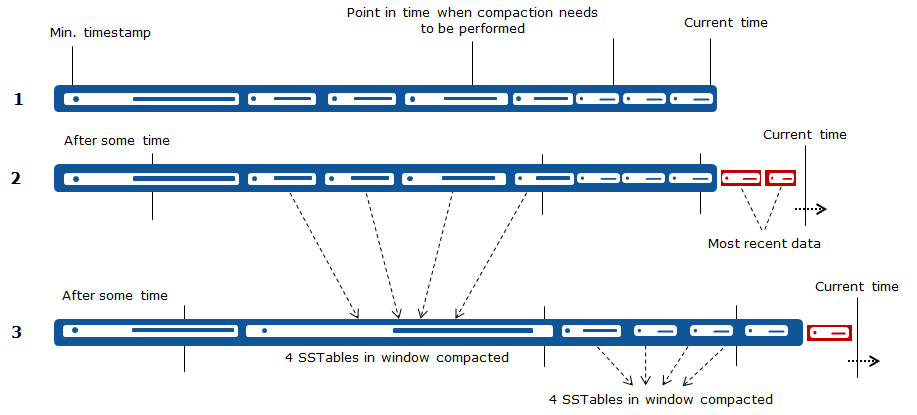
**Setting SizeTieredCompactionStrategy using CQL**



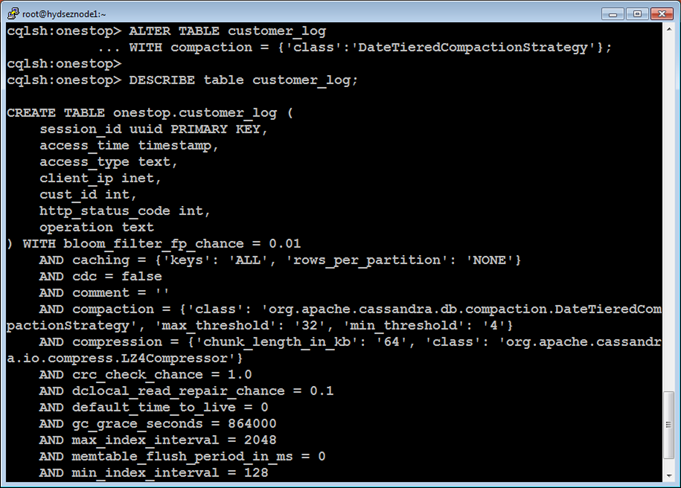
**DateTieredCompactionStrategy (DTCS)**

This strategy is mainly useful in use cases such as time series data. It groups SSTables into windows based on how old the data in the SSTable is so that new and old data don’t get mixed. The base\_time\_seconds option sets the size of the initial window and defaults to 1 hour. Time windows move as time passes. For example, the log data that was written in the last hour will be in that first window, and will be compacted with data from the same window.

Note: The size of the compaction window is configurable.



**Setting DateTieredCompactionStrategy using CQL**



Optionally, you can set the other configuration properties corresponding to DateTieredCompactionStrategy as follows:

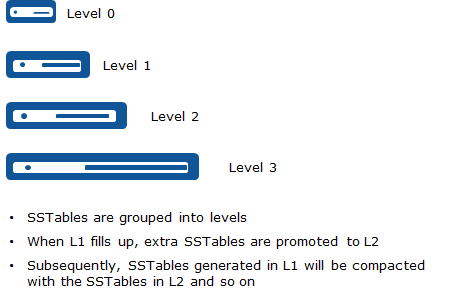
* timestamp\_resolution: Clients can set the timestamp either to MICROSECONDS (default) or MILLISECONDS
* base\_time\_seconds: Refers to the size of the first window, defaults to 3600 seconds (1 hour). The rest of the windows will be min\_threshold (default 4) times the size of the previous window
* max\_sstable\_age\_days: Used to specify the number of days after which these SSTables need not be compacted (contains very old data). For example, a year ago stock prices may not be relevant for us today

