Homework 3: Cassandra

We covered most of the tutorial in the lab session. We did not go over UUIDs, UTDs, or time series (pages 11 – 12) There are also a few additional tasks that I would like you to go through before starting with the deliverables (pages 14 -). Deliverable explained on the last two page.

Installing or running Cassandra

Installing Cassandra

You need

- At least 1 GB RAM
- A 64-bit processor
- Java 7 or 8 (you can check by typing `java -version` in a command prompt (download from Oracle if you do not have Java on your system).

Since I am installing Cassandra on a Windows (Windows 7, 64-bit system, with Java 1.8.0_66, and 4 GB of RAM), I will use a third party distribution from Datastax (http://www.planetcassandra.org/cassandra/). The third party distribution is also available for other systems, but you may also choose to download Cassandra directly from http://cassandra.apache.org/download/. You may then want to follow the installation guidelines that can be found here: http://wiki.apache.org/cassandra/GettingStarted.

If you want to use Datastax distribution (which I recommend), you can follow my installation guide:

1. Download the MSI installer for the current edition (in this case v. 3.2.1).
2. Double-click on the MSI installer and follow the steps. I used the default settings.
3. I did not register at the end.
4. In your start menu, you should see 'Cassandra CQL Shell' (cqlsh). If not, it will be in the DataStax Distribution of Apache Cassandra folder. Click on it and it will open the CQL shell. CQL stands for Cassandra Query Language and the CQL shell is the easiest and most primitive way to interact with Cassandra. You are done. This was easy. (This course should be called: a new query language every week. The good thing about this week is that CQL is very, very close to SQL. You will see.)

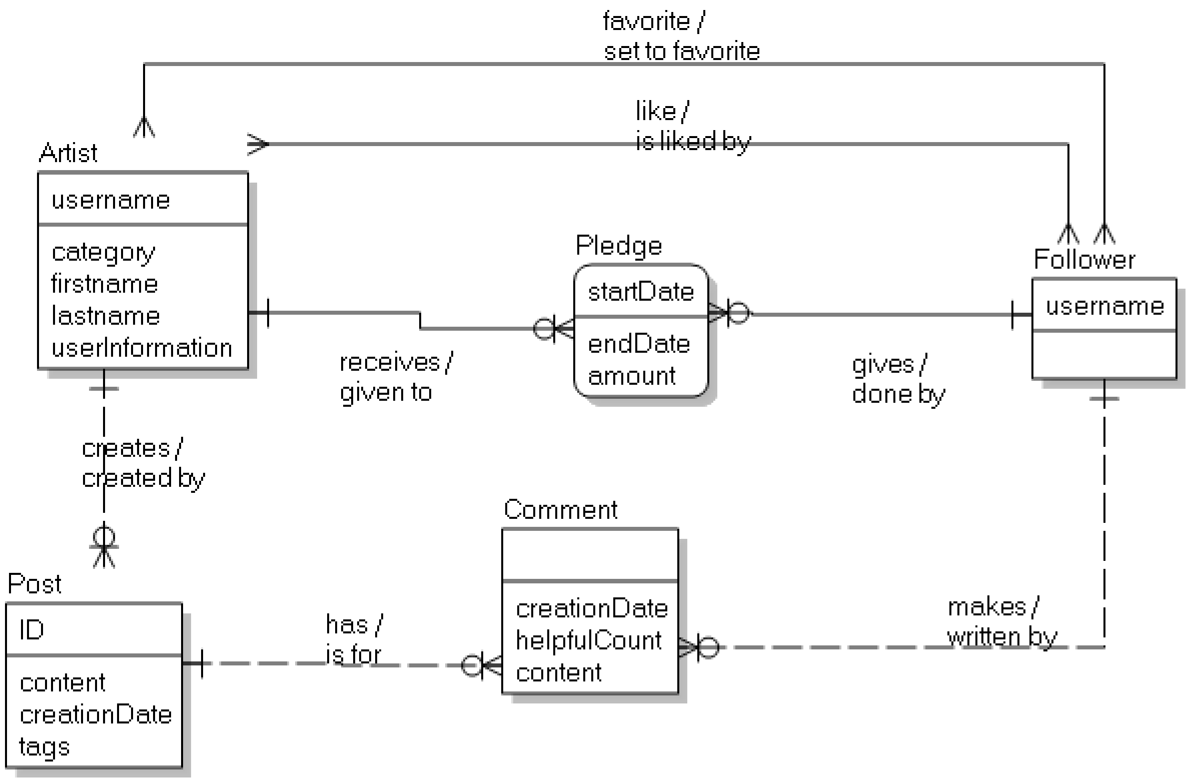
Running Cassandra at PKI

Cassandra will be available on the Student Computers that are located next to the Scott Café Express. In your start menu, you should find a folder Datastax Distribution with the Cassandra CQL Shell in it. Let me know if you are having troubles.

Note: The tutorial was created based on the documentation provided by Apache Cassandra (http://cassandra.apache.org/doc/cql3/CQL.html) and Datastax. Datastax has an excellent documentation (http://docs.datastax.com/en/cql/3.3/cql/cqlIntro.html) as well as free online tutorials that are excellent. I can recommend the tutorials to anyone who wants to delve deeper into Cassandra. Things are changing a lot from one version to the other, so make sure you have the current release as well as the matching documentation. When you go through this tutorial, copy & paste or type the commands into the cqlsh and try them out! This tutorial is meant to go through from the beginning to the end and the explanations build on each other. So, do not jump ahead, but go through it one step after another.

Data Model

The following ERD shows our first draft of the conceptual data model that we are using to explore Cassandra further. We have artists who can create postings. Followers who want to support artists can pledge an amount of money to artists. Pledges have a start date and an end date. Followers can also like artists or favorite them. Follower can make comments for postings.



