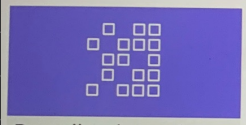
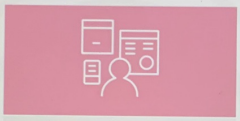
CQL Data Types

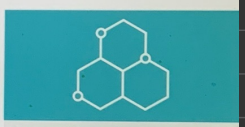
What you will learn



Describe the main data types in CQL



Explain how to use these data types when defining a Cassandra table



Describe the role of collection data types and user-defined data types

CQL data types

Data types applied when declaring Cassandra tables:

Built-in data types : basic predefined in Cassandra

Data Type	Data Type
Ascii	Int
Boolean	Text
Blob	Timestamp
Bigint	Timeuuid
Decimal	Tinyint
Double	Uuid
Float	Varchar

stands for Binary Large Object

Blob: Are arbitrary bytes. A blob type is suitable for storing a small image or short string (1MB)

audio

Bigint: Are used for 64-bit signed long integers. This data type stores a higher range of integers compared to the "int" data type

Varchar: Is used for strings, and represents a UTF8 encoded strings

Collection data types

- Collections are a way to group and store data together
- Example: A user has multiple email addresses
  - In a relational database: A many-to-one joined relationship between a 'users' table and an 'email' table
  - In Cassandra: No joins; Cassandra stores all the data in a collection column in the 'users' table
- Data for collection storage should be limited: No unbounded growth
  - Not suitable for storing sent messages or sensor events stored every second

↳ use a table with a compound primary key where data is stored in the clustering columns

Collection data types

collection one or more elements in the table

represent one or more sorted elements in a table

Lists

- When the order of the elements needs to be maintained
- Example: Entries in a log

value is to be stored multiple times

Maps

pairs

- Key:Value
- Example: Entries in a journal (date:text)

Sets

is stores a group of elements

- When elements are unique and do not need to be stored in a specific order
- Example: Email addresses

Collection data types: List

Users table
PRIMARY KEY(userid)

stores all the users in a single column

use list type, because we want to preserve the order of the jobs

```
USE intro_cassandra;
ALTER TABLE users ADD jobs list<text>;
```

add → beginning

```
UPDATE users SET jobs = ['Walmart'] + jobs where username = 'Alaind@gmail.com'; //add the last job change to the list
```

or at the end

```
UPDATE users SET jobs = jobs + ['Netflix'] where username = 'Alaind@gmail.com'; //add the last job change to the list
```

in specific position

```
UPDATE users SET jobs[0]='Reiss' where username = 'Alaind@gmail.com'; //replaces Walmart with Reiss (lists start from 0)
```

Collection data types: List

Username	age	jobs	occupation
Daveg@gmail.com	43	null	engineer
Elaine@yahoo.com	60	null	producer
dan@gmail.com	46	null	banker
Alaind@gmail.com	32	['Reiss', 'Netflix']	actor
Peterd@gmail.com	32	null	producer
Moirad@yahoo.com	35	null	dj

```
UPDATE users SET jobs = jobs - ['Reiss']
  WHERE username = 'Alaind@gmail.com';
SELECT * FROM users
  WHERE username = 'Alaind@gmail.com';
```

User-defined data types (UDTs)

- Can attach multiple data fields, each named and typed, to a single column
- The fields used to create a UDT may be any valid data type, including collections and other existing UDTs
- Once created, the user can alter, verify, and drop a field or the whole data type
- Once created, UDTs may be used to define a column in a table

User-defined data types (UDTs)

```
CREATE TYPE address (
Street text,
Number int,
Flat text);
```

new data type

```
CREATE TABLE users_w_address(
Userid int,
Location address,
Primary key (userid));
```

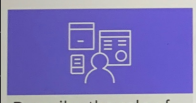
use it to define a column

```
INSERT INTO users_w_address(userid,location) VALUES (1, {street : 'Third', number : 34, flat : '34C'}); //insert data

DROP TYPE address; //we can drop a type
```

Apache Cassandra Keyspace Operations

What you will learn



Describe the role of keyspaces in Apache Cassandra

Describe the replication factor and replication strategy

Create, modify, and delete a keyspace

Keyspaces

- Must define before creating tables ← no default
- Can contain any number of tables, and a table belongs to only one keyspace
- Replication is specified at the keyspace level
- Specify the replication factor during the creation of a keyspace and is modifiable ← later when it's deployed