**LeveledCompactionStrategy (LCS)**

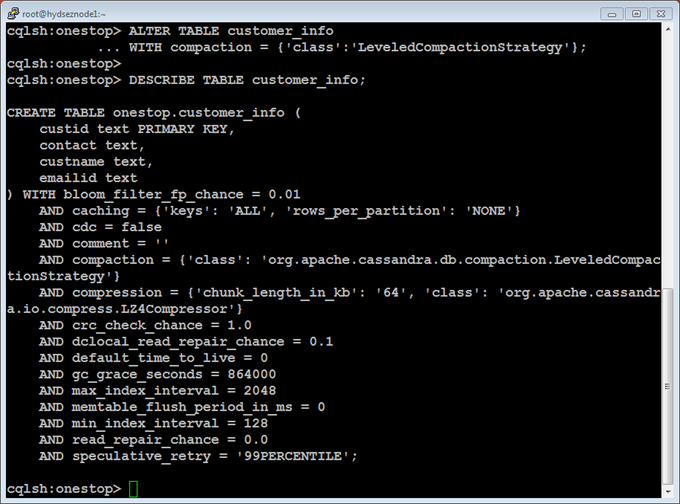
Leveled compaction creates SSTables of a fixed, relatively small size (5MB by default), that are grouped into “levels”. Within each level, SSTables are guaranteed to be non-overlapping. Each level is ten times as large as the previous one.

In Leveled compaction, new SSTables are added to the first level, L0, and immediately compacted with the SSTables in the next level L1. When L1 fills up, extra SSTables are promoted to L2. Subsequently, SSTables generated in L1 will be compacted with the SSTables in L2 and so on.

Leveled compaction guarantees that 90% of all reads will be satisfied from a single SSTable. At most 10% of space will be wasted by obsolete rows. Only enough space for 10x the SSTable size needs to be reserved for temporary use by compaction.

****

**Setting LeveledCompactionStrategy using CQL**

****

**compaction strategy**

**Which compaction strategy to use?**

**For time series data**

With respect to eCommerce customers, you may want to read the most recent purchases or item searches made.

But, issues with STCS and LCS are that:

* These don’t take into consideration when data was actually written
* Also, they mix new and old data

On the other hand, DTCS provides an option to stop compacting data that is old and rarely read thereby reducing the write amplification cost

Also, DTCS reduces the number of SSTables required to refer to for reads

**For heavy reads**

* LCS gives a great read performance but at a bigger write amplification cost as you need to re-compact data a lot

**For frequent writes**

* Use STCS

**Configuring compaction threshold**

* Compaction threshold refers to the number of SSTables that are in the queue to be compacted. By default, minimum number is 4 and maximum is 32
* If the threshold value is too small, you end up performing many frequent unnecessary compactions
* On the other hand, if the number is too large,  Cassandra ends up spending lot of resources performing many compactions at once leaving fewer resources available for clients
* The compaction threshold is set per table

**When to use and when not to use Cassandra**

ACID properties

* Don't use Cassandra if your data requires strict ACID properties to be applied (for example, Financial data)
* Don't use Cassandra if your application is transactional in nature (rollback, commit)
* In order to handle ACID properties, you need to write lot of application code that would be complex and tedious
* Also, time to market would be hit badly

Strong consistency

* Since Cassandra follows a peer to peer architecture where you can write to any available node, it is not the right choice if immediate consistency is your requirement as it supports eventual consistency

Read

* Write conflicts are resolved using latest time-stamped version of the data in Cassandra
* Therefore, a read before a write is an anti-pattern which results in race conditions and latency

Secondary indexes

* Secondary index entries are stored locally at the nodes containing the corresponding partition key
* They require querying most nodes in the cluster even if only a handful of rows is returned. Hence, use sparingly
* Also, Cassandra is not suitable where there is a need for multiple secondary indexes

SAN (Storage Area Network)

* Cassandra performs better without the use of SAN
* Acts as a SPoF (Single Point of Failure) on a SAN
* This also incurs a huge cost whereas Cassandra was designed for commodity hardware

