

Lab Overview

In this lab you will be introduced to work with Cassandra API through Azure portal and CQLSH (Cassandra Shell). You will learn to create and manage data, load and insert data, query data, migrate data using Cosmos DB Cassandra API, while learning specific Cassandra topics.

Documentation

- [Introduction to the Azure Cosmos DB Cassandra API](#)
- [Apache Cassandra features supported by Azure Cosmos DB Cassandra API \(Wire protocol support\)](#)
- [Frequently asked questions about the Cassandra API for Azure Cosmos DB](#)
- [Elastically scale an Azure Cosmos DB Cassandra API account](#)
- [Migrate your data to Cassandra API account in Azure Cosmos DB](#)
- [Azure Cosmos DB Cassandra API Overview \(video\)](#)
- [Apache Cassandra documentation](#)

Advanced topics

- [Secondary indexing in Azure Cosmos DB Cassandra API](#)
- [Handling rate limited requests in the Azure Cosmos DB API for Cassandra](#)
- [Change feed in the Azure Cosmos DB API for Cassandra](#)
- [Using the Change Feed with Azure Cosmos DB's API for Cassandra](#)
- [Connect to Azure Cosmos DB Cassandra API from Spark](#)
- [Migrate Cassandra workloads to Cosmos DB](#)

Features supported by Azure Cosmos DB Cassandra API

Some feature might not be supported by Cosmos DB Cassandra API and there are some key differences between Apache Cassandra and Cosmos DB Cassandra API:

- [Apache Cassandra features supported by Azure Cosmos DB Cassandra API](#)
- [Frequently asked questions about the Cassandra API in Azure Cosmos DB](#)

Lab Requirements & Pre-requisites:

To complete the labs, you will need the following:

- Microsoft Azure Subscription
- An active Azure Cosmos DB Cassandra API account
- [Download Apache Cassandra](#) (to use cqlsh)
- Java Development Kit (JDK) 1.8+
- Anaconda ([download here](#))
- Windows 10 machine (please use [Boot Camp Assistant](#) if you are using a MAC machine OR adapt the lab below to be used on your MAC machine OR create on a Azure Windows Virtual Machine)

1. Provision Cosmos DB Cassandra API account

This challenge requires one Cosmos DB account with Cassandra API.

Follow these [instructions](#) to create a new Azure Cosmos DB account for Cassandra API.

- Go to the Azure Portal: <https://ms.portal.azure.com/>.
- Search for “Azure Cosmos DB” in the search bar above and +New account.
- Enter a resource group name and location (suggest selecting a region near you). It is highly recommended that you create a New Resource Group for this lab.
- **Select Cassandra API.**
- Do not enable multi-region writes or geo-redundancy.
- In the first tab(Basics), click in **Review + create**. It should take approximately 15 to 20 minutes for your account to be created.

The screenshot shows the 'Create Azure Cosmos DB Account' page in the Microsoft Azure Portal. The page is titled 'Create Azure Cosmos DB Account' and has a search bar at the top. The 'Basics' tab is selected, and the 'Review + create' button is highlighted with a red box. The page contains the following fields and options:

- Subscription:** Microsoft Azure Internal Consumption
- Resource Group:** afrg (with a 'Create new' link)
- Instance Details:**
 - Account Name:** cassandra-test-account
 - API:** Cassandra
 - Location:** (Europe) West Europe
 - Capacity mode:** Provisioned throughput (selected), Serverless (preview)
- Apply Free Tier Discount:** Do Not Apply (selected)

At the bottom, there are three buttons: 'Review + create' (highlighted with a red box), 'Previous', and 'Next: Global Distribution'.

2. Install Java Development Kit

We need to ensure we have a Java Runtime environment installed. We are going to install Java Development Kit (JDK) 1.8+ in your computer.

1. Copy the following link to a browser in your local machine, download and install the JDK:
https://elearning.novaims.unl.pt/pluginfile.php/88741/mod_folder/content/0/jdk-8u251-windows-i586.exe?forcedownload=1

The jdk-8u251-windows-i586.exe is in the folder *****Challenge 3 – NoSQL*****:
<https://elearning.novaims.unl.pt/mod/folder/view.php?id=34165>.

For this lab we are going to use *Java SE Development Kit 8u251* (Windows x86).

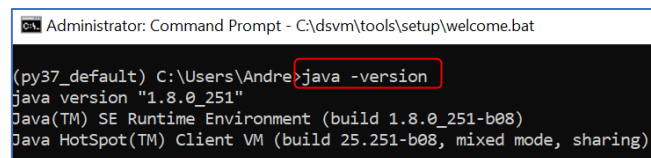
Note: If the file is not compatible with your machine or you want to download another JDK 8 version, please use the following link:

- <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>. You will need to register in order to download the Java Development Kit.

2. To ensure you have installed Java correctly open a command line and type:

Command Prompt:

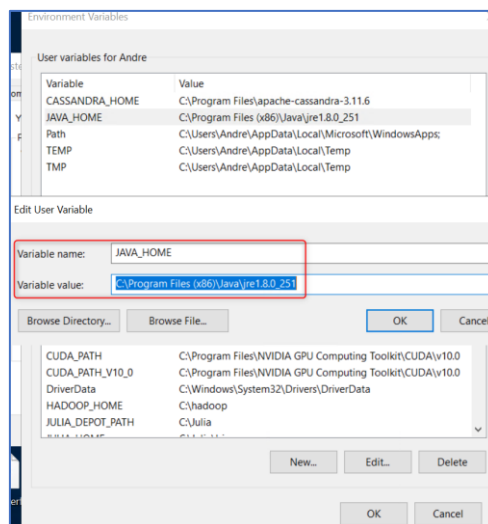
```
java -version
```



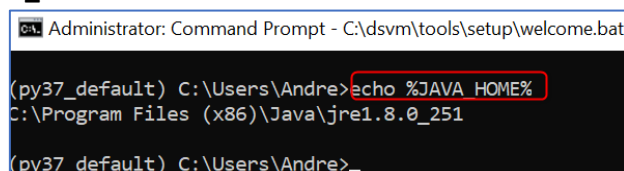
```
Administrator: Command Prompt - C:\dsvm\tools\setup\welcome.bat

(py37_default) C:\Users\Andre>java -version
java version "1.8.0_251"
Java(TM) SE Runtime Environment (build 1.8.0_251-b08)
Java HotSpot(TM) Client VM (build 25.251-b08, mixed mode, sharing)
```

4. We need to set the **JAVA_HOME** environment variable to point to the folder where the JRE is installed. To add the Java Runtime environment to our environment variables go to **Environment Variables** on your computer and add the value where your JRE is **C:\Program Files (x86)\Java\jre1.8.0_251**.



5. To make sure the environment variable has been set you can open a new command prompt and paste **echo %JAVA_HOME%** and it should return the location where we installed the JRE.



```
Administrator: Command Prompt - C:\dsvm\tools\setup\welcome.bat

(py37_default) C:\Users\Andre>echo %JAVA_HOME%
C:\Program Files (x86)\Java\jre1.8.0_251

(py37_default) C:\Users\Andre>
```

3. Install Apache Cassandra

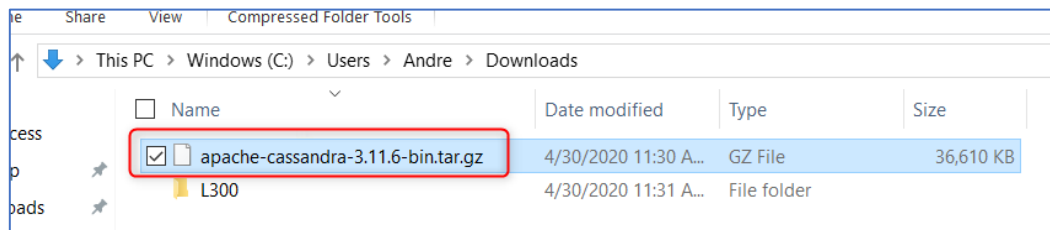
1. We need to install Apache Cassandra to work with **cqlsh**(Cassandra Shell).

To download the latest version of Apache Cassandra please click on the link below and download it to your computer(copy and paste the following URL to a browser in your computer):

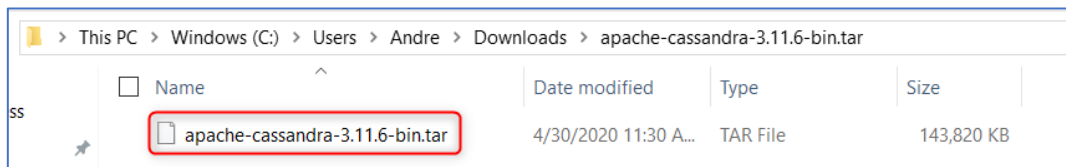
<https://mirrors.up.pt/pub/apache/cassandra/3.11.10/apache-cassandra-3.11.10-bin.tar.gz>

Or go to <https://elearning.novaims.unl.pt/mod/folder/view.php?id=34165> and download **apache-cassandra-3.11.10-bin.tar.gz**. In case that you cannot download Apache Cassandra through this link or want to download other versions of Apache Cassandra please check <https://cassandra.apache.org/download/>.

2. Once you have the file **apache-cassandra-3.11.10-bin.tar.gz** that you have just download you simply need to unzip / extract the files. (Some images below show 3.11.6, but it will be the same for 3.11.10).

