

```
%sh
STATUS="$(service cassandra status)"

if [[ $STATUS == *"is running"* ]]; then
  echo "Cassandra is running"
else
  echo " Cassandra not running .... Starting"
  service cassandra restart > /dev/null 2>&1 &
  echo " Started"
fi
```

READY

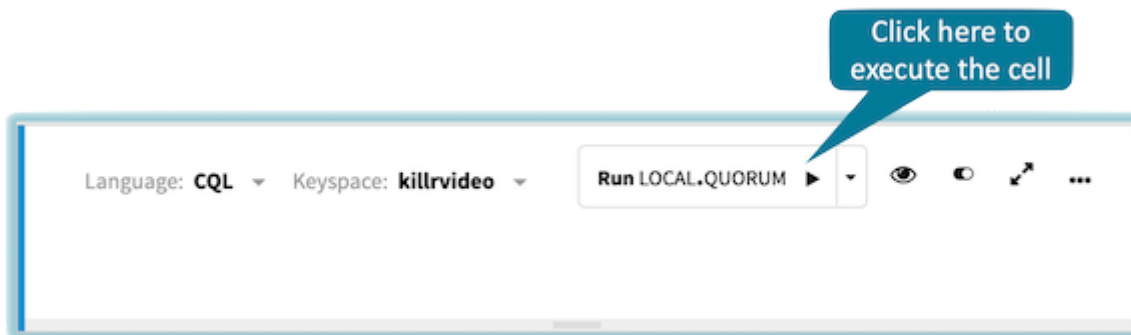
READY

## Set Up the Notebook

In this section, you will do the following things:

- Execute a CQL script to initialize the KillrVideo database for this notebook

**Step 1: Execute the following cell to initialize this notebook. Hover over the right-hand corner of the cell and click the *Run* button.**



**Note:** You don't see the CQL script because the code editor is hidden, but you can still run the cell.

READY

READY

## Let's Look at the `video_recommendations` Table and Data

In this section, you will do the following things:

## 4 \_Data\_Modeling

Familiarize yourself with the `video_recommendations` table using the `DESCRIBE` command

- Look at the contents of the table

- Do a simple query on the table

**Step 1:** In the following cell, describe the `video_recommendations` table to familiarize yourself with its definition.

*Need a hint? Click here.*

You can use the `DESCRIBE TABLE` command.

Remember, the keyspace name is `killrvideo` and the table name is `video_recommendations`.

*Want to see the command? Click here.*

```
DESCRIBE TABLE killrvideo.video_recommendations;
```

READY

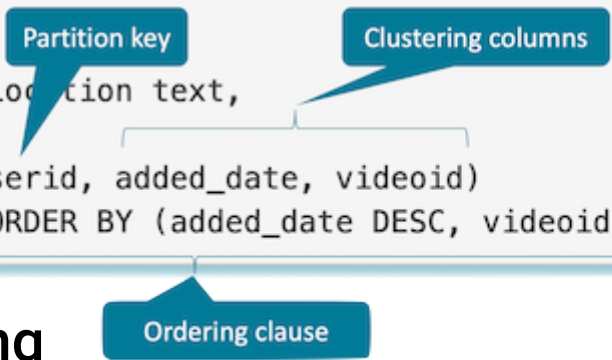
```
// Write the command to describe the user_crednetials table in this cell  
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

**Step 2:** Review details of this table definition.

READY

- Note that the primary key consists of a single partition key ( `userid` ) and two clustering columns ( `added_date` and `videoid` )
- Notice also the `WITH CLUSTERING ORDER BY` clause, which defines the storage order of the data within partitions

```
CREATE TABLE killrvideo.video_recommendations (  
  userid uuid,  
  added_date timestamp,  
  videoid uuid,  
  authorid uuid,  
  name text,  
  preview_image_location text,  
  rating float,  
  PRIMARY KEY (userid, added_date, videoid)  
) WITH CLUSTERING ORDER BY (added_date DESC, videoid ASC)
```



## 4 \_Data\_Modeling

**Step 3:** To see the `video_recommendations` data, execute the following cell.

**Note:** You don't see the CQL script because the code editor is hidden, but you can still run the cell.

## Points of interest in the data above

READY

- We have contrived some `uuid` values for these rows. You would use real `uuid`s in production, but these are constant, which makes the exercises consistent
- You see that the `authorid` and `preview_image_location` columns contain no data, which is OK because Cassandra supports sparse tables

**Step 4:** In the next cell, search for the rows where `userid = 11111111-1111-1111-1111-111111111111`.

*Need a hint? Click here.*

You can use a `SELECT` command to retrieve all columns.

Remember, the keyspace name is `killrvideo` and the table name is `video_recommendations`.

The `WHERE` clause will use

`userid = 11111111-1111-1111-1111-111111111111`.

*Want the code? Click here.*

```
SELECT * from killrvideo.video_recommendations WHERE userid = 11111111-1111-1111-1111
```

READY

```
// Write the command to retrieve from video_recommendations where userid is 11111111-1111-1111-1
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

We see both rows with the specified `userid`. That seemed to work as expected - no big surprises!