Create a keyspace

replicates its data, AKA table's partitions 3 times between the cluster node

```
CREATE KEYSPACE intro_cassandra WITH REPLICATION = {
'class' : 'NetworkTopologyStrategy',
'datacenter1' : 3,'datacenter2' : 2 };
```

Replication Strategy

Replication Factor

set at each datacenter level

```
DESCRIBE KEYSPACES;
DESCRIBE intro_cassandra;
```

To check if the keyspace has been created

Data replication

- Replication Factor
  - Number of replicas placed on different nodes in the cluster
- Replication Strategy
  - Which nodes are going to house the replicas ← be located ← cluster
- Replicas
  - All replicas are equally important with no primary or secondary replicas
  - Replication factor should not exceed the number of cluster nodes

- After data is initially distributed according to partition key based and pre-allocation, then
- data is also replicated according to Replication Factor and Strategy

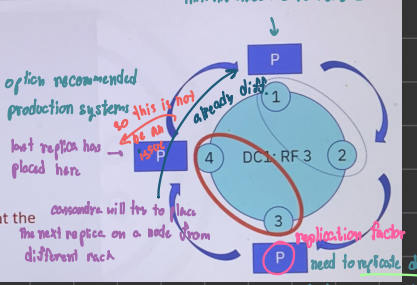
Single DC cluster: Rep factor 3

assume: this P partition has been initially allocated to node 1

```
CREATE KEYSPACE intro_cassandra
WITH REPLICATION = { 'class' :
'NetworkTopologyStrategy',
'datacenter1' : 3 };
```

the only option recommended for production systems

class replication level

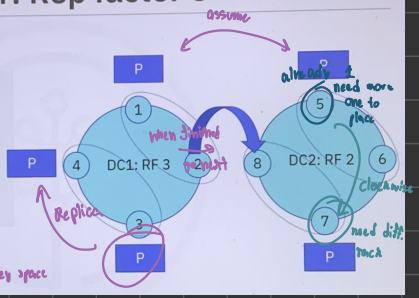



```
//Node 1 and 2 are in the same rack so that the
second replica is placed in node 3
```

Multiple DC cluster: Rep factor 5

```
CREATE KEYSPACE intro_cassandra WITH
REPLICATION = { 'class' :
'NetworkTopologyStrategy',
'datacenter1' : 3, 'datacenter2' : 2 };
```

2 data centers
5 replication factors
use call to create key space



Alter a keyspace

```
ALTER KEYSPACE intro_cassandra
        WITH REPLICATION = {
'class' : 'NetworkTopologyStrategy',
'datacenter1' : 3,
'datacenter2' : 3};
```

Modifies the keyspace replication factor

Replication factor: The number of copies of the data Cassandra creates in each data center

Drop a keyspace

```
DROP KEYSPACE intro_cassandra;
```

- Immediate removal of the keyspace, including objects such as tables, functions, and data the keyspace contains
- Cassandra takes a snapshot before dropping the keyspace

Cassandra Data Modeling and Querying Best Practices

This reading will teach you the best practices for Cassandra data modeling and querying.

Cassandra is an open-source, non-relational, or NoSQL distributed database that is continuously available across multiple data centers and cloud availability zones. It provides a reliable data storage engine for applications with a broad range.

Understanding Cassandra's distributed architecture and data storage principles is crucial for designing data models and crafting efficient queries. Let's examine the best practices for achieving high performance, scalability, and efficiency in Cassandra.

Improved read and write performance

An efficient data model enables data to be requested as a single request, reducing complex joins. However, it also enables Cassandra to retrieve data quickly from distributed storage.

Partition correctly

A good partition key restricts hotspots and disproportionately stores the amount of data on a few nodes. It also avoids selecting high-cardinality or frequently updated fields as partition keys and restricts large partitions.

Data duplication

Data duplication supports query patterns and uses primary keys across multiple tables to optimize queries without compromising performance.

Avoid full table scans

Avoiding full table scans minimizes them and range queries without specifying a partition key. Full table scans are resource-intensive, compromising performance for large datasets.

Specify required columns

The data query from Cassandra specifies the SELECT statement's columns. Obtaining only the required columns reduces the usage of the network bandwidth and minimizes the amount of data transferred between nodes.

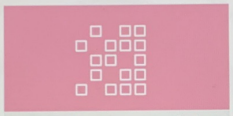
Prefer batch updates

The batch update achieves atomicity and consistency for a group of write operations. However, it minimizes the overhead associated with network communication and coordination between nodes in the Cassandra cluster.