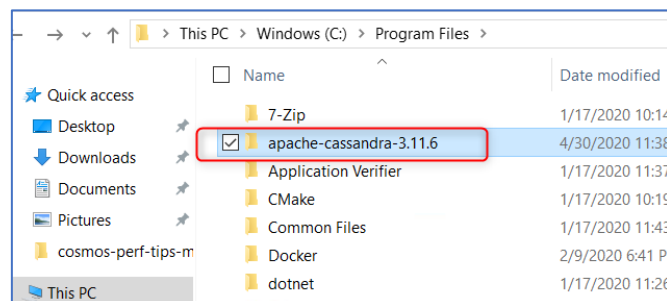


Go to the folder you have unzipped, and you will need to **unzip again** the file **apache-cassandra-3.11.10-bin.tar**. This should take a bit more time because there are more files to extract.

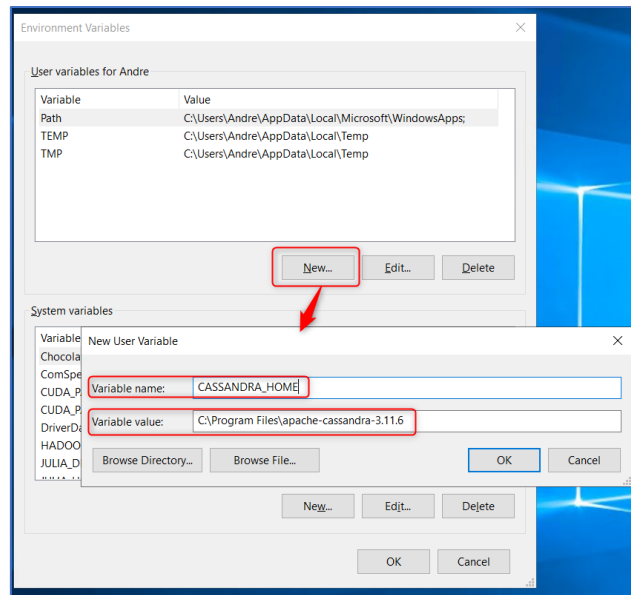


3. Once finished extracting all the files you should see the **apache-cassandra-3.11.10-bin**. You might get a warning saying there are duplicate files, this is ok, just say yes to all files and your installation should be fine.

4. You can copy the entire **apache-cassandra-3.11.10** (this is the folder inside the **apache-cassandra-3.11.10-bin** folder) to your **C:\Program Files** folder.



5. In your computer, we are going to set the **CASSANDRA_HOME** environment variable with the value: **C:\Program Files\apache-cassandra-3.11.10**.



6. To make sure the environment variable has been set you can open a new command prompt and paste **echo %CASSANDRA_HOME%** and it should return the location where we installed Cassandra.

```
C:\Administrator: Command Prompt - C:\dsvm\tools\setup\welcome.bat

(py37_default) C:\Users\Andre>echo %CASSANDRA_HOME%
C:\Program Files\apache-cassandra-3.11.6

(py37_default) C:\Users\Andre>_
```

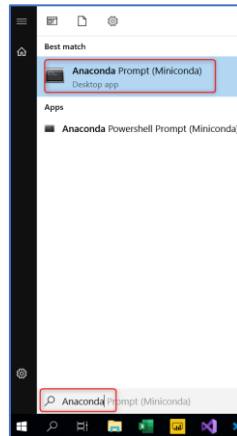
Please continue to the next page.

4. Using Anaconda Python 2.7 version to use CQLSH

We are going to work with **cqlsh** to connect to our Azure Cosmos DB Cassandra API. Because **cqlsh** requires a **Python 2 interpreter** we need to create a python 2 environment.

Make sure you install Anaconda ([download here](#)) on your computer. We are going to use it to create a **python version 2 environment**.

1. After downloading Anaconda, search for “**Anaconda**” in your computer you should see as result from search the Anaconda Prompt appearing



2. In the Anaconda prompt paste the following command to create a Python 2.7 environment.

Anaconda prompt:

```
conda create -n Python27 python=2.7
```

3. To activate the environment created execute the following command:

Anaconda prompt:

```
conda activate Python27
```

Important: Any time you need to open a new Anaconda prompt we need to activate this environment! You can pin Anaconda prompt to your Taskbar.

4. After activating the *Python27* environment, make sure the version of Python is indeed 2.7, for this run the following command:

Anaconda prompt:

```
python --version
```

```
Select Administrator: Command Prompt - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\c
# To activate this environment, use
#
# $ conda activate Python27
#
# To deactivate an active environment, use
#
# $ conda deactivate

(py37_default) C:\Users\Andre>
(py37_default) C:\Users\Andre>
(py37_default) C:\Users\Andre>conda activate Python27

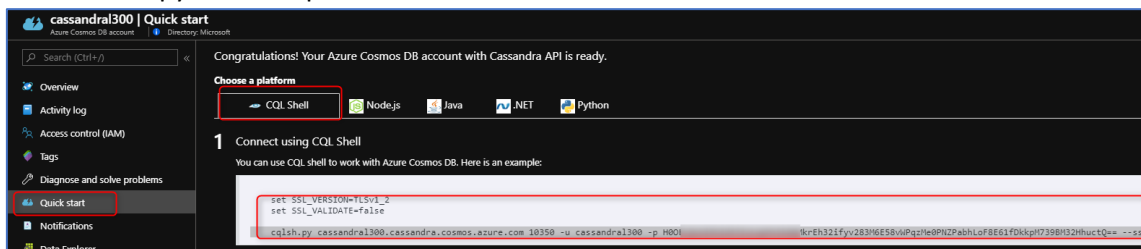
(Python27) C:\Users\Andre>python --version
Python 2.7.18 :: Anaconda, Inc.

(Python27) C:\Users\Andre>
```

1. Connecting CQLSH to Cosmos DB Cassandra Account

CQLSH is what allows us to interact through the command prompt with our Cassandra account in Azure Cosmos DB, this is where we can write the Cassandra query language, setup tables, add records to our tables and much more. You can communicate with the Azure Cosmos DB Cassandra API through Cassandra Query Language (CQL) v4 [wire protocol](#) compliant open-source Cassandra client [drivers](#).

1. In the Anaconda prompt navigate to **C:\Program Files\apache-cassandra-3.11.10** which is where we installed Apache Cassandra.
2. Inside this folder go to the **bin** folder (**C:\Program Files\apache-cassandra-3.11.10\bin**).
3. Go to your Cosmos DB Cassandra account that you created previously and go to **Quick start** menu and copy the example.



4. In your Computer, in the Anaconda prompt paste the code from the example:

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python...
(base) C:\Program Files\apache-cassandra-3.11.6\bin>conda activate Python27
(Python27) C:\Program Files\apache-cassandra-3.11.6\bin>set SSL_VERSION=TLSv1_2
(Python27) C:\Program Files\apache-cassandra-3.11.6\bin>set SSL_VALIDATE=false
(Python27) C:\Program Files\apache-cassandra-3.11.6\bin>python cqlsh.py cassandra1300.cassandra.cosmos.azure.com 10350 -u cassandra1300 -p H00EcQchi9UIdt3isLwpMxikaQWkrEh32ifv283M6E58vWpQzMe0PNZPabhLoF8E61fDkkpM739BM32HhuctQ== --ssl
WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.
Connected to CosmosProd at cassandra1300.cassandra.cosmos.azure.com:10350.
[cqlsh 5.0.1 | Cassandra 3.11.0 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cassandra1300@cqlsh>
cassandra1300@cqlsh>
cassandra1300@cqlsh>
```

5. Congratulations, you are now connected to Cosmos DB Cassandra account.

When you open a new Anaconda prompt always follow the steps in the picture above to successfully connect to your Cosmos DB Cassandra account! We have created a helper file to ease the connection to the Cosmos DB Cassandra API account and make sure you don't forget any step. Please download the **helper.txt** file here:

- <https://elearning.novaims.unl.pt/mod/folder/view.php?id=34165>

Note: Instead of running the following command

```
Anaconda prompt:
python cqlsh.py cassandra1300.cassandra.cosmos.azure.com 10350 -u cassandra1300 -p <PRIMARY PASSWORD>--ssl
```

You can just run just:

```
Anaconda prompt:
cqlsh cassandra1300.cassandra.cosmos.azure.com 10350 -u cassandra1300 -p <PRIMARY PASSWORD>--ssl
```

2. Keyspaces and Tables

In order to interact with Cassandra, we use the CQLSH and you will notice during this lab that the syntax of CQL is very similar to SQL. You should be able to use your existing knowledge of Cassandra to interact with a Cosmos DB Cassandra account.

Task 1: Create Keyspace and Tables

1. Once in the Cassandra Shell let's go ahead and create a [Keyspace](#). A Keyspace in Cosmos DB is the equivalent of a database in SQL API.

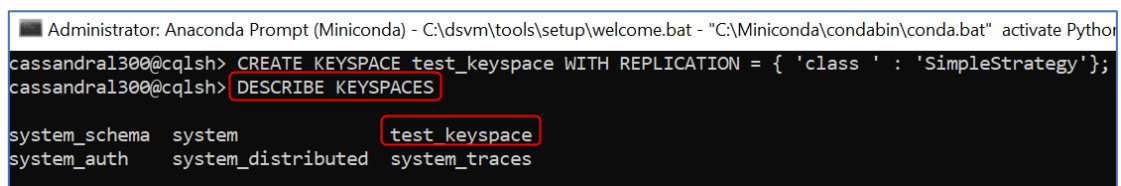
CQLSH:

