Cassandra Data Types and CQL

1. Introduction

Sometimes, we need a quick reference guide to get started in our learning path. In particular, a cheat sheet is a document that contains all the critical information.

In this article, we'll learn the essential concepts of Cassandra query language (CQL) and how to apply them using a cheat sheet that we'll build along the way.

2. Cassandra at a Glance

Apache Cassandra is an open-source, NoSQL, and distributed data storage system. This means instead of being able to live only on one server, it spreads across multiple servers. It's also known for its high availability and partition tolerance.

To put it another way, the design of the Cassandra database is inspired by the “AP” of the CAP theorem.

Furthermore, **Cassandra is a masterless architecture, is massively scalable, and above all, provides easy fault detection and recovery**.

3. Data Types

Generally, Cassandra supports a rich set of data types. These include native types, collection types, user-defined types, and tuples, together with custom types.

3.1. Native Types

The native types are the built-in types and provide support to a range of constants in Cassandra.

To begin with, a string is a very popular datatype in the programming world.

CQL offers four different datatypes for strings:

Data Type	Constants Supported	Description
ascii	*string*	ASCII character string
inet	*string*	IPv4 or IPv6 address string
text	*string*	UTF8 encoded string
varchar	*string*	UTF8 encoded string

A boolean has one of two possible values, either *true* or *false*:

Data Type	Constants Supported	Description
boolean	*boolean*	*true* or *false*

Using the blob data type, we can store images or multimedia data as a binary stream in a database:

Data Type	Constants Supported	Description
blob	*blob*	Binary large object. Arbitrary bytes

Duration is a three-signed integer that represents months, days, and nanoseconds:

Data Type	Constants Supported	Description
duration	*duration*	A duration value

Cassandra offers a wide range of data types for integer data:

Data Type	Constants Supported	Description
tinyint	*integer*	8-bit signed int
smallint	*integer*	16-bit signed int
int	*integer*	32-bit signed int
bigint	*integer*	64-bit signed long

varint	*integer*	A variable precision signed integer
counter	*integer*	Counter column (64-bit signed)

For integer and float, we have three data types:

Data Type	Constants Supported	Description
decimal	*integer, float*	Variable precision decimal
double	*integer, float*	64-bit floating-point
float	*integer, float*	32-bit floating-point

For date- and time-related needs, Cassandra provides three data types:

Data Type	Constants Supported	Description
date	*integer, string*	A date value (without time)
time	*integer, string*	A time value (without date)
timestamp	*integer, string*	A timestamp (with date & time)

Generally, we have to avoid collision while using the INSERT or UPDATE commands:

Data Type	Constants Supported	Description
uuid	*uuid*	A UUID (any version)
timeuuid	*uuid*	A version 1 UUID

- **uuid -** uuids ("UUID" stands for "universally unique identifier"). It is 128-bit Hex value and supposed to be unique even beyond the table. Convenience function uuid().
- **timeuuid -** Unique identifier. Based on MAC address, system time and a sequence number. Prevent duplicates. Convenience functions are now(), dateOf(), unixTimestampOf().

3.2. Collection Types

When a user has multiple values against one field in a relational database, it's common to store them in a separate table. For example, a user has numerous bank accounts, contact information, or email addresses. Therefore, we need to apply joins between two tables to retrieve all the data in this case.

Cassandra provides a way to group and store data together in a column using collection types.

Let's quickly look at those types:

- *set* – unique values; stored as unordered
- *list* – can contain duplicate values; order matters
- *map* – data stores in the form of key-value pairs

Data Type	Constants Supported	Description
frozen	Tuples, collections, user defined types	stores Cassandra types
list		Collection of elements
map		JSON style collection of elements
set		Collection of unordered unique elements
tuple		A group of 2,3 fields

3.3. User-Defined Types

User-defined types give us the liberty to attach multiple data fields in a single column:

```
CREATE TYPE student.basic_info (
  birthday timestamp,
```

```
  race text,
  weight text,
  height text
);
```

3.4. Tuple Type

A tuple is an alternative to a user-defined type. It's created using angle brackets and a comma delimiter to separate the types of elements it contains.

Here are the commands for a simple tuple:

```
-- create a tuple
CREATE TABLE subjects (
  k int PRIMARY KEY,
  v tuple<int, text, float>
);
-- insert values
INSERT INTO subjects  (k, v) VALUES(0, (3, 'cs', 2.1));
-- retrieve values
SELECT * FROM subjects;
```

4. Cassandra CQL Commands

Let's look at several categories of CQL commands.