Table Operations

What you will learn

Explain the role of Cassandra tables



Describe some of the properties of Cassandra tables

Create, alter, and delete a table

Logical Entities: Tables and Keyspaces

Table

- Logical entity that organizes data storage at cluster and node level (according to a declared schema)

Keyspace ← create this first, then you can declare a table

- Logical entity that contains one or more tables
- Replication and data centers' distribution is defined at keyspace level
- Recommended one keyspace per application

CREATE TABLE

It's an optional

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name
( column_definition [, ...]
      PRIMARY KEY (column_name [, column_name ...])
[WITH table_options
      | CLUSTERING ORDER BY (clustering_column_name order])


column_name cql_type_definition [STATIC | PRIMARY KEY] [, ...]
```

Dynamic CREAT TABLE example

```
USE intro_cassandra;

CREATE TABLE groups(
groupid int,
group_name text STATIC
username text,
age int,
PRIMARY KEY ((groupid), username));

SELECT * FROM groups; //no data
```

static ↓ it contains only one partition key

Partition key

Static CREATE TABLE example

dynamic table : has both partition and clustering key

```
USE intro_cassandra;
CREATE TABLE users(
   username text PRIMARY KEY,
   occupation text,
   age int;
```

clustering key

```
USE intro_cassandra;
CREATE TABLE users(
   username text,
   occupation text,
   age int,
PRIMARY KEY(username));
```

```
SELECT * FROM users; //there are no entries in our table so the
SELECT will yield no data;
```

Table properties and options

```
DESCRIBE groups;
CREATE TABLE intro_cassandra.groups (
    groupid int,
    username text,
    age int,
    group_name text static,
PRIMARY KEY (groupid, username)
) WITH CLUSTERING ORDER BY (username ASC)
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND compaction = {'class':
    'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
    'max_threshold': '32', 'min_threshold': '4'}
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND memtable_flush_period_in_ms = 0;
```

to order inside the partion

→ can set an expiration time for the data in the table

↳ data will be deleted

↳ not activated

write data → three situations → mem table full
→ commit log full
→ a specific interval of flushing data to disk

ALTER TABLE

- Add new columns
- Drop existing columns
- Rename columns
  - Regular columns
  - Clustering keys
- Changes table properties
- Restriction: Altering PRIMARY KEY columns is not supported
- Restriction: Changing data type of an existing column is not supported

```
ALTER TABLE groups ADD occupation TEXT;

ALTER TABLE groups DROP age;

ALTER TABLE groups RENAME username to
user_name;

ALTER TABLE groups WITH
default_time_to_live=10;
```

only works for regular and clustering keys

TRUNCATE and DROP TABLE commands

- Truncate: Removes all data from a table but not the table's schema definition
- Truncate: Needs Consistency ALL and all replicas available
- Drop: Removes all the data, including schema
- Cassandra takes a snapshot before truncating or dropping the table

```
TRUNCATE TABLE groups;
TRUNCATE groups;
DROP TABLE groups;
```

CRUD Operations - Part 1

How a write is done in Cassandra
↙ at the cluster and node level.

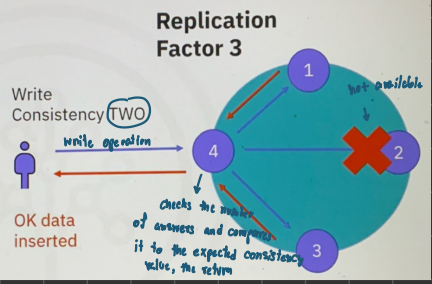
What you will learn

- Describe the Write process in Cassandra
- Explain the INSERT and UPDATE operations
- Explain the role of lightweight transactions

Write operations in Cassandra

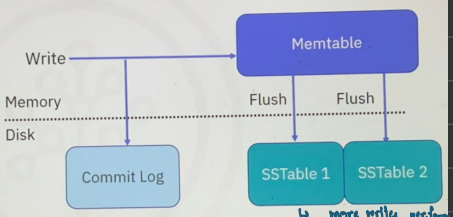
when write occurs at cluster level

- Receiving node is the coordinator for the operation : it will make sure to complete the operation and send the result back to the user
- Writes directed to all partition replicas
- Acknowledgement expected from consistency number of nodes



Write operations in Cassandra

- No reading before writing (by default)
- Node level
  - Writes are stored in memory and later flushed on disk (SSTables)
  - Every flush => new SSTable
  - All disk writes are sequential ones. Data will be reconciled through compaction
- Cassandra attaches a timestamp to every write



Write operations: Node level

↳ more writes performed, the faster the Memtable gets filled and flushes the data on disk

INSERT

can be done record by record

- Insert operations require full primary key
- Cassandra doesn't perform read before write: An INSERT can behave as an UPSERT
  - If we INSERT data in an existing entry, data is UPDATED
- Inserts require a value for all primary key columns but not other columns
  - Only specified columns are inserted/updated
- You can INSERT/UPDATE data with Time-To-Live

↳ we can do this at record level, data will be visible only for a defined time

INSERT