```
CREATE KEYSPACE test_keyspace WITH REPLICATION = { 'class' : 'SimpleStrategy'};
```

2. You can check the Keyspaces you have created by using the Describe command:

CQLSH:

```
DESCRIBE KEYSPACES
```



The screenshot shows a terminal window titled "Administrator: Anaconda Prompt (Miniconda)". The user has entered the command `CREATE KEYSPACE test_keyspace WITH REPLICATION = { 'class' : 'SimpleStrategy'};` and then `DESCRIBE KEYSPACES`. The output shows a table of keyspace information:

system_schema	system	test_keyspace
system_auth	system_distributed	system_traces

We can see the Keyspace that we have just created as well as some system Keyspaces that are there by default

3. You can drop a Keyspace by using the [Drop](#) command. You can drop and recreate the Keyspace again.

CQLSH:

```
DROP KEYSPACE test_keyspace;
```

4. You can specify the Keyspace by using the [USE](#) command or explicitly specify *keyspace.tablename* in your future operations:

CQLSH:

```
USE test_keyspace;
```

5. Next we will create a simple table inside the **test_keyspace** Keyspace that we have just created.

Some important points we need to take into consideration before proceeding:

- In Cassandra, a table contains a set of rows which contains many key-value pair columns
- Cassandra deals with wide rows that consists of a Primary Key and a large number of columns.
- A **Primary Key** uniquely identifies a row.
- In Cassandra, every row has a **Primary Key** which should be specified for data access.
- The **Primary Key** is a general concept to indicate one or more columns used to retrieve data from a Table.
- The **Primary Key** is also known as the **Composite Key** which is made of the Partition key, which decides on which nodes data will be stored.

We are going to [create a table](#) called **employee_by_id**, where *id* is unique for the employees and is used as our Primary Key. A table is the equivalent to a container in Cosmos DB.

In Azure Cosmos DB Cassandra API you will need to use the option [cosmosdb_provisioned_throughput](#) to set the value of the provisioned throughput for that table. If you don't specify the Throughput value, the Table will be created with a default value of 400 RUs.

CQLSH:

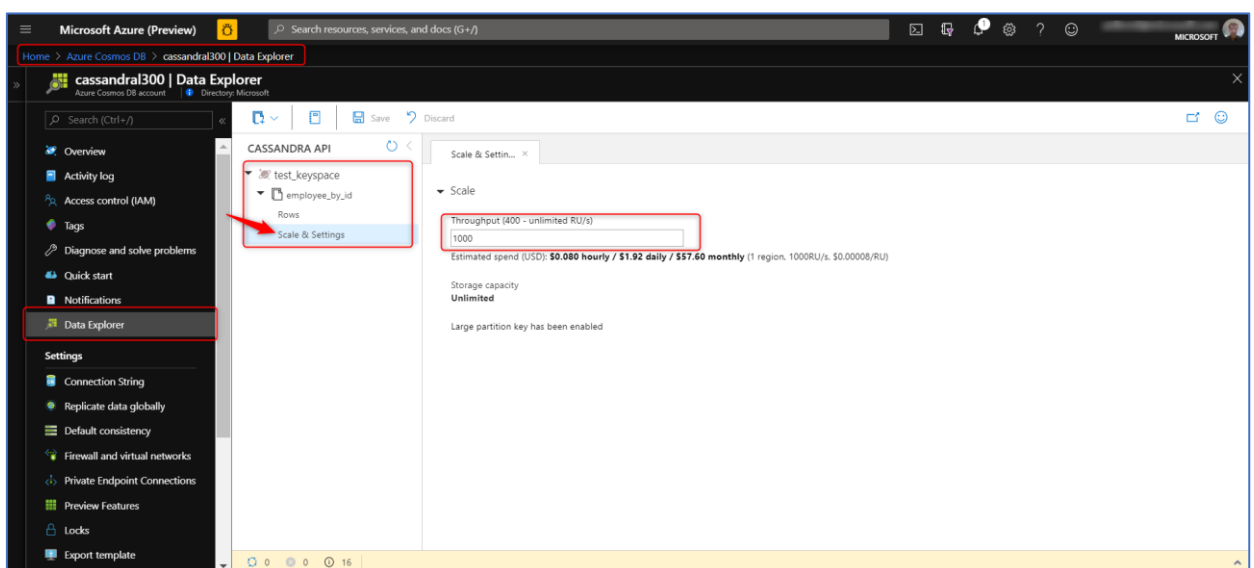
```
CREATE TABLE employee_by_id (id int PRIMARY key, name text, position text) WITH cosmosdb_provisioned_throughput=400;
```

```
Select Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python27 - python cqlsh.py cassandra300.cassandra.c...

(Python27) C:\Program Files\apache-cassandra-3.11.6\bin>python cqlsh.py cassandra300.cassandra.cosmos.azure.com 10350 -u cassandra300 -p H80EcQcWi9Udt3is...
WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to CosmosProd at cassandra300.cassandra.cosmos.azure.com:10350.
[cqlsh 5.0.1 | Cassandra 3.11.0 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cassandra300@cqlsh>
cassandra300@cqlsh> CREATE KEYSPACE test_keyspace WITH REPLICATION = { 'class' : 'SimpleStrategy' };
cassandra300@cqlsh> USE test_keyspace;
cassandra300@cqlsh:test_keyspace> CREATE TABLE employee_by_id (id int PRIMARY key, name text, position text) WITH cosmosdb_provisioned_throughput=1000;
cassandra300@cqlsh:test_keyspace>
```

6. In the [Azure portal](#), go to your Cosmos DB Cassandra API account that you are using on this lab and you should be able to see that the table we just created is available through portal. Go to **Data Explorer** and inside the table *employee_by_id* click in **Scale & Settings** and confirm the provisioned throughput value is set **to 400 RUs!** To take advantage of the Azure Student Subscription, otherwise you will pay for the excess and your credit will be reduced!



7. In order to see the table that are created you can run the following command:

CQLSH:

```
DESCRIBE Tables;
```

8. You can drop a Table by using the [Drop](#) command:

CQLSH:

```
DROP TABLE employee_by_id;
```

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python27 - python cqlsh.py cassandra300.cassandra.cosmos...

cassandra300@cqlsh:test_keyspace>
cassandra300@cqlsh:test_keyspace> DESCRIBE Tables;

employee_by_id

cassandra300@cqlsh:test_keyspace> DROP TABLE employee_by_id;
cassandra300@cqlsh:test_keyspace> DESCRIBE Tables;

<empty>

cassandra300@cqlsh:test_keyspace> CREATE TABLE employee_by_id (id int PRIMARY key, name text, position text) WITH cosmosdb_provisioned_throughput=1000;
```

9. To change the provisioned throughput value of your Table you can also execute the following command:

```
CQLSH:
ALTER TABLE employee_by_id WITH cosmosdb_provisioned_throughput=400;
```

10. Go to the Azure portal and **confirm that the throughput has been set to 400 RU/s.**

Task II: Composite Key

We are going to create another table called ***employee_by_car_make***, in this case car make is not unique by employee, so we cannot take the same approach as in the table ***employee_by_id***. In this case, we have to apply a **Primary Key** called a **Composite Key**, where we use the ***car_make*** as the **Partition Key** and we also apply a number of clustering columns to make the Primary Key unique.

1. In the Cassandra Shell create the table ***employee_by_car_make***:

```
CQLSH:
CREATE TABLE employee_by_car_make (car_make text, id int, car_model text, PRIMARY KEY(car_make, id));
```

- The combination of ***car_make*** and ***id*** it will be unique, so that is what we use to create a Composite Key.
- In this situation we have a **Composite Key**, the "first part" of the key is called **PARTITION KEY** (in this example ***car_make*** is the partition key) and the second part of the key is the **CLUSTERING KEY** (in this example ***id***). In Cassandra, the Clustering Key helps to manage how the data will be sorted.

2. To make sure it has been created go ahead and execute:

```
CQLSH:
DESCRIBE TABLES;
```

3. To describe an individual table you run the following command:

```
CQLSH:
DESCRIBE TABLE employee_by_car_make;
```



```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python27 - python cqlsh.py cassandra300.cassandra.cosmos.a...
cassandra300@cqlsh:test_keyspace> CREATE TABLE employee_by_car_make (car_make text, id int, car_model text, PRIMARY KEY(car_make, id));
cassandra300@cqlsh:test_keyspace> DESCRIBE TABLES;

employee_by_car_make  employee_by_id

cassandra300@cqlsh:test_keyspace> DESCRIBE TABLE employee_by_car_make;

CREATE TABLE test_keyspace.employee_by_car_make (
  car_make text,
  id int,
  car_model text,
  PRIMARY KEY (car_make, id)
) WITH CLUSTERING ORDER BY (id ASC)
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
AND cdc = false
AND comment = ''
AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
AND crc_check_chance = 1.0
AND dlocal_read_repair_chance = 0.0
AND default_time_to_live = 0
AND gc_grace_seconds = 7776000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 3600000
AND min_index_interval = 128
AND read_repair_chance = 0.0
AND speculative_retry = '99PERCENTILE';

cassandra300@cqlsh:test_keyspace>
```

You will get several details about the table, it will contain the schema of the table, including the Primary Key and will provide the **CLUSTERING ORDER BY (id ASC)** which is how the data will be stored.