READY

**Note:**

## 4 \_Data\_Modeling

It may not be obvious from this simple example, but notice that Cassandra orders the results by the clustering columns as specified in the `WITH CLUSTERING ORDER` clause used when creating the table. This default ordering is one reason for clustering columns and using this

clause.

**Thought question:** If you only wanted a single row, how would you change the query?

READY

Same Table, Different Query...

In this section, you will do the following things:

- Perform a query without using the partition key and see what happens

Let's say we want to find all the top reviews. If you are used to the relational world, the obvious answer is to query based on `rating`.

**Step 1:** In the following cell, formulate and execute a query based on `rating`.

*Need a hint? Click here.*

You can use a `SELECT` command to retrieve all columns.

Remember, the keyspace name is `killrvideo` and the table name is `video_recommendations`.

The `WHERE` clause will use `rating = 5.0`.

But here's the real hint: *The query isn't going to work. try it out and see.*

*Want the code? Click here.*

```
SELECT * from killrvideo.video_recommendations WHERE rating = 5.0;
```

READY

```
// Write a query to retrieve all rows with the rating = 5.0
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

**What just happened?**

READY

## 4 \_Data\_Modeling



In the previous cell, we tried to do a query without using a partition key. This doesn't work in Cassandra because it would require a full-table scan!

The big secret about Cassandra: *Cassandra is just a big distributed hash-table!*

- Which means it's fast but, you must use the full partition key in *all* queries because that is what Cassandra hashes to find the partition
- Partitions are sets of rows and, within each partition, Cassandra orders the rows according to the clustering columns
- Fast queries retrieve a single (relatively small) partition

OK, let's come clean here. Cassandra is really much more than just a distributed hashtable, but as a logical model to help you think about Cassandra data modeling, the hashtable idea works reasonably well.

#### Note:

Cassandra hashes the full partition key to retrieve a partition. This has implied consequences:

Retrieving a single partition is very fast

Retrieving multiple partitions will be slower

Partition keys have no inherent order, so you cannot perform "greater than" or "less than" types of operations on partition keys

Since Cassandra orders the rows *within a partition* based on clustering columns, you *can* perform "greater than" or "less than"

operations on clustering columns *after* you specify the partition key

## 4 \_Data\_Modeling

The bottom line: If you want your Cassandra queries to be fast:

- Create tables where the *full* partition key is how you will query the table
- The clustering columns are how you want to order the results
- Do *not* require more than one partition key per query

**Note:**

In addition to ordering query results, you can also use clustering columns to create a unique primary key for each row (if necessary).

READY

What's Up, Sert?

In this section, you will do the following things:

- Try to insert a previously inserted row and see what happens
- Understand why Cassandra handles inserts this way

Let's look at an existing row in the `users` table (and practice our CQL skills in the process). As a reminder, this table has the following definition.

```
CREATE TABLE users (  
    userid      uuid,  
    firstname   text,  
    lastname    text,  
    email       text,  
    created_date timestamp,  
    PRIMARY KEY (userid)  
);
```

**Step 1: In the following cell, write and execute a query to retrieve the row associated with `userid = 11111111-1111-1111-1111-111111111111`.**

*Need a hint? Click here.*

You can use a `SELECT` statement that retrieves all columns ( `*` ).  
Remember, the keyspace name is `killrvideo` and the table name is `users` .

The `WHERE` clause should specify

`userid = 11111111-1111-1111-1111-111111111111` .

## 4 \_Data\_Modeling

*Want the code? Click here.*

```
SELECT * FROM killrvideo.users WHERE userid = 11111111-1111-1111-1111-111111111111;
```

READY

```
// Write a query to retrieve the row associated with userid =
```

```
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

We see that this row exists and that the value of the `firstname` column is `Jeffrey`. READY

**Step 2: In the following cell, write and execute an `INSERT` statement to insert a row with a different `firstname` as follows.**

- `userid` is `11111111-1111-1111-1111-111111111111`
- `created_date` as `NOW ( toTimestamp(now()) )`
- `email` as `jc@datastax.com`
- `firstname` as `Jeff`
- `lastname` as `Carpenter`

If you are coming from a relational database background, you anticipate that this command will fail because of the existing row...