```
INSERT INTO groups(groupid,username,group_name, age) VALUES
(12,'aland@gmail.com','baking',32);

INSERT INTO groups(groupid,username,group_name,age) VALUES
(12,'Elaine@yahoo.com','baking',60);

INSERT INTO groups(groupid,username) VALUES (45,'Moirad@yahoo.com');

INSERT INTO groups(groupid,username,group_name)
VALUES (25,'Johns@gmail.com','vegan cooking') USING TTL 10;

INSERT INTO groups(groupid,username,group_name,age)
VALUES (12,'aland@gmail.com','baking',45);
```

primary key is mandatory

↳ cannot insert data without fully specifying it

↳ after 10 s, data will not be available for query anymore

↳ updated to 45

can use insert as an update, when it is done on existing data

UPDATE

groupid	username	groupname	age
45	Moirad@yahoo.com	null	null
12	Elaine@yahoo.com	baking	62
12	aland@yahoo.com	baking	62

it's a static column, we can update by partition key

```
UPDATE groups SET group_name = 'grilling' WHERE groupid = 45;

UPDATE groups SET AGE = 60 WHERE groupid = 12 and username ='aland@gmail.com';

UPDATE groups SET age = 55, group_name = 'coffee' WHERE groupid = 20 and username =
'bella@yahoo.com';
```

→ update can update non-existing column

↳ update can use as an insert

Lightweight transactions (LWT)

introducing id in insert and update statement

groupid	username	groupname	age
45	Moirad@yahoo.com	null	null
45	bella@yahoo.com	coffee	55
12	Elaine@yahoo.com	baking	60
12	aland@yahoo.com	baking	60

```
UPDATE groups SET AGE = 62 WHERE groupid=12 and username = 'aland@gmail.com' IF
EXISTS; //TRUE, age will be updated

UPDATE groups SET AGE = 63 WHERE groupid=12 and username = 'aland@gmail.com' IF age
= 62; //TRUE, age will be updated

INSERT INTO groups(groupid,username,group_name,age) VALUES
(12,'Elaine@yahoo.com','baking',60) IF NOT EXISTS; //FALSE, no insert
```

IF EXISTS → if record exists

exists, so not insert

Lightweight transactions (LWT)

Lightweight transactions are at least four times slower than the normal INSERT/UPDATE in Cassandra. Use them sparingly in your application.

```
UPDATE groups SET AGE = 62 WHERE groupid=12 and username = 'aland@gmail.com' IF
EXISTS; //TRUE, age will be updated

UPDATE groups SET AGE = 63 WHERE groupid=12 and username = 'aland@gmail.com' IF age
= 62; //TRUE, age will be updated

INSERT INTO groups(groupid,username,group_name,age) VALUES
(12,'Elaine@yahoo.com','baking',60) IF NOT EXISTS; //FALSE, no insert
```

CRUD Operations - Part 2

What you will learn

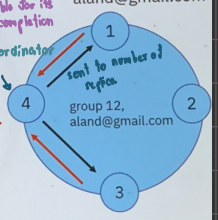
- Describe the READ process in Cassandra clusters
- Explain the Do's and Don'ts of SELECT statements
- Describe how to delete data in Cassandra

How to read

Read operations in Cassandra

- Receiving node is the coordinator for the operation
- Reads are directed only to the number of replicas required for consistency
- Inconsistencies between contacted nodes are repaired during the Read process

Replication Factor 3
Read Consistency 2
only 2 nodes of three replica will be contacted
responsible for its completion
this node become coordinator of the read op
read operation
Data
then result will be sent back
sent to number of replica
group 12, aland@gmail.com



group 12, aland@gmail.com

READ/SELECT rules in Cassandra

- Start your query using the Partition Key (always start with)
  - Limit Reads to specific nodes containing your data
- Follow the order of your Primary Key columns

Ex
Primary Key(PartitionKey, ClusteringKey1, ClusteringKey2
specific value
* work: Partition key only support equal(=) and IN operation