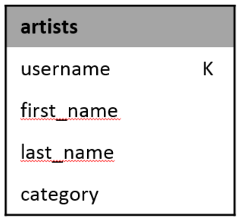
In Cassandra, writes are cheap. One reason is that Cassandra is not doing look-ups before writing, but simply writes the new data. The use of timestamps helps to resolve conflicts. So, we do not worry about duplicate data and if an extra write improves our read performance, we will rather have data duplication. Since Cassandra does not allow certain operations if they make reads slower, we need to know how Cassandra stores the data physically and match our queries to how we store data. Our data model for Cassandra will be query driven. We will have to write tables to answer queries even if that means data duplication. Joins are not possible in Cassandra, so instead, we will need to denormalize. Btw. joins are not done, because data is distributed across different partitions (chunks of data), and partitions are distributed across servers (nodes). Joining partitions would be very slow and is thus not done. So, if we need data that would reside in two or more tables in a relational data model, we need to denormalize. This will add additional duplication. We do not care about this as long as we can optimize for read performance.

So, in summary, in order to answer a query, we need to construct tables that minimize the number of partitions we read. We will see how this is done with the following examples.

Creating a Table with a simple Primary Key

Our first query will simply lookup artists by username.

Q1: Find artists by username.



Before we start creating a table, we need to create the keyspace which is comparable to a database.

In cqlsh type in

```
CREATE KEYSPACE patron WITH REPLICATION = { 'class' : 'SimpleStrategy',
'replication_factor' : 1 };
```

We can see what kind of keyspaces we have with `DESCRIBE keyspaces;`

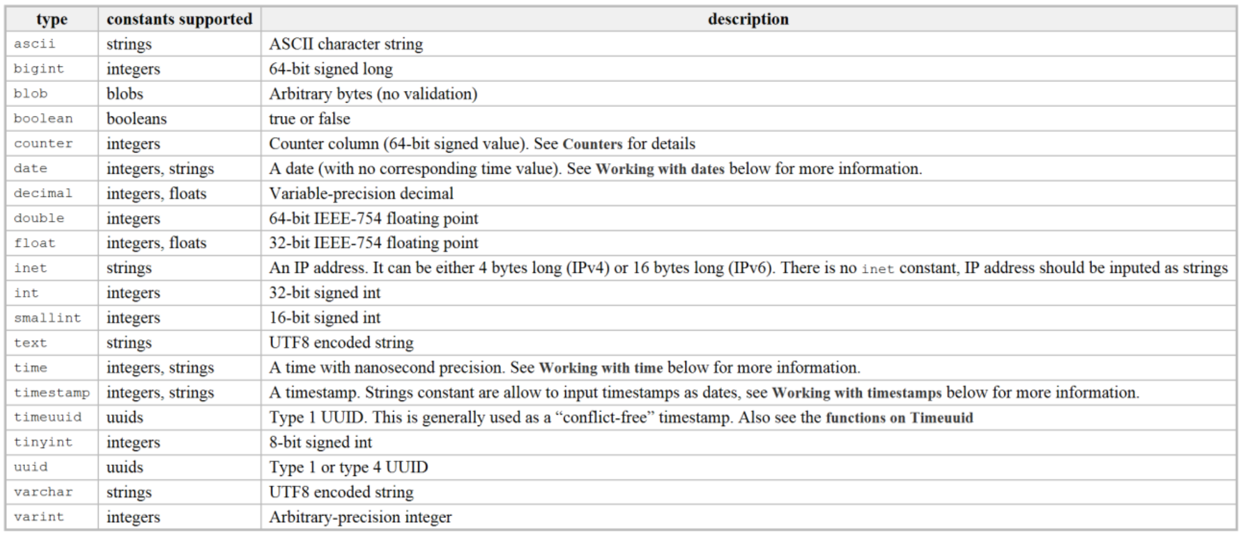
In order to switch to the newly created keyspace type `USE patron;`

The table creation is very similar to the SQL command. In Cassandra, a table is the equivalent to a column family. Cassandra has changed terminologies to be more familiar to SQL developers. The table we create here, is not the same as a relational table in a relational database.

```
CREATE TABLE artists (

 username text PRIMARY KEY,

 first_name text,

 last_name text,

 category text

);
```

Similar to SQL statements, the statement can span multiple lines and will end with a semicolon. If you forget the semicolon, you will see three points indicating that Cassandra is waiting for more lines in the statement or a semicolon. You can paste the above statement into cqlsh (right-click on the top bar, Edit -> Paste).

Data types: Cassandra supports different data types including user-defined types, universal unique ids, and counters. Following a table from http://cassandra.apache.org/doc/cql3/CQL.html.



type	constants supported	description
`ascii`	strings	ASCII character string
`bigint`	integers	64-bit signed long
`blob`	blobs	Arbitrary bytes (no validation)
`boolean`	booleans	true or false
`counter`	integers	Counter column (64-bit signed value). See **Counters** for details
`date`	integers, strings	A date (with no corresponding time value). See **Working with dates** below for more information.
`decimal`	integers, floats	Variable-precision decimal
`double`	integers	64-bit IEEE-754 floating point
`float`	integers, floats	32-bit IEEE-754 floating point
`inet`	strings	An IP address. It can be either 4 bytes long (IPv4) or 16 bytes long (IPv6). There is no `inet` constant, IP address should be inputed as strings
`int`	integers	32-bit signed int
`smallint`	integers	16-bit signed int
`text`	strings	UTF8 encoded string
`time`	integers, strings	A time with nanosecond precision. See **Working with time** below for more information.
`timestamp`	integers, strings	A timestamp. Strings constant are allow to input timestamps as dates, see **Working with timestamps** below for more information.
`timeuuid`	uuids	Type 1 UUID. This is generally used as a "conflict-free" timestamp. Also see the **functions on Timeuuid**
`tinyint`	integers	8-bit signed int
`uuid`	uuids	Type 1 or type 4 UUID
`varchar`	strings	UTF8 encoded string
`varint`	integers	Arbitrary-precision integer