Row cache

* Row cache stores full rows. So if your query tries to retrieve a fewer columns, storing the entire row/partition in the cache would waste resources
* Also, writes invalidate the entire row in the cache, so frequent writes would lead to low performance
* So don't use row caches if your application has dynamic queries on different columns (for e.g., searching the product catalog based on various criteria and ordering preferences)

Compression

* Compression algorithms are super fast and optimize disk storage
* During reads, Cassandra uses some fast paths for minimizing disk seeks
* But then compression disables these fast paths thus leading to performance degradation

Compaction

* Tombstones are row or column data that have been marked for deletion
* Generally these don't get deleted immediately, so need to run major compaction to remove unwanted data (tombstones)
* Avoid performing repairs while compaction is in progress as this leads to retaining tombstones for long even when this data is not required
* For example, perform compaction during week days and run repairs during weekends so that they don't overlap

Joins

* Cassandra does not support joins
* Also, it has limited support for performing aggregation
* Denormalize with User Defined Types (UDTs). For example, modeling one-to-many relationship between the customer order and items contained within it

CREATE TYPE ItemData  
(item\_id text,  
item\_desc text,  
qty int,  
price float  
);

CREATE TABLE OrderItems  
(order\_id uuid,  
order\_date timestamp,  
cust\_id text,  
shipping\_addr address,  
items frozen<set<itemdata>>,  
PRIMARY KEY (order\_id, order\_date)  
) WITH CLUSTERING ORDER BY (order\_date DESC);

Bucketing

* Use bucketing (composite partition key) for time series data so as to avoid hot spots (very large partitions). For example, customers searching for items can be modeled as shown below

CREATE TABLE search\_list  
(cust\_id text,  
category text,  
searched\_at timestamp,  
item\_searched text,  
PRIMARY KEY ( (cust\_id, category), searched\_at)  
);

Queues

* Also don't use Cassandra for implementing queues

Summarizing anti-patterns in Cassandra

* Data modeling dynamic schema
* Heavy writes on the same column
* Queue implementation
* CQL null values

Summarizing where to use Cassandra

* Sensor data
* Time series data
* Anti fraud
* Account activation (use TTL within which the new account created is to be validated)

**Cassandra best practices**

Here, you will see some of the best practices that may be followed for ensuring better performance and high availability.

**Size of the Cassandra database**

* Since Cassandra is based on Java API, RAM should not be less than 8 GB in production and 4 GB in development (minimal requirement for JVM to run)
* Also, since Cassandra loads data into memtables in memory and uses row cache and key cache for faster access, it is ideal for each node to have RAM between 32 GB and 512 GB
* Also, Cassandra is CPU-intensive, therefore, 16 cores are recommended (with not less than 2 cores for development)

**Choosing the replication factor (RF)**

* Replicate across data centers for higher availability and reliability
* Ideal replication factor in production is 3
* Lower RF means data can be lost, but useful for queries with higher consistency levels (CONSISTENCY ALL)
* On the other hand, higher RF means ensuring reliability, but queries may get delayed for higher consistency levels (coordinator needs to wait for responses from majority or all of the replicas)

**Cassandra versus other NoSQL databases**

Here you will see some of the differences between Cassandra and other most popular NoSQL databases.

| **Characteristics** | **Cassandra** | **MongoDB** | **HBase** | **Riak** | **Redis** | **Couchbase** |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | Wide column family | Document-oriented | Wide column family | Key/Value store | Key/Value store | Document-oriented |
| **Architecture** | Peer to Peer | Master/Slave | Master/Slave | Master-less | Master/Slave | Peer to Peer |
| **Written in** | Java | C++ | Java | Erlang and C | C | Erlang and C |

**Cassandra Vs HBase**

Cassandra and HBase are both wide column family stores. However, they differ in several aspects that is illustrated in the table below.