Task III: Multiple Clustering Keys

In Cassandra it's also possible to specify multiple Clustering columns (**Clustering Keys**) for one or more tables. We might want to do this when we want our data stored by two different columns. For instance, we may want to have another table *employee_by_car_make_sorted* where we still use the *car_make* as the **Partition Key**, **Clustering Key** for *age* and *id*. So, in this case the list of employees will first be ordered by *age* and then for a certain age they will be ordered within that by their *id*.

1. To achieve what was just mentioned in CQLSH write the following command:

CQLSH:

```
CREATE TABLE employee_by_car_make_sorted (car_make text, age int, id int, name text, PRIMARY KEY(car_make, age, id));
```

- The **Partition Key** is still *car_make*.
- We specify multiple **Clustering Keys**: *age* and *id*.

2. To make sure it has been created in CQLSH execute:

CQLSH:

```
DESCRIBE TABLES;
```

Task IV: Partition Key with multiple columns

Until now the **Primary Key** consisted of a single column which is the **Partition Key** and multiple **Clustering Keys**. However, we can also specify multiple columns to make **Partition Key**. In this example, we'll use multiple columns to come up with the **Partition Key** and then have a single or more than one **Clustering Key**. The combination of the **Partition Key** and the **Clustering Key** will be unique to make the **Primary Key**.

1. In CQLSH write the following command:

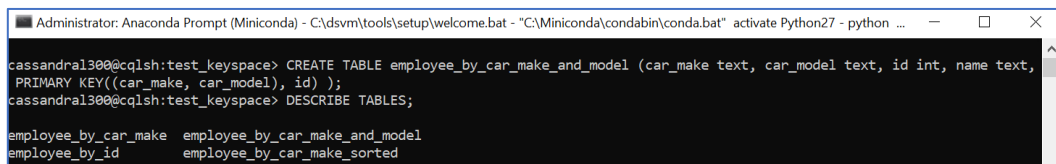
CQLSH:

```
CREATE TABLE employee_by_car_make_and_model (car_make text, car_model text, id int, name text, PRIMARY KEY((car_make, car_model), id));
```

2. To make sure it has been created go execute in CQLSH:

CQLSH:

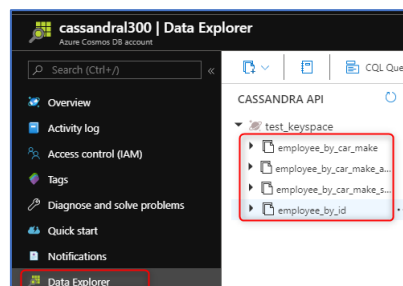
```
DESCRIBE TABLES;
```



```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python27 - python ...
cassandra300@cqlsh:test_keyspace> CREATE TABLE employee_by_car_make_and_model (car_make text, car_model text, id int, name text,
PRIMARY KEY((car_make, car_model), id));
cassandra300@cqlsh:test_keyspace> DESCRIBE TABLES;

employee_by_car_make  employee_by_car_make_and_model
employee_by_id        employee_by_car_make_sorted
```

You should have available 4 tables at this moment in your Cosmos DB Cassandra API account. You can check them on the Azure Portal:



3. To describe the last table run the following command in CQLSH:

CQLSH:

```
DESCRIBE TABLE employee_by_car_make_and_model;
```

You can now confirm that the Primary Key is formed like **PRIMARY KEY ((car_make, car_model), id)**.

```
cassandra1380@cqlsh:test_keyspace> DESCRIBE employee_by_car_make_and_model;  
  
CREATE TABLE test_keyspace.employee_by_car_make_and_model (  
  car_make text,  
  car_model text,  
  id int,  
  name text,  
  PRIMARY KEY ((car_make, car_model), id)  
) WITH CLUSTERING ORDER BY (id ASC)  
AND bloom_filter_fp_chance = 0.01
```

Before proceeding, we are going to summarize some concepts we learnt until now with a simple example.

- A **Primary Key** uniquely identifies a row. the partition key is used to place the entity in the right location. In Azure Cosmos DB, it's used to find the right logical partition that's stored on a physical partition.
- A **Composite Key** is a key formed from multiple columns.
- A **Partition Key** is the primary lookup to find a set of rows, i.e. a partition.
- A **Clustering Key** is the part of the **Primary Key** that isn't the **Partition Key** (and defines the ordering within a partition).

Example:

- **PRIMARY KEY (a):** The partition key is a.
- **PRIMARY KEY (a, b):** The partition key is a, the clustering key is b.
- **PRIMARY KEY ((a, b)):** The composite partition key is (a, b).
- **PRIMARY KEY (a, b, c):** The partition key is a, the composite clustering key is (b, c).
- **PRIMARY KEY ((a, b), c):** The composite partition key is (a, b), the clustering key is c.
- **PRIMARY KEY ((a, b), c, d):** The composite partition key is (a, b), the composite clustering key is (c, d).

3. Consistency, Inserts and Selects

Task I: Consistency

When using Azure Cosmos DB Cassandra API, applications get a full set of consistency levels offered by Apache Cassandra, with even stronger consistency and durability guarantees.

Unlike Azure Cosmos DB, Apache Cassandra does not natively provide precisely defined consistency guarantees. Instead, Apache Cassandra provides a write consistency level and a read consistency level, to enable the high availability, consistency, and latency trade-offs.

When using Azure Cosmos DB's Cassandra API:

- The write consistency level of Apache Cassandra is mapped to the default consistency level configured on your Azure Cosmos account. Consistency for a write operation (CL) can't be changed on a per-request basis.
- Azure Cosmos DB will dynamically map the read consistency level specified by the Cassandra client driver to one of the Azure Cosmos DB consistency levels configured dynamically on a read request.

This document shows the corresponding Azure Cosmos DB consistency levels for Apache Cassandra and Apache Cassandra consistency levels:

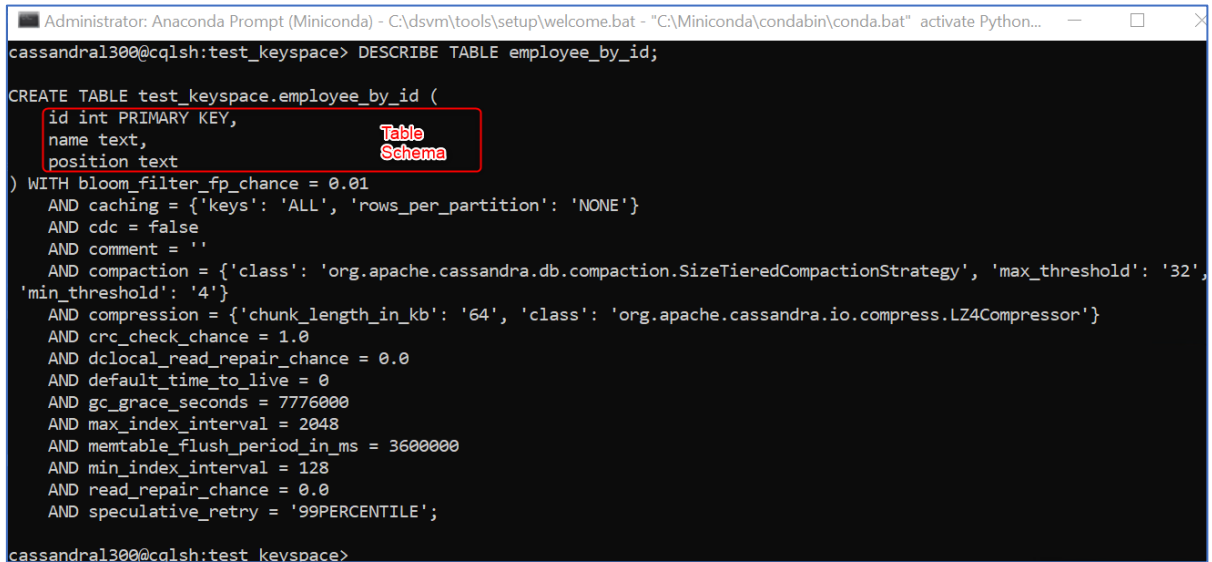
[Mapping between Apache Cassandra and Azure Cosmos DB consistency levels.](#)

Task II: Inserts, Selects and Order By

1. We are going to learn how to insert data into the tables we created before. In order to get the schema of a specific table run the following command:

CQLSH:

```
DESCRIBE TABLE employee_by_id;
```



```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python...
cassandra1300@cqlsh:test_keyspace> DESCRIBE TABLE employee_by_id;

CREATE TABLE test_keyspace.employee_by_id (
  id int PRIMARY KEY,
  name text,
  position text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.0
    AND default_time_to_live = 0
    AND gc_grace_seconds = 7776000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 3600000
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';

cassandra1300@cqlsh:test_keyspace>
```