Verify that the table was created with `DESCRIBE TABLES;`

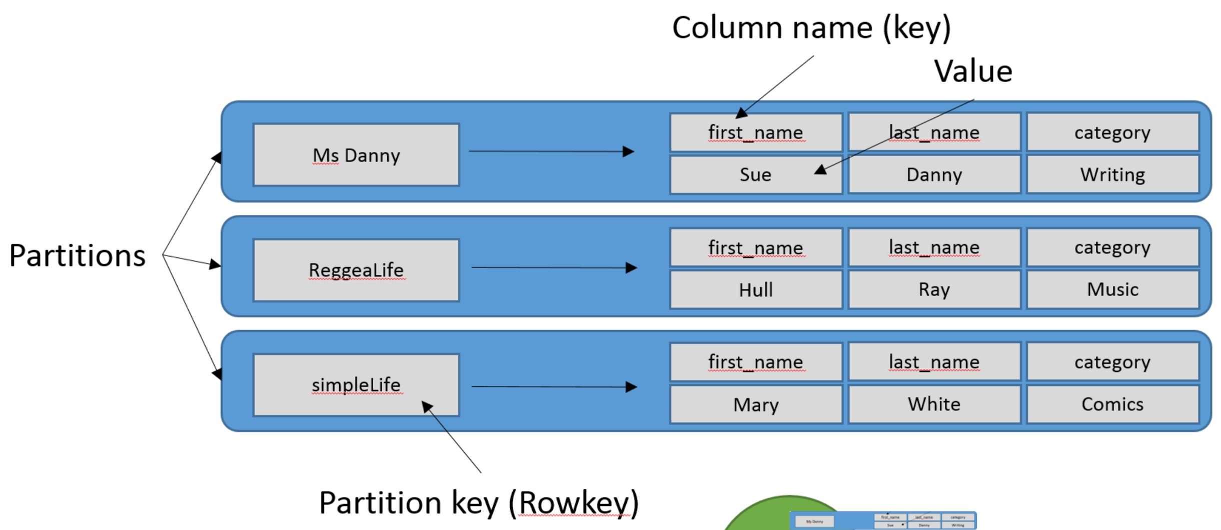
You can review the table structure with `DESCRIBE artists;`

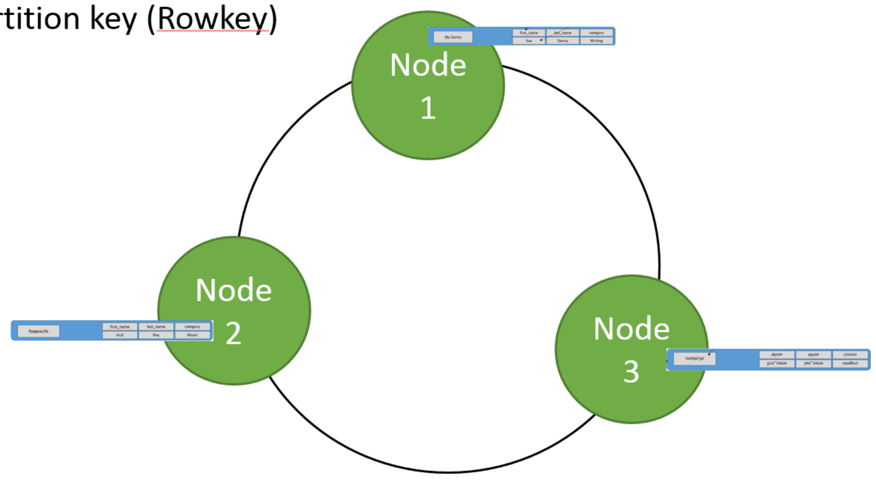
Inserting data

Inserting data into the new table is simple

```
INSERT INTO artists (username, first_name, last_name, category)

 VALUES (    'simpleLife', 'Mary', 'White', 'Comics' );

INSERT INTO artists (username, first_name, last_name, category)

 VALUES (    'ReggaeLife', 'Hull', 'Ray', 'Music' );

INSERT INTO artists (username, first_name, last_name, category)

 VALUES (    'Ms Danny', 'Sue', 'Danny', 'Writing' );
```

The column username is the primary key. A primary key defines the location, partition, and order of the data. The primary key consists of only one column and is called a simple primary key. In this case, the primary key is the partition key and determines how the data in the table is partitioned. Every row is one partition. The physical storage will look like this:



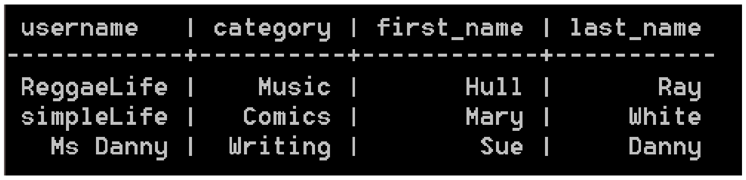


Every row is a partition and will be stored separately on a node. A partition will be stored in whole on one node (and probably on more, depending on the replication factor that is chosen). Thus, a partition cannot be larger than the storage capacity of a node. Of course, multiple partitions can be stored on one node as well. So, in the above example, all three partitions could be stored on one node or one on each node. The partition key is the primary key and will determine on which node the partition is stored. There are several methods that determine how partitions are distributed. The default is a hash function which is applied to the partition key.

Querying data

Now, let's select the data we just entered with a `SELECT * FROM artists;`

Cassandra gives us the results in a nice tabular way.



```
username   | category | first_name | last_name
------------+----------+------------+-----------
ReggaeLife |    Music |       Hull |       Ray
simpleLife |   Comics |       Mary |     White
  Ms Danny |  Writing |        Sue |     Danny
```

Try out to select a certain number of columns or count or use WHERE such as

```
SELECT first_name, last_name FROM artists;
SELECT count (*) FROM artists;
```

```
SELECT * FROM artists WHERE username = 'Ms Danny';
```

So far, so good. This is familiar. Now, try the following:

```
SELECT DISTINCT category FROM artists;
```

```
SELECT username FROM artists WHERE category = 'Music';
```

Okay, this is where we have to be careful and not treat the tables in Cassandra like tables in a relational database. The first error message indicates that we can use distinct only with a partition key or static column. The second error message indicates that there is no support for secondary indexes. So, why did we get these error messages? Remember, Cassandra distributes the data in partitions across different nodes. If we lookup data according to the primary key, Cassandra can run the hash function and knows on which node the data is located. However, there is no way for Cassandra to know where data within a non-primary key column is located. Cassandra would need to go to each node and then search each partition. That is obviously not efficient. Same with distinct that is used on a non-primary key column. So, is the solution a secondary index? Now, remember, in relational databases secondary indexes are used to quickly access data in columns and avoid a whole table scan. So, should we just add a secondary index on the columns we need? No. The problem is that primary indexes are global but secondary indexes local. This means, Cassandra would still need to query each node. Not all partitions, but still each node. Again, not efficient. So, in summary. If you query on any field other than the primary key, Cassandra needs to search all partitions on all nodes. With secondary indexes, Cassandra will need to search all nodes. So, the solution is to change our table. Or to create a new table.