| **Characteristics** | **HBase** | **Cassandra** |
| --- | --- | --- |
| **Architecture** | Master/Slave | Peer to Peer |
| **SPoF (Single Point of Failure)** | Master monitors all region servers across the cluster | De-centralized master-less architecture, so no SPoF |
| **High velocity writes** | Supports single-write master, therefore becomes a bottleneck with heavy writes | Can read/write anywhere, so supports high velocity random reads and writes on any available node |
| **Read/Write workloads** | Optimized for reads | Optimized for writes |
| **CAP theorem** | Supports Consistency and Partition tolerance (CP) | Supports Availability and Partition tolerance (AP) |
| **Query Pattern** | Supports row-scans and so well suited for range based scans | Does not support range based row-scans, but has excellent single-row read performance |
| **Partitioning Strategy** | Based on Ordered partitioning | Supports Random partitioning by default |
| **Suitable for** | Data warehousing and large scale data processing and analysis | Real time transaction processing and serving of interactive data |
| **Real use case** | Facebook Messenger | Twitter |

**Integrate Cassandra with Hadoop.**

**Why integrate Cassandra with Hadoop?**

Cassandra is highly scalable and available. Hadoop's map reduce programming is very strong. Hence, integrate Cassandra with Hadoop to improve performance which in turn drives great business value for the organization.

In general, integrating Cassandra with Hadoop has the following advantages:

* Integrated workloads
* No Single Point of Failure (as Cassandra has a master-less architecture)
* High availability
* Easy to deploy

**Cluster setup**

The Cassandra cluster and Hadoop cluster can be setup in various ways as listed below:

* **Dedicated infrastructures**

Configure Cassandra on a set of machines different from those used for the Hadoop cluster. Doing so gives you a flexibility of executing time-series based applications on the Cassandra cluster while batch analytics can be done using the Hadoop cluster.

* **Partially integrated infrastructures**

Since Cassandra is highly scalable, set up Cassandra all the nodes in the cluster and configure Hadoop only on a part of the Cassandra cluster**.**

* **Fully integrated infrastructures**

Configure Cassandra and Hadoop on the same set of nodes in the cluster. Advantages of doing so are:

* Shared resources
* Nodes are reusable
* No SPOF

**Solution 1**

1. Export Cassandra table data to delimited files using the **COPY**command already discussed
2. Using HDFS **put**command, load this onto Hadoop Distributed File System
3. Use Hadoop **Map Reduce programming**, **Pig**or **Hive**to process this data in parallel across the nodes in the Hadoop cluster

This approach is suitable for batch processing.

**Solution 2**

1. Use java based **Map Reduce API** to read the input from Cassandra based on implementations of **InputSplit**, **InputFormat**and **RecordReader**
2. Process the data read from Cassandra table using a Mapper and Reducer
3. Write the map reduce result to Cassandra table

Cassandra's Hadoop support implements the same interface as HDFS to achieve input data locality.

**Integrate Cassandra with Spark.**

**Why to integrate Cassandra with Spark?**

By now, you know that Cassandra is a highly scalable NoSQL database apt for write heavy workloads. But it falls back when performing aggregations and data analysis. Because of its limited querying options, this massive data is not optimally utilized. When integrated with Cassandra, Spark resolves all these issues. It is a good choice for storing (Cassandra) and processing (Spark) large scale data at lightening speed.

**Use of Spark**

Spark is used for high speed in-memory distributed processing. It is 100X times faster than Hadoop MapReduce in memory, or 10X faster on disk. Though written in Scala, Spark has APIs for Java, Python, R etc.

In Spark, data is represented using RDD ( Resilient Distributed Datasets) that are distributed across the Spark cluster and replicated among nodes for fault tolerance (similar to partitioning in Cassandra).

Also, it supports server-side filters (WHERE clause) that Cassandra can use, to extract only that data required for processing.

You have learnt to model and query the data in Cassandra database using CQL.  The course concludes here.

**In this course, you have learnt the following:**

* Cassandra Vs Relational databases
* Installation and configuration of Apache Cassandra
* Designing data model using CQL
* Working with CRUD operations in Apache Cassandra
* Use of indexes and materialized views
* Working with native functions and user defined functions
* The NoSQL ecosystem
* Key components of Cassandra
* How to design the data model for Cassandra
* Tuning resources for improved performance
* When to use and when not to use Cassandra
* Cassandra best practices
* Integration of Cassandra with Hadoop
* Integration of Cassandra with Spark