*Need a hint? Click here.*

You can use an `INSERT` command.

Remember, the keyspace name is `killrvideo` and the table name is `users`.

The columns are `userid`, `created_date`, `email`, `firstname` and `lastname`.

The associated `VALUES` are `11111111-1111-1111-1111-111111111111`, `toTimestamp(now())`, `jc@datastax.com`, `Jeff` and `Carpenter`.

*Want the code? Click here.*

```
INSERT INTO killrvideo.users (userid, created_date, email, firstname, lastname)
VALUES(11111111-1111-1111-1111-111111111111, toTimestamp(now()), 'jc@datastax.com',
```

## 4 \_Data\_Modeling

READY

```
// Write an INSERT statement to insert a row with the firstname Jeff  
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

READY



No errors! Why?

It appears that the `INSERT` statement worked! How can that be?

Let's query the row again and look at the column values...

Step 3: In the following cell, perform the same query you executed in Step 1.

*Need a hint? Click here.*

Seriously? You just wrote this query two steps ago! :)

Go back to Step 1 and copy that query and paste it in the following cell.

*Want the code? Click here.*

```
SELECT * FROM killrvideo.users WHERE userid = 11111111-1111-1111-1111-111111111111;
```

READY

```
// Write a query to retrieve the row associated with userid =  
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

## 4 \_Data\_Modeling

READY

Step 4: Compare the results from the Step 1 query to the results from the Step 2 query.

READY



Careful analysis reveals the second `INSERT` command updated the `firstname` column value for this row!

### Useful discussion (worth the read):

Yeah, this is a bit of a shock to those coming from the relational world.

But here's the reason: Cassandra does *not* do a read before writing because that would cripple Cassandra's performance.

Instead, Cassandra assumes that if you are inserting a row, the row doesn't exist and Cassandra proceeds to quickly write the row.

A similar situation occurs with updates.

Cassandra does not read anything before writing an update.

So, if the record we are updating does not exist, Cassandra merely creates it.

We refer to each of these situations as an *upsert*.

Upserts are not earth-shattering, but you need to be aware of them.

If you absolutely *must* perform a read before writing due to a race condition, Cassandra provides *lightweight transactions*.

Lightweight transactions force Cassandra to read before writing, but they have serious performance implications and should be avoided if possible.

We will cover lightweight transactions in a different notebook.

Once again, you can find more in the free DataStax Academy Data Modeling (<https://academy.datastax.com/resources/ds220>) course.

READY

## A Cassandra Data Modeling Recipe

In this section, you will do the following things:

- Learn a process for Cassandra Data Modeling
- Practice each of the steps of this process

### Introduction

When modeling in the relational world, the traditional approach is to create E/R diagrams, identify tables and then normalize until the cows come home.

This approach won't work for Cassandra because you can end up with tables that won't support your queries, or will perform poorly.

## 4 \_Data\_Modeling

So instead, Cassandra data modeling starts with the queries in mind. Here are the steps to the process.

1. Enumerate all use-cases and their interdependencies
2. Use the use cases to identify all queries the app will perform
3. Use the queries to drive the table definitions

Let's see how this works...

Since the use-cases drive the queries, it is useful first to identify the use-cases. Also, since the app may obtain information during one use-case that may be used in another use-case, it is also important to note the dependency of the use-cases.

Let's consider the use-cases associated with the KillrVideo app - specifically with respect to users of the app. This works best if you can collaborate.

**Step 1:** Find a buddy (or two) around you to collaborate with.



**Step 2:** Discuss the user-related use-cases associated with the KillrVideo app.

READY

- Start by enumerating the user-centric use-cases (e.g., What's the first thing a user must do? What's the next thing?)
- Order the use-cases to identify any dependent data (Does performing a use-case require you first to perform another uses-case? If so, note the order.)

There is nothing to execute in this step - just note the use-cases and their dependencies.

*Want a hint? Click here.*

If we think of the natural flow for a user of the app, probably the first thing a user will do is to register with the app.

So the first use-case should be *Register-User*.

## 4 \_Data\_Modeling

Another thing a user might want to do would be to see some sort of state relative to the app.

For example, they may want to review their profile, viewing history, upload history, etc.

We'll refer to this use-case as *Get-User-Info*.

Users will need to authenticate with the app so that the app can interact with the user according to the user's preferences and permissions.

We'll refer to this use-case as *Authenticate-User*.