Q2: Find artists by category.

How do we do this? How about the following table:

```
CREATE TABLE artists_by_category (
 category text PRIMARY KEY,
 username text,
 first_name text,
 last_name text
);
```

Before we continue, let's learn how to upload data in a CSV file.

Download the artist_by_category.csv file from Blackboard and download into apache_cassandra/bin of your Datastax distribution (For me, that was c:/Program Files/Datastax-DCC/apache_cassandra/bin. Then, in cqlsh type in

```
copy artists_by_category FROM 'artists_by_category.csv' WITH DELIMITER = ',' AND
HEADER=TRUE;
```

Now, open the artists_by_category.csv file and see how many rows are imported. Then, do a

```
SELECT * FROM artists_by_category;
```

Did the rows match?

Why not? Well, obviously we have only the category as primary key. So, we only expect to see unique values here. But why didn't we get an error warning? Shouldn't a primary key be unique? Yes, but remember, Cassandra does not read before write. So, Cassandra will not verify whether a value already exists. Cassandra does not differentiate between

INSERT and UPDATE. It does an UPSERT. UPSERT means that a row will be updated whether it exists or not. If it exists, then the new value is valid. If the row does not exist, then a new row will added. So, the first time we inserted a new category a new row is written. The next row with the same value for category updated the first row (well, actually it did insert a new row, but we only retrieve the latest one – remember, we store a timestamp with the values). So, that did not work. (Btw. uniqueness of the primary key is thus something the application needs to take care of because Casandra will not be able to enforce it. The same could happen with a new artist. If an artist decides to have the same name as an already existing one, then Cassandra will not give an error message but simply add the row with the new information.)

Creating a table with a compound Primary Key

Let us drop the table with

```
DROP TABLE artists_by_category;
```

and try again with the following statement.

```
CREATE TABLE artists_by_category (
 category text,
 username text,
 first_name text,
 last_name text,
 PRIMARY KEY ((category, username))
);
```

Here, we specify that the primary key consists of the two fields category and username. Let's upload the data again with

```
copy artists_by_category FROM 'artists_by_category.csv' WITH DELIMITER = ',' AND
HEADER=TRUE;
```

and do a

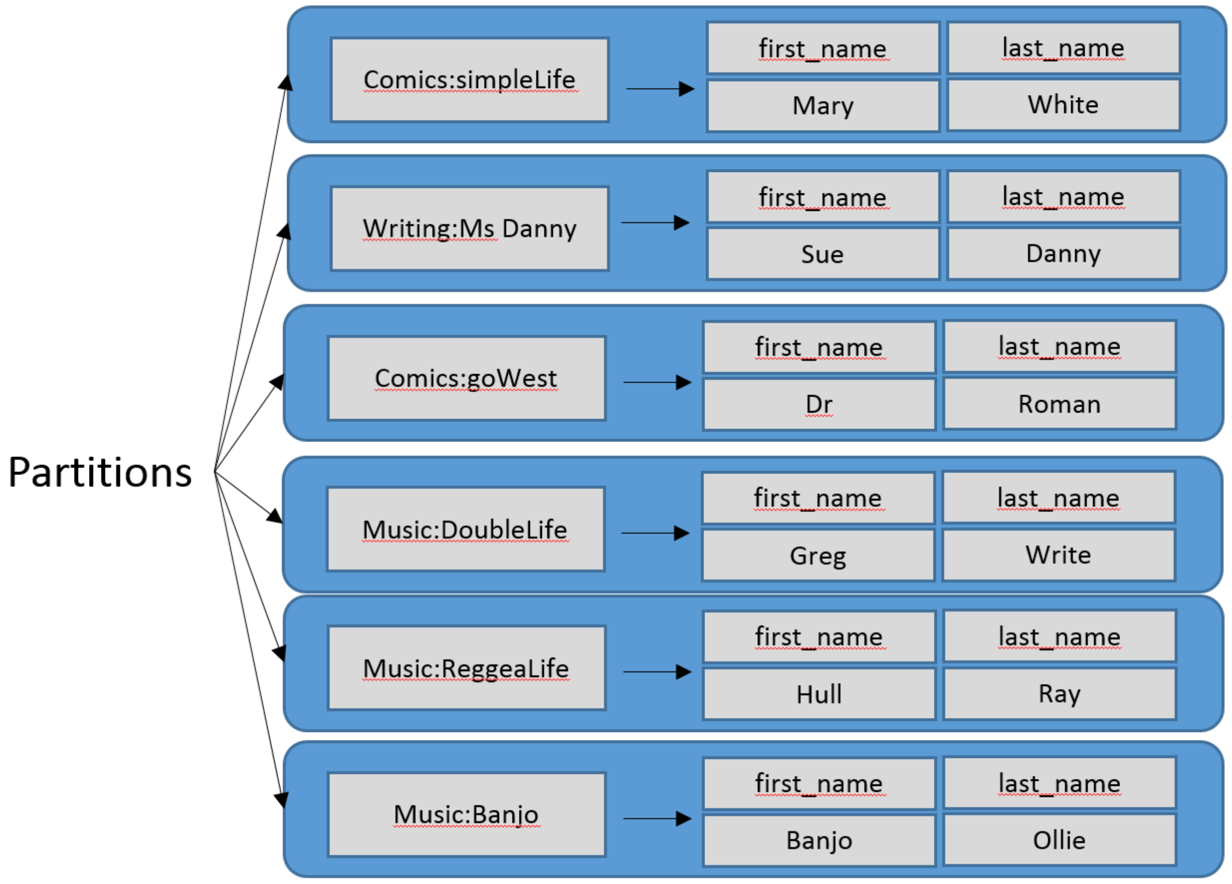
`SELECT *` and `SELECT COUNT(*);`

This time it worked. We uploaded six rows and we can select or count six rows. Let's try to search for all artists in the category Music.

```
SELECT * FROM artists_by_Category WHERE category = 'Music';
```

Did it work? No. Why not?