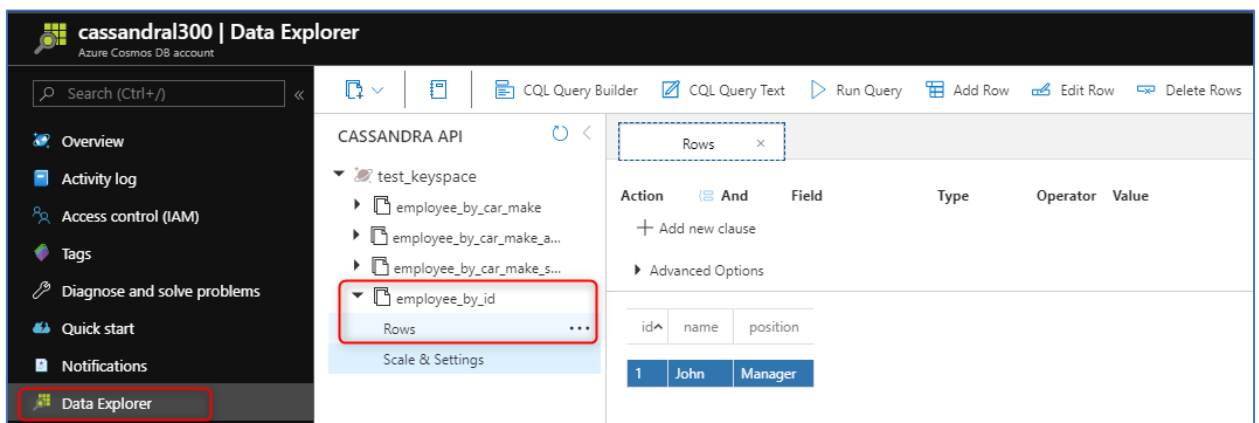
2. To add data we will use the [INSERT INTO](#) command:

CQLSH:

```
INSERT INTO employee_by_id (id, name, position) VALUES (1, 'John', 'Manager');
```

- Don't forget that in Cassandra we don't need to add data to all the columns.
- Note that we are using Single Quotes if you use Double Quotes, you'll get an error.

3. Now go to the Azure portal and confirm that the data has been inserted into the table **employee_by_id**. Go to **Data Explorer** and inside the table **employee_by_id** click in **Rows**:



4. Go back to CQLSH. We are going to query the table **employee_by_id** to get all the data inside it.

CQLSH:

```
SELECT * FROM employee_by_id;
```

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python...
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_id (id, name, position) VALUES (1, 'John', 'Manager');
cassandra1300@cqlsh:test_keyspace>
cassandra1300@cqlsh:test_keyspace>
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position
-----+-----+-----
1 | John | Manager

(1 rows)
```

5. We will add a second employee to our database:

CQLSH:

```
INSERT INTO employee_by_id (id, name, position) VALUES (2, 'Bob', 'CEO');
```

6. Now, we will just query by *id* equal to 2. We will need to use a [Where](#) clause in this case.

CQLSH:

```
SELECT * FROM employee_by_id WHERE id=2;
```

This query will just return Bob as expected.

7. Let's imagine we just want to get data from John, and we don't know John's *id*. We are going to query based on *name* column:

CQLSH:

```
SELECT * FROM employee_by_id WHERE name='John';
```

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python...
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id WHERE id=2;

id | name | position
-----+-----+-----
2 | Bob | CEO

(1 rows)
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id WHERE name='John';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cassandra1300@cqlsh:test_keyspace>
```

This Query does not work because the field name is not part of the **Primary key** as *name* is not part of the **Primary Key**. In Cassandra we always want to query data based on the **Primary Key** which in this case is *id*.

Since *John* is not part of the **Primary Key**, if we are querying on the *name* the data could be spread across multiple partitions in the Cosmos DB account. In this situation, it's possible to query by *name* but is not recommended because it can lead to a very poorly performing query. In Cosmos DB we call this type of query a [Cross-Partition](#) query.

8. Let's add some data to our **employee_by_car_make** table, but first we should describe the table to get the schema.

CQLSH:

```
DESCRIBE TABLE employee_by_car_make;
```

The **Primary Key** is *car_make* with the *id*, where *id* is the **Clustering Key**. The *id* field is the **Partition Key**.

```
cassandra1300@cqlsh:test_keyspace> DESCRIBE TABLE employee_by_car_make;

CREATE TABLE test_keyspace.employee_by_car_make (
  car_make text,
  id int,
  car_model text,
  PRIMARY KEY (car_make, id)
) WITH CLUSTERING ORDER BY (id ASC)
```

9. To add data we will use the [INSERT INTO](#) command as we did before:

CQLSH:

```
INSERT INTO employee_by_car_make (car_make, id, car_model) VALUES ('BMW', 1, 'Sports Car');
```

10. Again as we did before, in CQLSH query the table **employee_by_car_make** to get all the data inside it.

CQLSH:

```
SELECT * FROM employee_by_car_make;
```

11. We will add a more rows to the table **employee_by_car_make**.

CQLSH:

```
INSERT INTO employee_by_car_make (car_make, id, car_model) VALUES ('BMW', 2, 'Sports Car');
INSERT INTO employee_by_car_make (car_make, id, car_model) VALUES ('AUDI', 4, 'Truck');
INSERT INTO employee_by_car_make (car_make, id, car_model) VALUES ('AUDI', 5, 'Hatchback');
```

12. Execute the following query:

CQLSH:

```
SELECT * FROM employee_by_car_make WHERE car_make = 'BMW';
```

This is supported because **car_make** is part of the **Primary Key**, **car_make** is the **Partititon Key**.

13. Now we are going to see what happens if we want to search just by the **id**. In CQLSH run:

CQLSH:

```
SELECT * FROM employee_by_car_make WHERE id = '1';
```

This query will not be supported because **id** is not the **Partition Key**.

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make WHERE id = '1';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering
and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW
FILTERING"
cassandra1300@cqlsh:test_keyspace> _
```

14. The **Clustering Key** **id** is used to order our results. Here we are going to use [ORDER BY](#).

Execute the following query:

CQLSH:

```
SELECT * FROM employee_by_car_make WHERE car_make = 'BMW' ORDER BY id;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make WHERE car_make = 'BMW' ORDER BY id;

car_make | id | car_model
-----+---+-----
BMW      | 1  | Sports Car
BMW      | 2  | Sports Car

(2 rows)
```

15. If we try to order by a field we haven't specified as the Clustering Key, for example, we will try to order by **car_model** this it will not be supported.

CQLSH:

```
SELECT * FROM employee_by_car_make WHERE car_make = 'BMW' ORDER BY car_model;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make WHERE car_make = 'BMW' ORDER BY car_model;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Order by is only supported on the first clustered columns of
the PRIMARY KEY, got car_model"
cassandra1300@cqlsh:test_keyspace> _
```

As you can see in the error message, it says "Order by is only supported on the first clustered columns of the PRIMARY KEY".

16. Finally we will have a look at our table *employee_by_car_make_and_model*. Describe the table to be reminded of the schema.

CQLSH:

```
DESCRIBE TABLE employee_by_car_make_and_model;
```

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda...
cassandra1300@cqlsh:test_keyspace> DESCRIBE TABLE employee_by_car_make_and_model;

CREATE TABLE test_keyspace.employee_by_car_make_and_model (
  car_make text,
  car_model text,
  id int,
  name text,
  PRIMARY KEY ((car_make, car_model), id)
) WITH CLUSTERING ORDER BY (id ASC)
    AND bloom_filter_fp_chance = 0.01
```

The **Primary Key** here is made of *car_make* and *car_model* with *id*, where *car_make* and *car_model* are the **Partition Key** and *id* is the **Clustering Key**.

The **Primary Key** is unique, as there should never be a duplicate combination of *car_make*, *car_model* and *id*.

17. We are going to insert data to it:

CQLSH:

```
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('BMW', 'HATCHBACK', 1, 'John');
```

18. Again as we did before, in CQLSH query the table *employee_by_car_make_and_model* to see if the previous data was added successfully.

CQLSH:

```
SELECT * FROM employee_by_car_make_and_model;
```

19. Now we are going to see what happens if we try to insert data to this table, but we do not specify the *car_model*. Because the *car_model* is part of the **Partition Key**, this will not be allowed:

CQLSH:

```
INSERT INTO employee_by_car_make_and_model (car_make, id, name) VALUES ('BMW', 1, 'John');
```

```
Administrator: Anaconda Prompt (Miniconda) - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda...
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('BMW', 'HATCHBACK', 1, 'John');
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make_and_model;

 car_make | car_model | id | name
-----+-----+---+----
      BMW | HATCHBACK |  1 | John

(1 rows)
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_car_make_and_model (car_make, id, name) VALUES ('BMW', 1, 'John');
InvalidRequest: Error from server: code=2200 [Invalid query] message="Missing Partition key component in insert request"
cassandra1300@cqlsh:test_keyspace> _
```

As you can check in the error message, you will get the error message: "Missing Partition key component in insert request".