```
SELECT * FROM table WHERE PartitionKey = ;
SELECT * FROM table WHERE PartitionKey IN (,); //list of values
SELECT * FROM table WHERE PartitionKey =  AND ClusteringKey1 = ;
SELECT * FROM table WHERE PartitionKey =  AND ClusteringKey1
=  AND ClusteringKey2 =  ; ← will read 1 record
SELECT * FROM groups;  // not okay performance wise
SELECT * FROM groups WHERE clustering key1 = ; // will not work
SELECT * FROM groups WHERE regular_column = ; // will not work
```

OK
work but performance will be really bad
NOT OK
ERROR*

* Unless ALLOW FILTERING

Secondary indexes

```
USE intro_cassandra;
SELECT * FROM groups WHERE groupid=12;
SELECT * FROM groups WHERE groupid=12 and username='aland@gmail.com';
```

start then

```
SELECT * FROM groups WHERE age = 12; //will not work
CREATE INDEX ON groups(age);
SELECT * FROM groups WHERE age = 12; //will work
```

```
SELECT * FROM groups WHERE groupid=12 AND age = 12;
```

Groups
PRIMARY KEY (groupid,username)

A best practice is to use indexes inside a partition

Always start your queries with the partition key

DELETE

- Record
- Cell
- Range
- Partition

groupid	username	group_name	age
45	Moriad@yahoo.com	grilling	Null
20	bella@yahoo.com	coffee	55
~~12~~	~~elaine@gmail.com~~	~~baking~~	~~60~~
12	aland@gmail.com	baking	32
~~12~~	~~peterd@gmail.com~~	~~baking~~	~~32~~

```
DELETE FROM groups WHERE groupid=12 AND
username='Elaine@yahoo.com';
DELETE age FROM groups WHERE groupid=12
AND username='aland@gmail.com';
SELECT * FROM groups;
DELETE FROM groups where groupid=12;
```

delete all group id=12

DELETE

- Deleting data in distributed systems is trickier than in relational databases
- Especially in peer-to-peer databases like Cassandra
- Reads and writes can be directed to any of partition's replicas, so there's no primary node for a writes or reads
- Cassandra marks all deleted items with a "tombstone" containing the time of the delete operation

↳ when an item has been marked with a tombstone. its data is not visible any longer to queries

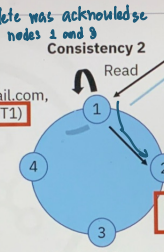
in delete has a special value to indicate that data has been deleted and the time of delete it is ↘

Tombstones

- A special value to indicate data has been deleted + time of delete
- Tombstones prevent deleted data from being returned during reads
- Tombstones are deleted at 'gc_grace_seconds' during compaction process

10 days by default

delete was acknowledge by nodes 1 and 3
Consistency 2
Read
primary key
group 12, aland@gmail.com, DeletedAt (T1)
No Data with time T1
initial 1 the insert done T0
group 12, aland@gmail.com, (T0)
group 12, aland@gmail.com, DeletedAt (T1)
delete when configurable period has elapsed



it is

Term	Definition
Cluster	A group of interconnected servers or nodes that work together to store and manage data in a NoSQL database, providing high availability and fault tolerance.
Consistency	In the context of CAP, consistency refers to the guarantee that all nodes in a distributed system have the same data at the same time.
Dynamic table	A dynamic table allows flexibility in the columns that the database can hold.
Keyspace	A keyspace in Cassandra is the highest-level organizational unit for data, similar to a database in traditional relational databases.
Lightweight transactions	Lightweight transactions provide stronger consistency guarantees for specific operations, though they are more resource-intensive than regular operations.
Partition key	The partition key is a component of the primary key and determines how data is distributed across nodes in a cluster.
Primary key	The primary key consists of one or more columns that uniquely identify rows in a table. The primary key includes a partition key and, optionally, clustering columns.
Replication factor	The replication factor specifies the number of copies of data that should be stored for fault tolerance.
Replicas	Replicas in Cassandra refer to the copies of data distributed across nodes.
Replication strategy	The replication strategy determines how data is copied across nodes.
Secondary indexes	Secondary indexes allow you to query data based on non-primary key columns.
Static table	A static table has a fixed set of columns for each row.
Table	A table is a collection of related data organized into rows and columns.