Here, we need again consider the physical storage of the data in Cassandra. Each partition is determined by the partition key. We determined that the partition key be the combination of category and username (since category alone did not work). This means, every partition's row key will be the combination of category and username as depicted here.



Six partitions. Every combination of category and username is one partition. The problem is that the partition is determined by the partition key. Thus, the hash function to determine partitioning and distribution of partitions requires both category and username. We cannot apply the hash function to only one part of the partition key. So, this is did not work either. What is the solution? What we want is to have one row for each category. Since we want to efficiently read, our goal should be to only have to access one partition for a query. So, we want to put each category in one partition. Then we will only need to read on partition if we want to retrieve the artists for one category. The partition key needs to be category. But we tried this before and it did not work. The solution is called a clustering key. A clustering key groups rows in one partition together. This is how the table looks like.

artists_by_category	
category	K
username	C↓
first_name	
last_name	

Drop the table and type in the new create table statement.

```
CREATE TABLE artists_by_category (
 category text,
 username text,
 first_name text,
 last_name text,
 PRIMARY KEY ((category), username)
 );
```

This is an example of a compound primary key. There are two parts to it. The first one will be used as the partition key (in this case the additional parenthesis are actually not needed, but you could have multiple attributes for the partition key and the clustering key). The second one is the clustering key that groups rows. (Now it probably also makes sense why we had two parenthesis around category and username in the statement that did not work above. We defined that the partition key is category and username. If we had left out one pair of parenthesis, then Cassandra would treat the first column as partition key and the second column as clustering key.)

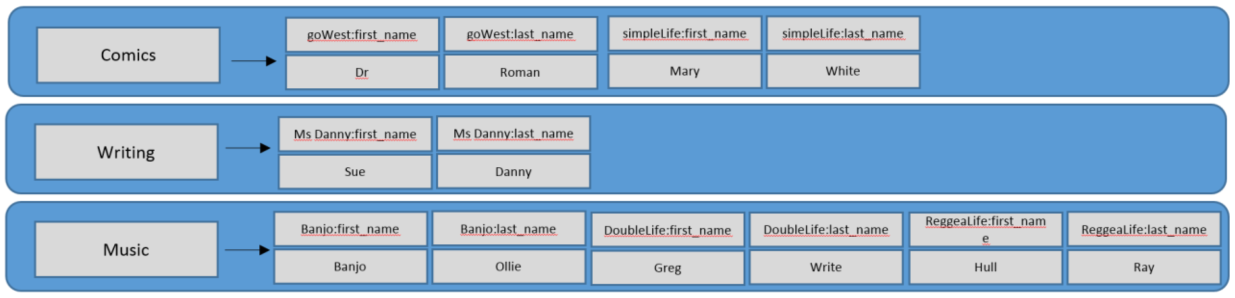
Copy the data from the .csv file and

```
SELECT * FROM artists_by_category;
```

All six rows are there. Then do a

```
SELECT * FROM artists_by_category WHERE category = 'Music';
```

Finally. So, how does Cassandra store the data? Since the partition key is category only, each row will contain all data for one category. The column names are interesting. The username will be used to group and order the artists. See how the column names appear in the figure.



If you type in

```
DESCRIBE artists_by_category;
```

you will see that the rows are clustered by username ASC. You could specify a different order during table creation.

We can now also do a

```
SELECT DISTINCT category FROM artists_by_category;
```

What we can also do is to range queries on the clustering columns, e.g.

```
SELECT * FROM artists_by_category where category = 'Music' AND username>'Banjo';
```

We cannot do range queries on any other columns, though. Here, like with the lookup on non-clustering or non-primary key columns, we would need to search all partitions. This is not what Cassandra allows.

Collections

Let's expand a bit on the data model. Let's assume we want to store the year an artist started and the top supporters. The year will be an int and the supporters will be a set. A set stores a list of unique values. A set is a type of collection and meant to store a small set of data.

First, we alter the artists table.

```
ALTER TABLE artists ADD start_year int;

ALTER TABLE artists ADD top_supporters set <text>;
```

Download the csv file artists.csv and store into your apache-cassandra/bin folder. Use the following command in cqlsh to upload the data into the artists file.

```
copy artists FROM 'artists.csv' WITH DELIMITER = ',' AND HEADER=TRUE;
```

Counter

Let's include a new column number_of_likes an artist receives. We can implement that with a counter. A counter column allows two operations – incrementation and decrementation. When you work with a counter in Cassandra, you can only use the update statement. A counter is initialized the first time it is incremented/decremented and treated as it were zero.

If you work with a counter then the only table can only contain counters as non-primary key columns. No other column is allowed. Either the columns belong to the primary key or are counters.

Let's create the table and add two counters. One that count the number of likes an artist receives. And one that counts the number an artist was set as a favorite.

Q3: Increase the number of likes for an artist.

Q4: Increase the number of favorite for an artist.

```
CREATE TABLE artist_counts (
 username text PRIMARY KEY,
 number_of_likes counter,
 number_of_favorites counter
);
```

We can now use the counter by using UPDATE.

```
UPDATE artist_counts SET number_of_likes = number_of_likes + 1 WHERE username =
'simpleLife';

SELECT * FROM artist_counts WHERE username = 'simpleLife';
```

Note: The other column indicates NULL. But remember, Cassandra does not store anything if no value exists. So, the counter for number_of_favorite does not exist yet. Also note, that we used UPDATE to insert a new row.

You can run more update queries to increase the counters.

This is all for now. More in part 2. You will learn more about user defined types, UUID, and maybe time series. You will also learn about the DevCenter from DataStax which is a very nice visual query client for Cassandra. And of course, some more hands on tasks for you to do.