20. If we don't specify the *id* we also get an error, as *id* is also part of the **Primary Key**. In CQLSH run the following command:

```
CQLSH:
INSERT INTO employee_by_car_make_and_model (car_make, car_model, name) VALUES ('BMW', 'HATCHBACK', 'John');
```

Again, we will the same error message:

```
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_car_make_and_model (car_make, car_model, name)
VALUES ('BMW', 'HATCHBACK', 'John');
InvalidRequest: Error from server: code=2200 [Invalid query] message="Missing Cluster key component in in
sert request"
cassandra1300@cqlsh:test_keyspace> .
```

21. Now we are going to insert another piece of data with the same **Primary Key** as before. But first, run a simple query to view the data:

```
CQLSH:
SELECT * FROM employee_by_car_make_and_model;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make_and_model;

 car_make | car_model | id | name
-----+-----+---+-----
      BMW | HATCHBACK | 1 | John
(1 rows)
```

22. Insert the data with the same **Primary Key**, and we can change the *name*.

```
CQLSH:
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('BMW', 'HATCHBACK', 1, 'Bob');
```

```
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, n
ame) VALUES ('BMW', 'HATCHBACK', 1, 'Bob');
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make_and_model;

 car_make | car_model | id | name
-----+-----+---+-----
      BMW | HATCHBACK | 1 | Bob
(1 rows)
```

We can see that the *name* has been changed to **Bob**.

23. We can also run the same query without specifying the *name*, since *name* is not part of the **Partition Key**:

```
CQLSH:
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id) VALUES ('BMW', 'HATCHBACK', 2);
```

We can see there will not be any name for that row. The name will be set to *null*.

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make_and_model;

 car_make | car_model | id | name
-----+-----+---+-----
      BMW | HATCHBACK | 1 | Bob
      BMW | HATCHBACK | 2 | null
(2 rows)
```

24. Insert the following data using CQLSH:

CQLSH:

```
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id) VALUES ('BMW', 'HATCHBACK', 3);
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('BMW', 'HATCHBACK', 8, 'FRANK');
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('AUDI', 'TRUCK', 7, 'AMY');
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('BMW', 'TRUCK', 4, 'TIM');
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('AUDI', 'SPORTS CAR', 5, 'JIM');
INSERT INTO employee_by_car_make_and_model (car_make, car_model, id, name) VALUES ('AUDI', 'SPORTS CAR', 6, 'NICK');
```

25. Query to confirm we have successfully added the data:

CQLSH:

```
SELECT * FROM employee_by_car_make_and_model;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make_and_model;

car_make | car_model | id | name
-----+-----+---+----
AUDI     | SPORTS CAR | 5 | JIM
AUDI     | SPORTS CAR | 6 | NICK
AUDI     | TRUCK      | 7 | AMY
BMW      | HATCHBACK  | 1 | Bob
BMW      | HATCHBACK  | 2 | null
BMW      | HATCHBACK  | 3 | null
BMW      | HATCHBACK  | 8 | FRANK
BMW      | TRUCK      | 4 | TIM

(8 rows)
```

26. We now want to Query Where *car_make* is equal to BMW and *car_model* is equal to HATCHBACK, we use AND operation to help.

CQLSH:

```
SELECT * FROM employee_by_car_make_and_model WHERE car_make='BMW' AND car_model='HATCHBACK';
```

4. Timestamps, Update, TTLs, Collections and Secondary Indexes

1. Before starting this part, please insert the following data

CQLSH:

```
INSERT INTO employee_by_car_make (car_make, car_model, id) VALUES ('MERC', 'Saloon', 3);
INSERT INTO employee_by_car_make (car_make, car_model, id) VALUES ('MERC', 'Saloon', 6);
INSERT INTO employee_by_car_make (car_make, car_model, id) VALUES ('BMW', 'Sports Car', 7);
INSERT INTO employee_by_car_make (car_make, car_model, id) VALUES ('AUDI', 'Sports Car', 8);
```

2. Confirm the data has been added successfully by querying the table:

CQLSH:

```
SELECT * FROM employee_by_car_make;
```

You should get the below result:

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make
... ;

car_make | id | car_model
-----+---+-----
BMW      | 1 | Sports Car
BMW      | 2 | Sports Car
BMW      | 7 | Sports Car
AUDI     | 4 | Truck
AUDI     | 5 | Hatchback
AUDI     | 8 | Sports Car
MERC     | 3 | Saloon
MERC     | 6 | Saloon

(8 rows)
```

Task I: Timestamps

1. We can check the write time when data was last written to a specific column in Cassandra, for instance, in the table **employee_by_car_make** to achieve this we need to use the function [writetime](#).

CQLSH:

```
SELECT car_make, car_model, writetime(car_model) FROM employee_by_car_make;
```

When you run this query, it should return the last write time for the *car_model* column.

```
cassandra1300@cqlsh:test_keyspace> SELECT car_make, car_model, writetime(car_model) FROM employee_by_car_make;
```

car_make	car_model	writetime(car_model)
BMW	Sports Car	1588602741000000
BMW	Sports Car	1588608556000000
BMW	Sports Car	1588667452000000
AUDI	Truck	1588608556000000
AUDI	Hatchback	1588608556000000
AUDI	Sports Car	1588667452000000
MERC	Saloon	1588667451000000
MERC	Saloon	1588667451000000

(8 rows)

Task II: Update

In Cassandra we can Update the values for a specific rows and specific columns.

1. For example, let's update the BMW *car_model* with *id* equal to 1 to *Truck* value. We can do this using the [Update](#) keyword. To update the *car_model* we need to SET that column to the value we want to change, and we need to use the WHERE clause to specify exactly which rows we want this to be applied, we will need to supply the *id* and the *car_make* because both fields are part of the **Primary Key**.

CQLSH:

```
UPDATE employee_by_car_make SET car_model='TRUCK' WHERE car_make='BMW' AND id=1;
```

2. Query for all data to see the changes that occurred.

CQLSH:

```
SELECT * FROM employee_by_car_make;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_car_make SET car_model='TRUCK' WHERE car_make='BMW' AND id=1;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make;
```

car_make	id	car_model
BMW	1	TRUCK
BMW	2	Sports Car
BMW	7	Sports Car
AUDI	4	Truck
AUDI	5	Hatchback
AUDI	8	Sports Car
MERC	3	Saloon
MERC	6	Saloon

Task III: Time to Live (TTL)

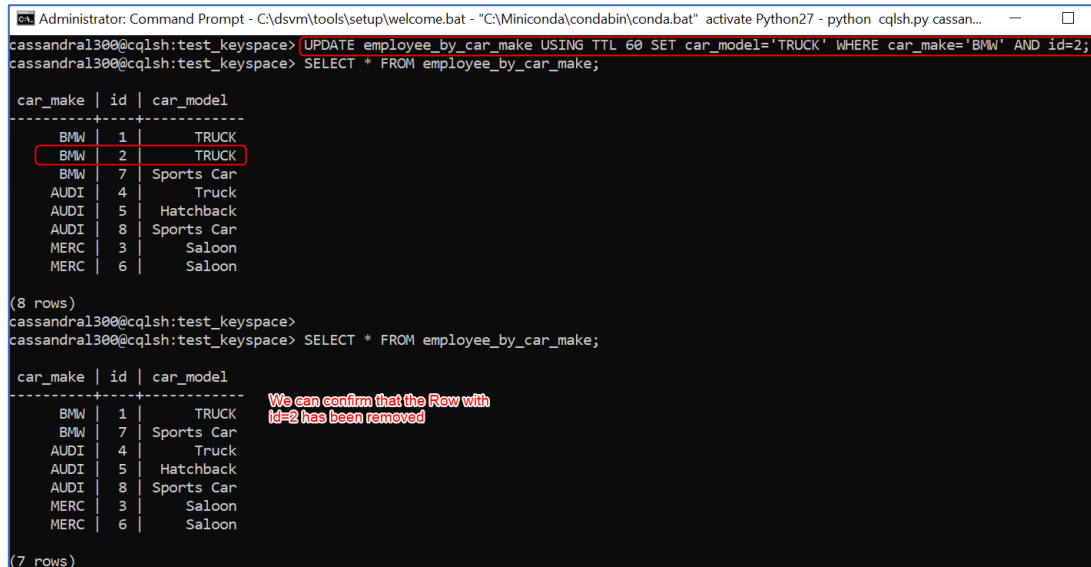
To provide the ability to expire data that is no longer needed we use a time to live function (TTL). In order to demonstrate this will again need to use another UPDATE statement.

1. We will need to run the following UPDATE statement, the time is in seconds.

CQLSH:

```
UPDATE employee_by_car_make USING TTL 60 SET car_model='TRUCK' WHERE car_make='BMW' AND id=2;
```

2. After waiting 60 seconds query all the data in the table **employee_by_car_make** to confirm the data has been deleted.



```
Administrator: Command Prompt - C:\dsvm\tools\setup\welcome.bat - "C:\Miniconda\condabin\conda.bat" activate Python27 - python cqlsh.py cassan...
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_car_make USING TTL 60 SET car_model='TRUCK' WHERE car_make='BMW' AND id=2;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make;

car_make | id | car_model
-----+---+-----
BMW      | 1  | TRUCK
BMW      | 2  | TRUCK
BMW      | 7  | Sports Car
AUDI     | 4  | Truck
AUDI     | 5  | Hatchback
AUDI     | 8  | Sports Car
MERC     | 3  | Saloon
MERC     | 6  | Saloon

(8 rows)
cassandra1300@cqlsh:test_keyspace>
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_car_make;

car_make | id | car_model
-----+---+-----
BMW      | 1  | TRUCK
BMW      | 7  | Sports Car
AUDI     | 4  | Truck
AUDI     | 5  | Hatchback
AUDI     | 8  | Sports Car
MERC     | 3  | Saloon
MERC     | 6  | Saloon

(7 rows)
```

We can confirm that the Row with id=2 has been removed

In Cosmos DB the row just gets deletes, it is not reverted back to its previous value.