4.1. Keyspace Commands

The first thing to remember is that a keyspace in Cassandra is much like a database in RDBMS. It is an outermost container of data that defines the replication strategy and other options, particularly for all the keyspace tables. With this in mind, a good general rule is one keyspace per application.

Let's look at the related commands:

Command	Example	Description
CREATE keyspace	CREATE KEYSPACE *keyspace_name* WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 2};	To create a keyspace.
DESCRIBE keyspace	DESCRIBE KEYSPACES;	It will list all the key spaces.
USE keyspace	USE *keyspace_name*;	This command connects the client session to a keyspace.
ALTER keyspace	ALTER KEYSPACE *keyspace_name* WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 } AND DURABLE_WRITES = false;	To alter a keyspace.
DROP keyspace	DROP KEYSPACE *keyspace_name*;	To drop a keyspace.

4.2. Table Commands

In Cassandra, a table is also referred to as a column family. We already know the importance of a primary key. However, it is mandatory to define the primary key while creating the table.

Let's review these commands:

Command	Example	Description
CREATE table	CREATE TABLE table_name (column_name UUID PRIMARY	To create a table.

	KEY, column_name text, column_name text, column_name timestamp);	
ALTER table	ALTER TABLE table_name ADD column_name int;	It will add a new column to a table.
ALTER table	ALTER TABLE table_name ALTER column_name TYPE datatype;	We can change the data type of an existing column.
ALTER table	ALTER TABLE table_name WITH caching = {'keys' : 'NONE', 'rows_per_partition' : '1' };	This command helps to alter the properties of a table.
DROP table	DROP TABLE table_name;	To drop a table.
TRUNCATE table	TRUNCATE table_name;	Using this, we can remove all the data permanently.

4.3. Index Commands

Instead of scanning a whole table and waiting for results, we can use indexes to speed up queries. However, we must remember that **the primary key in Cassandra is already indexed. Therefore, it cannot be used for the same purpose again.**

Let's look at the commands:

Command	Example	Description
CREATE index	CREATE INDEX *index_name* on *table_name* (*column_name*);	To create an index.
DELETE index	DROP INDEX IF EXISTS *index_name*;	To drop an index.

4.4. Basic Commands

These commands are used to read and manipulate the table values:

Command	Example	Description
INSERT	INSERT INTO *table_name* (*column_name1*, *column_name2*) VALUES(*value1*, *value2*);	To insert a record in a table.
SELECT	SELECT * FROM *table_name*;	The command is used to fetch data from a specific table.
WHERE	SELECT * FROM *table_name* WHERE *column_name=value*;	It filters out records on a predicate.
UPDATE	UPDATE *table_name* SET *column_name2=value2* WHERE *column_name1=value1*;	It is used to edit records.
DELETE	DELETE *identifier* FROM *table_name* WHERE *condition*;	This statement deletes the value from a table.

4.5. Other Commands

Cassandra has two different types of keys: partition key and clustering key. A partition key indicates the node(s) where the data is stored.

In comparison, the clustering key determines the order of data within a partition key:

Command	Example	Description
ORDER BY	SELECT * FROM *table_name* WHERE *column_name1 = value* ORDER BY *cloumn_name2* ASC;	For this, the partition key must be defined in the WHERE clause. Also, the ORDER BY clause represents the clustering column to use for ordering.
GROUP BY	SELECT *column_name* FROM *table_name* GROUP BY *condition1*, *condition2*;	This clause only supports with Partition Key or Partition Key and Clustering Key.

LIMIT	SELECT * FROM table_name LIMIT 3;	For a large table, limit the number of rows retrieved.

5. Operators

Cassandra supports both arithmetic and conditional types of operators. Under the arithmetic operators, we have +, -, *, /, %, and – (unary) for addition, subtraction, multiplication, division, reminder, and negation, respectively.

The WHERE clause is significant in Cassandra. The conditional operators are used in this clause with certain scenarios and limitations. These operators are CONTAINS, CONTAINS KEY, IN, =, >, >=, <, and <=.

6. Common Functions

Without a doubt, functions, either aggregate or scalar, play an essential part in transforming values from one to another. For this reason, Cassandra offers several native functions in both categories.

Let's look at those functions:

- Blob conversion functions
- UUID & Timeuuid functions
- Token function
- WRITETIME function
- TTL function
- TOKEN function
- MIN(), MAX(), SUM(), AVG()

Along with these native functions, it also allows users to define the functions and aggregates.

7. Conclusion

In this short article, we've seen what the building blocks of Cassandra's query language are. First, we studied the data types it supports and how to define them. Then, we looked at common commands to perform database operations. Finally, we discussed the operators and functions of the language.