User-defined types (UDT)

User-defined types allow you to specify multiple, related fields in a column. The classical example is an address.

```
CREATE TYPE address (
     street text,
     city text,
     state text,
     zip int
);
```

Let's change the table artists to include an address.

```
ALTER TABLE artists ADD address frozen <address>;
```

The keyword frozen means that you cannot only change part of the UDT address, but you need to change the whole address. The current release of CQL does not allow to update only part of an UDT.

You can also use UDTs in combination with collections. For example, if we want to add different types of credit cards, we could first define the type and then add a set of credit cards.

```
CREATE TYPE credit_card (
     number bigint,
     cvv int,
     name_on_card text
);
ALTER TABLE artists ADD credit_cards set <frozen <credit_card>>;
```

You can also nest UDT, e.g., adding an expiration month and year to the credit card.

```
CREATE TYPE expiration_date (
     month text,
     year text
);
ALTER TYPE credit_card ADD expiration_date frozen <expiration_date>;
```

We can now insert a new artist.

```
INSERT INTO artists (username, address, category, credit_cards, first_name, last_name)
VALUES ('squirrel',
      { street: '1523 Main', city: 'Omaha', zip: 68185, state: 'NE' },
      'Music',
      {
            { number: 1423123124132343, name_on_card: 'John', cvv: 2243,
expiration_date : { month: '04', year: '15' }},
            { number: 1412312312332342, name_on_card: 'John', cvv: 24234}
      },
      'John',
      'Gartner'
);
```

Since UDT need to be updated completely, you shouldn't use them if parts of the UDT need to be updated. I can see the best fit when you want to use complex data structures in combination with collections such as using a map data type to allow multiple types of addresses (home, shipping, etc.) or using a set to store multiple credit cards as demonstrated in the example.

UUID

An UUID (Universally unique identifier) is a 128-bit-value and allow to implement an artificial (database-generated) key. UUID are used in distributed systems to uniquely identify rows. Coordination of keys is difficult in distributed systems since we would need coordination among nodes to make sure that a number has not been used or is not currently assigned at the same time by another node. With 128-bit it is unlikely that two row will have the same number due to the sheer large amount of possible numbers.

Cassandra also has a timeuuid that contains a time component. Timeuuid columns are stored chronologically in Cassandra which is an advantage if we want to retrieve events in a chronological order.

Let's create the table posts with a UUID.

```
CREATE TABLE posts (

      post_id uuid PRIMARY KEY,

      content text,

      creation_date timestamp,

      tags set <text>

);
```

We can insert a new value into the table.

```
INSERT INTO posts (post_id, content, creation_date, tags)
VALUES ( uuid(), 'Hi, this is my first post.', '2016-02-28 20:16', {'Welcome',
'First'})
```

Use `SELECT * FROM posts;` to view the UUID in a hexadecimal form.

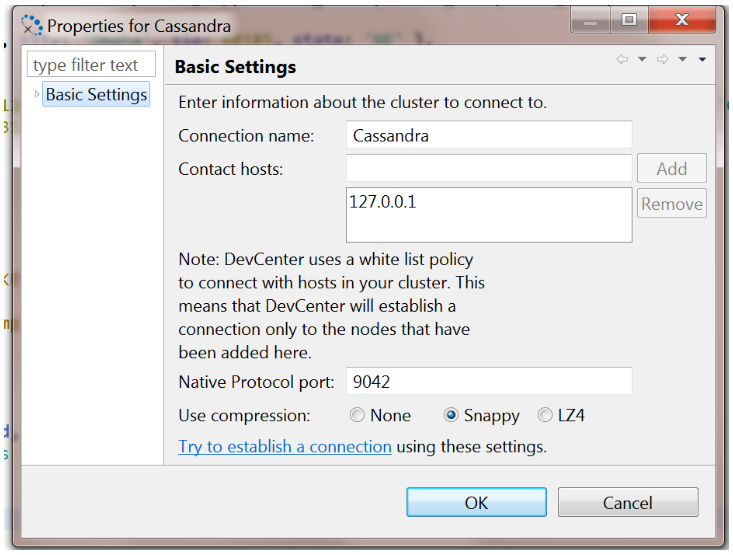
Time Series

Cassandra works well if you want to store many time values. Since I couldn't really find a good use case for our example, I refer to the documentation in Datastax (e.g., https://academy.datastax.com/demos/getting-started-time-series-data-modeling). The idea is to use a timestamp as the clustering key which allows to sort and retrieve timestamps and their values sequentially. Partitions would depend on how many values you are expecting. E.g., for a stock market analysis of daily values, you could have the stock identifier as the partition key which would then store one stock in a partition and all daily values in the columns. For intra-day values, you could use the stock identifier and the day as one partition, and again the timestamp in the columns.

DevCenter – a query tool for Cassandra

Working with the cqlsh can be a bit bothersome. Datastax has a nice query tool which gives you some of the nice features we come to expect from query tools such as error messages, formatting, etc.

You can download Datastax DevCenter from http://www.datastax.com/what-we-offer/products-services/devcenter. I simply followed the instructions and had no trouble installing the product. You should also find them at PKI on the computers next to Scott Café Express. Find the developer in the same directory as the cql shell. When you open DevCenter first you need to create a new connection. These are the settings that worked for me:



Properties for Cassandra

type filter text

Basic Settings

Basic Settings

Enter information about the cluster to connect to.

Connection name: Cassandra

Contact hosts: 127.0.0.1

Note: DevCenter uses a white list policy to connect with hosts in your cluster. This means that DevCenter will establish a connection only to the nodes that have been added here.

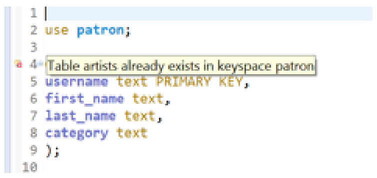
Native Protocol port: 9042

Use compression: None [✓ Snappy] LZ4

Try to establish a connection using these settings.