In summary, After you set the TTL at a container or at an item level, Azure Cosmos DB will automatically remove these items after the time period, since the time they were last modified. Time to live value is configured in seconds. When you configure TTL, the system will automatically delete the expired items based on the TTL value, without needing a delete operation that is explicitly issued by the client application.

Task IV: Collections (Lists)

In Cassandra we can have a column that holds a collection of data values, for example, in the table **employee_by_id** we can create an additional column to hold all the phone numbers of an employee, and because the employee can have multiple phone numbers it's appropriate to use a collection here.

1. To add a new column to the table, we can use the ALTER command. Here we are going to use a *set* which is an unordered collection of values, or we could also use *list* which is an ordered collection of values.

CQLSH:

```
ALTER TABLE employee_by_id ADD phone set<text>;
```

2. To check if this has worked correctly, we are going to query all the data in the table *employee_by_id*

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position
---+-----+-----
 2 | Bob  | CEO
 1 | John | Manager

(2 rows)
cassandra1300@cqlsh:test_keyspace> ALTER TABLE employee_by_id ADD phone set<text>;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position | phone
---+-----+-----+-----
 2 | Bob  | CEO      | null
 1 | John | Manager  | null

(2 rows)
```

3. Now we want to add the phone number to the *phone* column. So to this we use the UPDATE statement.

CQLSH:

```
UPDATE employee_by_id SET phone = {'123', '456'} WHERE id=1;
```

4. Observe that the phone numbers were added:

CQLSH:

```
SELECT * FROM employee_by_id;
```

We can confirm that the phone numbers for the employee with *id* 1 have been added.

```
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_id SET phone = {'123', '456'} WHERE id=1;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position | phone
---+-----+-----+-----
 2 | Bob  | CEO      | null
 1 | John | Manager  | {'123', '456'}

(2 rows)
```

5. We can add an additional phone number to the same employee:

CQLSH:

```
UPDATE employee_by_id SET phone = phone + {'789'} WHERE id=1;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_id SET phone = phone + {'789'} WHERE id=1;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position | phone
---+-----+-----+-----
 2 | Bob  | CEO      | null
 1 | John | Manager  | {'123', '456', '789'}

(2 rows)
```

6. We can also remove a phone number from that employee. We use the same syntax has before but instead of the "+" we use the "-" sign.

CQLSH:

```
UPDATE employee_by_id SET phone = phone - {'123'} WHERE id=1;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_id SET phone = phone - {'123'} WHERE id=1;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

id | name | position | phone
---+-----+-----+-----
 2 | Bob  | CEO      | null
 1 | John | Manager  | {'456', '789'}

(2 rows)
```

7. We can also remove all phone numbers from the employee by simply setting the phone equal to an empty set.

CQLSH:

```
UPDATE employee_by_id SET phone = {} WHERE id=1;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE employee_by_id SET phone = {} WHERE id=1;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id;

 id | name | position | phone
-----+-----+-----+-----
  2 | Bob  | CEO      | null
  1 | John | Manager  | null
(2 rows)
```

Task V: Secondary Indexes

Secondary Indexes let's access data with a field that is not part of the **Primary Key**.

The Cassandra API in Azure Cosmos DB leverages the underlying indexing infrastructure to expose the indexing strength that is inherent in the platform. However, unlike the core SQL API, Cassandra API in Azure Cosmos DB does not index all attributes by default. Instead, it supports secondary indexing to create an index on certain attributes, which behaves the same way as Apache Cassandra.

1. Let's start by executing the following query using a WHERE clause filtering by *name* which is not part of the **Primary KEY**.

CQLSH:

```
SELECT * FROM employee_by_id WHERE name='John';
```

2. The previous query is just supported if we add to it "ALLOW FILTERING" keywords.

CQLSH:

```
SELECT * FROM employee_by_id WHERE name='John' ALLOW FILTERING;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id WHERE name='John';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id WHERE name='John' ALLOW FILTERING;

 id | name | position | phone
-----+-----+-----+-----
  1 | John | Manager  | null
```

In general, it's not advised to execute filter queries on the columns that aren't partitioned. You must use **ALLOW FILTERING** syntax explicitly, which results in an operation that may not perform well. In Azure Cosmos DB you can run such queries on low cardinality attributes because they fan out across partitions to retrieve the results.

Note: Secondary index is not supported on the following objects:

- data types such as frozen collection types, decimal, and variant types.
- Static columns
- Clustering keys

To find out more details about Secondary indexing in Cosmos DB Cassandra API visit this [document](#).

3. Another way to this, it to add a secondary index to the data. In order to [Create an Index](#) we run the following query:

CQLSH:

```
CREATE INDEX ON employee_by_id (name);
```

After creating an index on the "name" field, you can now run the previous query successfully. With Cassandra API in Azure Cosmos DB, you do not have to provide an index name.

4. Now we should be able to run the query we performed previously without specifying the ALLOW FILTERING:

CQLSH:

```
SELECT * FROM employee_by_id WHERE name='John';
```

```
cassandra1300@cqlsh:test_keyspace> CREATE INDEX ON employee_by_id (name);
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_id WHERE name='John';

id | name | position | phone
---+-----+-----+-----
 1 | John |  Manager |  null

(1 rows)
```

5. UUIDs and Counters

We are going to look at UUIDs or Universally Unique Identifiers and we will look at Counters as well.

In database systems we often want to generate unique value to serve as the identifier or often the **Primary Key** for a row. We can increment a numeric value, this will guarantee the **Primary key** to be unique.

Task I: UUIDs

In Cassandra there are two forms of UUID:

- a **normal UUID**: which we will use when we are storing data and we just want to generate a guaranteed unique value without worrying about ordering
- **timeuuid**: which generates unique value but sorts UUIDs in chronological order so they accessed using time values.

To use a UUID in Cassandra we need to have a table where one of the columns has a type UUID, in most cases this is part of the primary.

1. We will create a table in CQLSH using the syntax we've seen before:

CQLSH:

```
CREATE TABLE employee_by_uuid (id uuid PRIMARY KEY, first_name text, last_name text);
```

2. Once the table is created we can see what is in the table using SELECT statement:

CQLSH:

```
SELECT * FROM employee_by_uuid;
```

3. To insert data into this table we can use the INSERT INTO command. in order to generate a new unique ID for each row we use the function **uuid()**.

CQLSH:

```
INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(),'tom', 'dunne');
```

4. We will confirm if it was successfully inserted.

CQLSH:

```
SELECT * FROM employee_by_uuid;
```

We can see the unique ID has been generated for this row successfully.

```
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(),'tom', 'dunne');
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_uuid;
```

id	first_name	last_name
306f5328-d9e6-4c10-b122-889a835904ef	tom	dunne

(1 rows)

5. We are going to insert some additional rows:

CQLSH:

```
INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(),'tim', 'smith');
INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(),'bob', 'hanson');
```

6. Let's check the rows we have just inserted:

CQLSH:

```
SELECT * FROM employee_by_uuid;
```

```
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_uuid;
```

id	first_name	last_name
306f5328-d9e6-4c10-b122-889a835904ef	tom	dunne
c5736017-3a24-470c-a9f2-d76abbf5ce56	bob	hanson
fa740ff2-8c5e-4424-bc94-937c179e8457	tim	smith

(3 rows)

For every row in the field *id* we get a completely unique identifier

7. Cassandra also supports timeuuid which use the time component of UUID to sort the data in the table. Now, first we create a new table:

CQLSH:

```
CREATE TABLE employee_by_timeuuid (id timeuuid PRIMARY KEY, first_name text, last_name text);
```

8. We can add data as we did before to this table using timeuuid, where this differs slightly from inserting into a UUID instead of using the function *uuid()* we can use the function *now()* to generate a time based UUID.

CQLSH:

```
INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'tim', 'jones');
INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'ally', 'smith');
INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'kate', 'smith');
```

9. Let's check the rows we have just inserted:

CQLSH:

```
SELECT * FROM employee_by_timeuuid;
```

```
cassandra1300@cqlsh:test_keyspace> CREATE TABLE employee_by_timeuuid (id timeuuid PRIMARY KEY, first_name text, last_name text);
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'tim', 'jones');
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'ally', 'smith');
cassandra1300@cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(),'kate', 'smith');
cassandra1300@cqlsh:test_keyspace> SELECT * FROM employee_by_timeuuid;
```

id	first_name	last_name
779d8012-8edb-11ea-d1c7-da6206632941	tim	jones
a3c2c25b-8edb-11ea-2dfa-d59a810b72a1	ally	smith
a3d15162-8edb-11ea-0085-a7170caf3fbd	kate	smith

(3 rows)

Note the *ids* are more similar than compared with the original UUID, this happens because we used the time function *now()* which means they have a similar time component as they were created approximately around a similar time.

Taks II: Counters

A counter is a special type of column which is used to store an integer that changes only in increments or decrements. This is useful for example when you want to know the number of views on a different webpage. Some important points to take in consideration when using Counter:

- Counters can only be created in dedicated tables and they can't be assigned to the column that serves as Primary or Partition Key.
- We cannot index or delete a counter column.
- Tables that represent a counter usually consist of the **Primary Key** and then the Counter column itself, and this will have the Counter data type.

1. In Cassandra we use the table syntax we have seen before:

CQLSH:

```
CREATE TABLE purchases_by_customer_id (id uuid PRIMARY KEY, purchases counter);
```

2. Interesting thing about Counter is that data is Updated rather than Inserted into counter columns and counter columns cannot be set only incremented or decremented. Therefore, to add a new row to our table **purchases_by_customer_id** we need to use the UPDATE command:

CQLSH:

```
UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
```

3. The last command should insert into our table a row with a unique randomly generated identifier and the number of purchases should be equal to 1.

CQLSH:

```
SELECT * FROM purchases_by_customer_id;
```

```
cassandra1300@cqlsh:test_keyspace> CREATE TABLE purchases_by_customer_id (id uuid PRIMARY KEY, purchases counter);
cassandra1300@cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
cassandra1300@cqlsh:test_keyspace> SELECT * FROM purchases_by_customer_id;
```

id	purchases
691b760b-ee1e-46c9-9e0e-7620282ac386	1

(1 rows)

4. Use the same command to insert a number of different customers into our table:

CQLSH:

```
UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
```

5. Run the query to check all the rows in the table again:

CQLSH:

```
SELECT * FROM purchases_by_customer_id;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
cassandra1300@cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id=uuid();
cassandra1300@cqlsh:test_keyspace> SELECT * FROM purchases_by_customer_id;
```

id	purchases
f71dfc3b-50eb-4d45-bc25-eb4fbd26bc9	1
bb743c59-580f-4a7d-bb01-678d52f19895	1
691b760b-ee1e-46c9-9e0e-7620282ac386	1

(3 rows)

6. If we want to increment the counter for a particular customer, we need to specify the *id* in the WHERE clause. So, we use a similar statement as before. Grab one of the *ids* from the table and change it in the query below:

CQLSH:

```
UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id= 691b760b-ee1e-46c9-9e0e-7620282ac386;
```

7. Again, run the query to check all the rows in the table again:

CQLSH:

```
SELECT * FROM purchases_by_customer_id;
```

```
cassandra1300@cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 where id= 691b760b-ee1e-46c9-9e0e-7620282ac386;
cassandra1300@cqlsh:test_keyspace> SELECT * FROM purchases_by_customer_id;
```

id	purchases
f71dfc3b-50eb-4d45-bc25-eb4fbd26bc9	1
bb743c59-580f-4a7d-bb01-678d52f19895	1
691b760b-ee1e-46c9-9e0e-7620282ac386	2

(3 rows)

8. We can also simply decrement the value of *purchases* by setting *purchases=purchases-1*.

6. Importing and Exporting Data

Task I: Importing

We are going to copy data from CSVs to Cassandra (importing), to do this the table must already exist in Cassandra. We must have a table where at least some of the columns in the table line up to some of the columns in the CSV file.

We must ensure that every **Primary Key** column in the database is satisfied by a column in our CSV for instance if the primary key in this case was made up of *car_make*, *car_model* and *start_year* in the table we must ensure that our CSV has at least those three columns, any column that's missing in the CSV but exists in the table that is not part of the **Primary Key** will be set to null when we import the data.

1. Copy the CSV **Cassandra.csv** to your Local Computer (in this lab we are going to copy the CSV file to our Desktop). We are going to use it to import data into Cassandra.

Link: <https://elearning.novaims.unl.pt/mod/folder/view.php?id=34165>

If you cannot download the file from here directly to your computer, you can copy this link and paste it in your computer browser to download the file directly

https://elearning.novaims.unl.pt/pluginfile.php/88741/mod_folder/content/0/cassandra.csv?forcedownload=1

2. So, in order to do this, we'll jump back into CQLSH or and create the table we need to import the data. Notice that we are using the same column names that we have in the CSV.

CQLSH:

```
CREATE TABLE test_csv_import (car_make text, car_model text, start_year int, id int, first_name text, last_name text, department text, PRIMARY KEY(car_make, car_model, start_year, id));
```

3. We can begin importing the data from the CSV into our newly created table. We also need to know the location of our CSV in order to import it into Cassandra (in this lab we copied the file *cassandra.csv* to the Desktop). We will need to specify if the DELIMITER and HEADER.

CQLSH:

```
COPY test_csv_import (car_make, car_model, start_year, id, first_name, last_name, department) FROM  
'C:\Users\anferrei\Downloads\cassandrascv\cassandra.csv' WITH DELIMITER=';' AND HEADER=TRUE;
```

In the output you should get a message saying the rows have been imported from 1 file. You might get some errors or warnings, however, it should be absolutely fine.

4. Let's check that the import was successful by running:

CQLSH:

```
SELECT * FROM test_csv_import;
```

```
Processed: 5 rows; Rate:      1 rows/s; Avg. rate:      2 rows/s  
5 rows imported from 1 files in 3.071 seconds (0 skipped).  
cassandral300@cqlsh:test_keyspace> select * from test_csv_import;  
  
car_make | car_model | start_year | id | first_name | last_name | department  
-----+-----+-----+---+-----+-----+-----  
LEXUS    | Sports    | 2011       | 3  | Pit        | IT        | Brad  
BMW      | Saloon    | 2011       | 1  | Depp       | IT        | Johnny  
AUDI     | Saloon    | 2013       | 2  | Streep     | HR        | Meryl  
NISSAN   | Saloon    | 2013       | 9  | Roberts    | FI        | Julia  
(4 rows)
```

For more details about Migrating data using CQLSH COPY command in Cosmos DB you get go to the following [document](#).

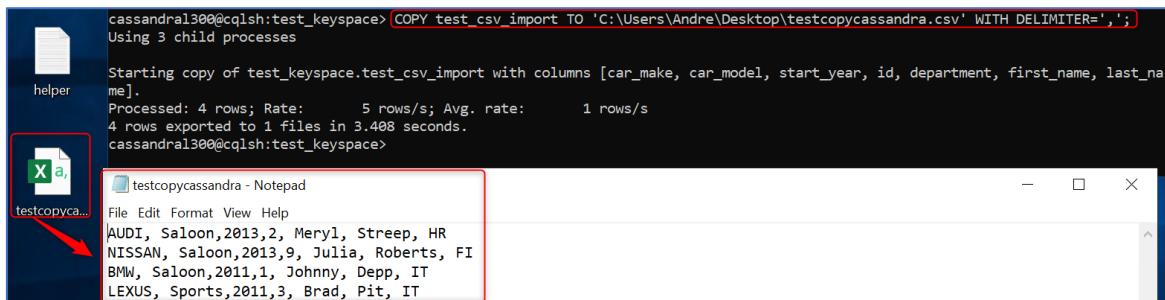
Task II: Exporting

1. To export you just need to run the below command (you need to change the TO to your local machine Desktop path):

CQLSH:

```
COPY test_csv_import TO 'C:\Users\Andre\Desktop\testcopycassandra.csv' WITH DELIMITER=';';
```

2. Go to your Desktop and confirm that the file was created there.



3. You can also specify what you want to export. If we just want to export *car_make*, *department* and *first_name*, for example, we just need to execute the following command:

CQLSH:

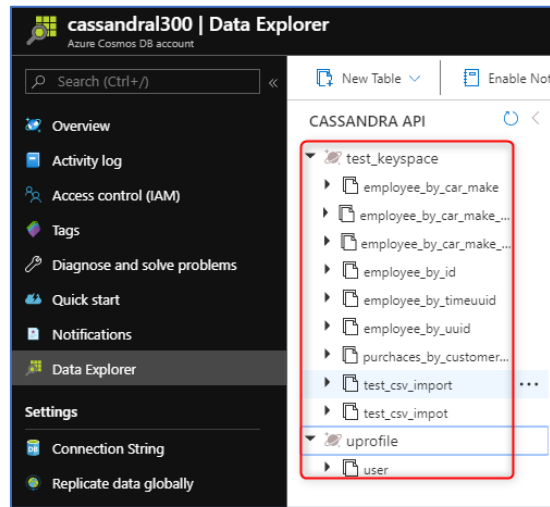
```
COPY test_csv_import (car_make, department, first_name) TO  
'C:\Users\anferrei\Downloads\cassandrascv\testcopycassandra2.csv' WITH DELIMITER=';';
```

4. Go to the local machine to the folder you selected and check the file *testcopycassandra2* has been created successfully with just 3 columns.

Clean Up Resources

JUST CLEAN THE RESOURCES AFTER ANSWERING THE QUESTIONS IN THE MOODLE

In the end of these labs you should have the following Keyspaces and respective Tables:



When you're done with the Challenge 3 Quiz, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

You may **delete all the tables for this sample**, or delete the **test_keyspace** or just delete the Cosmos DB Cassandra API account. This ensures that you avoid being charged for the resources when you are not using them. You can also delete the **Resource group**. This ensures that you avoid being charged for the resources when you are not using them.

To delete the Resource Group:

1. In the Microsoft Azure portal, click Resource Groups.
2. On the Resource groups blade, click the resource group that contains your Cosmos DB accounts and then on the Resource group blade, click Delete.
3. On the confirmation blade, type the name of your resource group, and click Delete.
4. Wait for your resource group to be deleted, and then click All Resources, and verify that the Cosmos DB account that was created has been removed.
5. Close the browser.