The DevCenter has example scripts for a video database and you can look over them and run some of the queries if you want. It is my understanding that the scripts are supposed to be run completely. If you want to only run selected queries, then you need to highlight the query or queries and then click on run. I found this quite cumbersome as I am used to SQL developer where I just need to place my cursor inside a query and then run it. I did not find another way to run single queries. Let me know if you find a better way.

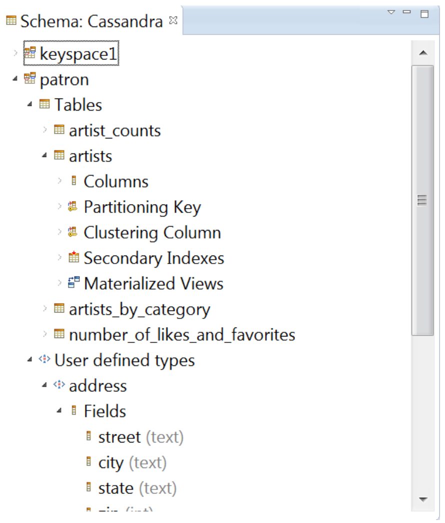
You may get error signs on the left. If you hover your mouse arrow over them, they will tell you what's wrong. Sometimes, an error is shown because the script is supposed to be run completely, e.g., when a table is already defined. You can ignore these error messages. An example is shown in the next picture.



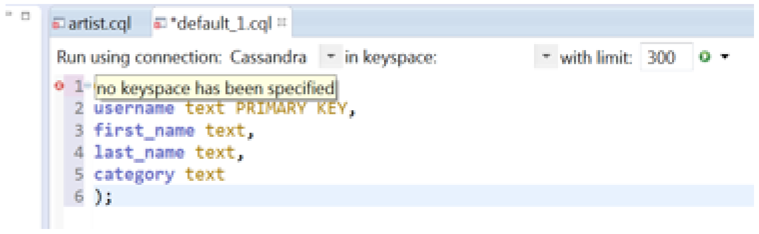
```

use patron;

Table artists already exists in keyspace patron
username text PRIMARY KEY,
first_name text,
last_name text,
category text
);

```

In general, I found the error messages to be very helpful.

On the right, you can see your keyspaces, tables, and other objects you created. From here, you can also change or drop tables or columns if you want.

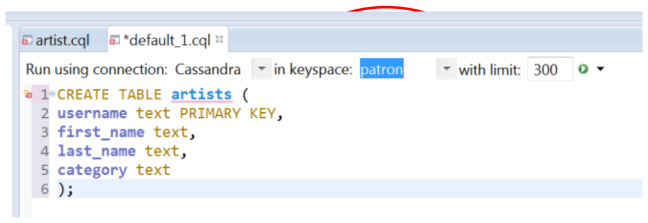


The first thing if you create a new script is to set the keyspace to the one you are using. Otherwise, you will always get the error: 'no keyspace has been specified'.



```
no keyspace has been specified
username text PRIMARY KEY,
first_name text,
last_name text,
category text
);
```

You can set the keyspace on the top of the script:



```
CREATE TABLE artists (
username text PRIMARY KEY,
first_name text,
last_name text,
category text
);
```

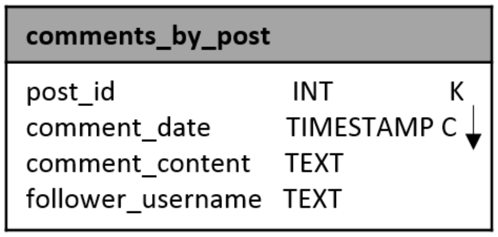
Alternatively, you can always include the keyspace in front of the table name, e.g., patron.artists.

Creating new tables and queries in Developer

Now, it's time for you to develop some own table, insert data, and run queries. I provide the solution on Blackboard. Note that this exercise is **not** the deliverable.

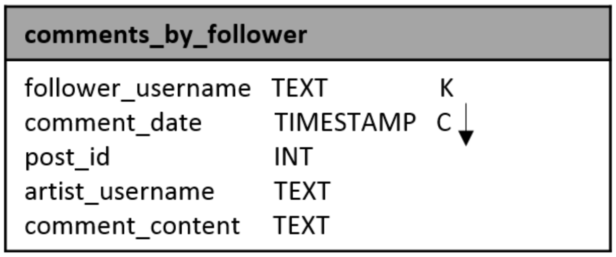
As an exercise, we develop tables that allow us to retrieve comments on posts and pledges for artists. Please note that we develop two tables for each in order to retrieve comments by post and by follower as well pledges by artist and by followers. Again, writes are cheap and in order to optimize data retrieval we store the same fact twice. So, when a follower adds a comment to a post we write the same information twice. When a follower pledges money to an artist, we will write the same information twice. This is still faster in Cassandra as compared to storing and retrieving the same data in a relational DBMS. By storing comments and pledges in two different ways, we can very efficiently retrieve comments for a post, comments for a follower, pledges for an artist, and pledges done by a follower.

Table 1: Lookup comments by post (K = partition key, C = clustering key, arrow down = DESC, arrow up = ASC)



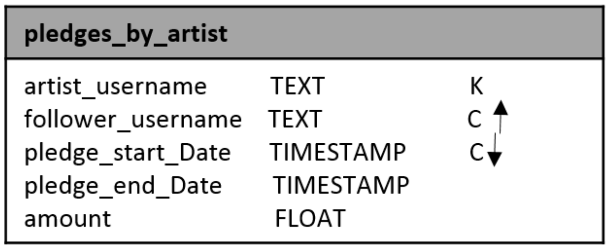
comments_by_post		
post_id	INT	K
comment_date	TIMESTAMP	C ↓
comment_content	TEXT	
follower_username	TEXT	

Table 2: Lookup comments by follower



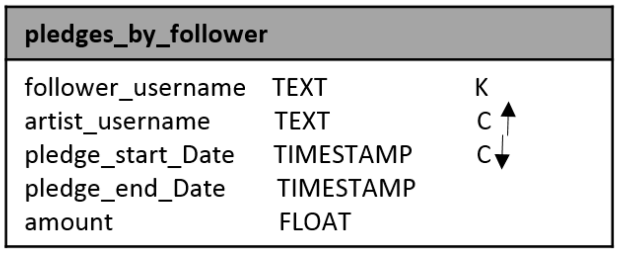
comments_by_follower		
follower_username	TEXT	K
comment_date	TIMESTAMP	C ↓
post_id	INT	
artist_username	TEXT	
comment_content	TEXT	

Table 3: Lookup pledges by artist



pledges_by_artist		
artist_username	TEXT	K
follower_username	TEXT	C ↑
pledge_start_Date	TIMESTAMP	C ↓
pledge_end_Date	TIMESTAMP	
amount	FLOAT	

Table 4: Lookup pledges by follower



pledges_by_follower		
follower_username	TEXT	K
artist_username	TEXT	C ↑
pledge_start_Date	TIMESTAMP	C ↓
pledge_end_Date	TIMESTAMP	
amount	FLOAT	

Go into Datastax Developer and select File -> New -> CQL script. It will open a new cql script. Save it with File -> Save as and enter file name *cassandra-tutorial-exercise-create-tables.cql*. Write the following queries:

- CREATE statement to create a keyspace called *tutorial* with replication simple strategy and replication factor 1.
- Switch to the keyspace you created with command USE.
- CREATE statement for first table. See figures above for names of table, columns, data types, primary key definition (partition and clustering key).
    - You can define the clustering order by appending WITH CLUSTERING ORDER (<clustering_key1> DESC, <clustering_key2> ASC, …) after the table definition. E.g.,
      ```
      CREATE TABLE example (
      ```

```
            col1 text,
            col2 text,
            col3 int,
            PRIMARY KEY ((col1), col2)
        ) WITH CLUSTERING ORDER (col2 DESC);
```

    - Timestamp data can be inserted in the format: '2016-04-12 21:49:00'
- CREATE statement for second table, third table, and fourth table.
- Several INSERT statements for each table. Remember that the values for the tables that store the same fact in different ways need to match (e.g., values for comment by post and comments by follower).
- Run, test, and debug your script. Yellow warnings on the left may be ignored (e.g., if you use upper letters in a column name, it will tell you that column names not put into quotes will not be case sensitive). Right warnings need to be taken care since you cannot run the query. The exception is after you created a table, ignore comments that tell you that you already created a table in the keyspace. The goal is that your script runs in one go without errors and produces the tables and inserts data. You may need to drop tables or the tutorial keyspace to make sure that it runs without problems.

Now, let's retrieve some data. Create a new CQL script and name it *Cassandra-tutorial-exercise-queries.cql*.

Write the queries that allow the following:

- The first statement should switch to the keyspace tutorial.
- Display all comments for one follower. (In order to run the SELECT query, you may have to select the keyspace tutorial from the drop-down box "in keyspace: " above the script.)
- Count the comments a follower has done.
- Display all comments for one follower and one artist. Did it work?
- Display all comments for one post.
- Display all comments for all posts after a specific data.
- If you get the comment " Bad Request: 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.", this is a good explanation of what is happening: http://www.datastax.com/dev/blog/allow-filtering-explained-2
- Display all comments for one post after a specific data.
- These are all queries that I wrote so far. Try out different queries. If you have questions about the queries or you find some interesting queries or messages, please share them on the learner-to-learner discussion board.
- Your scripts are saved under *user-home /.devcenter/DevCenter/.default*

Deliverable

Now, it is up to you to create your own tables and queries for the deliverable. We are continuing the movie and user rating example from Movielens.

Summary of tasks (explanation of tasks below):

- Create two .cql scripts, zip them together, and upload them to Blackboard.
    - First .cql script contains
        - CREATE KEYSPACE statement
        - USE statement
        - 3 CREATE TABLE statements
        - 15 INSERT statements
    - Second .cql script contains 8 queries accompanying with explanations in comments (//)
- Write your reflection (200 to 300 words) on the discussion boards on Blackboard.

Explanation of tasks:

1. Develop tables and insert data (replace *yourBlackBoardUserName* with your username on Blackboard) (50 points)
    - Create a new script called: *yourBlackBoardUserName* -create-tables.cql.
        - The first statement should create a keyspace called *yourBlackBoardUserName* with replication simple strategy and replication factor 1.
        - The second statement should switch to the keyspace you created (command USE).
        - Write the CREATE statements for the following four tables. K = Partition key, C = clustering key, arrow up = ASC (default), arrow down (DESC). You can add additional attributes if you want to try other data types (e.g., collections or UDTs).

        Table 1: Lookup table for movie

movies		
movie_ID	INT	K
title	TEXT	
year	INT	
category	SET <TEXT>	

        Table 2: Lookup ratings by movie

ratings_by_movie		
movie_ID	INT	K
user_ID	INT	C ↑
rating	FLOAT	
timestamp	INT	

        Table 3: Lookup ratings by user

ratings_by_movie			
user_ID	INT	K	
movie_ID	INT	C	↑
rating	FLOAT		
timestamp	INT		

Table 4: Create either a lookup table for the user or a table to lookup movies (ID and name) by category

- Write the statements to insert five rows in each table. The data you use should be consistent and you can use the data found in the user and movie tables in the Access database or Excel files found in the previous homework assignments. What I mean with consistent is that if you insert a rating for a movie then the movie_ID should be in the movies table (not necessarily the other way around, since it makes sense to have a movie without ratings.) (15 insert statement in total)

2. Write queries for your tables (50 points)
   - Create a new script called: *yourBlackBoardUserName* -queries.cql.
   - The first statement should switch to the keyspace you create in the first script (*yourBlackBoardUserName*;).
   - Then write 8 different queries for the four tables. You should use the keywords WHERE, COUNT, > or < (range query), and AND each at least once in a query.
   - For each query write an explanation of the query in comments (i.e., what the query does). A comment in cql starts with two forward-slashes `// Comment`
3. Write your reflection on the discussion board in BB (10 points)

   200 to 300 words about your experience creating and querying the tables in the Thread "Please post your reflection for homework 3 here" on the discussion board "Reflections". Do not open your own thread, but append to this one.