If we limit our consideration to these three use-cases, we see that Register-User must occur before Authenticate-User.

We also see that Get-User-Info depends on Register-User and Authenticate-User.

*Want to compare your answers? [Click here](#).*

User use-cases:

- Register-User
- Authenticate-User (depends on Register-User)
- Get-User-Info (depends on Register-User and/or Authenticate-User)

**Armed with the use-cases, you are ready to identify the queries.**

READY

**Step 3: Use the use-cases to identify the queries as follows:**

- For each use-case, determine the database access you will need to support the use-case
- For each access, identify the inputs and the outputs
- For queries, the inputs become keys and the outputs become data columns

**There is nothing to execute in this step - just note the database accesses, their associated use-cases, any key data and the returned data.**

*Want a hint? [Click here](#).*

Given the use-cases, we can think through the process of each and identify any database access associated with the use-case.

When we determine that a use-case requires a query, we must think about the query in terms of what data the query must return, and how we can identify that data.

## 4 \_Data\_Modeling

The identifying data becomes the key and the returned data becomes, uh, the data.

Take time to enumerate the database accesses.

For each access, you may want to keep track of its associated use-case.

For queries, you want to note the key data and the returned data.

Also, if queries return multiple rows, note any requirements for row ordering.

*Want to compare your answers? [Click here](#).*

DB Access: CreateUser (use-case: Register-User)

- Input: User's email address, user's password, user ID
- Output: None

DB Access: GetPasswordAndIdByEmail (use-case: Authenticate-User)

- Input: User's email address
- Output: User's password (hash), user ID

DB Access: GetUserInfoById (use-case: Get-User-Info)

- Input: User's ID
- Output: User's name (first and last), email, created date

Let's use the queries to define the schema.

READY

First Big Ah-Ha:

Wait! Does this mean I create a new table for each query?

- Generally, yes!
- Sometimes you can use a table to satisfy more than one query, but not always

*[Click here for further explanation](#).*

In the relational world, you might create a single table and query it in different ways.

This seems like a good approach because data is normalized, which means we only need to store it once.

The problem is that the relational world may also require joins and projections which do not scale - your app will not keep up.

## 4 \_Data\_Modeling

Instead, in the Cassandra world, we create tables to support specific queries.

This means that the results for a query are precomputed and are sitting in a partition, ready to be retrieved quickly.

This approach is fast and scales.

Second Big Ah-Ha:

Won't this cause denormalization?!!!!!!

- Yes!
- But it's alright

*Click here for further explanation.*

Denormalization is a way of life in the world of Cassandra.

We realize your college professor taught you that denormalization is evil, etc., but relax!

Denormalization is not the evil boogie-man.

Many large companies denormalize massive amounts of data - it's what allows them to scale and be successful!

Yes, your app will need to keep track of what data is where and be sure to update the necessary tables, but it's not really that difficult once you get used to it.

**Step 4:** For each database access, determine the table design.

READY

- Identify the columns and their data types
- Determine the partition key, which are the inputs in the previous step
- If the output has multiple rows and requires ordering, identify the clustering columns accordingly
- Make sure each row has a unique primary key

*Want a hint? Click here.*

Define the tables you will need based on the database access you determined in the previous step.

For queries, the input usually becomes the partition key.

The output from the queries become the other columns of the table.

If the query retrieves multiple columns and if you need to have the resulting rows ordered, make the ordering columns clustering columns.

## 4 \_Data\_Modeling

If the query returns multiple columns, then you may also need to add some clustering columns to ensure the primary key is unique. Remember, if the primary key is not unique, then when you go to insert a second row, you may cause an upsert that updates an existing row.

Once you think you have considered these concerns, try creating the CQL to create the table.

*Want to compare answers? Click [here](#).*

Notice that the CreateUser access will require a table with:

- An `email` column as the partition key
- `password` and `userid` columns as payload data
- We'll name this table `user_credentials`

The GetPasswordAndIdByEmail query will also use the `user_credentials` table.

The GetUserInfoById query will require a table with:

- `userid` as the partition key
- Various payload data such as `firstname`, `lastname`, `email` and `created_date`
- This defines a `users` table - which is the same table we created in the CQL notebook, so we can use that

Now, we are ready to write the actual CQL! Here's the `user_credentials` table design.

In production, we would never store the password as plain text. However for this exercise, to keep it simple we will use plain text.

Column name	Data Type	Notes
<code>email</code>	TEXT	Partition key
<code>password</code>	TEXT	Payload data
<code>userid</code>	UUID	Payload data

**Step 5:** In the next cell, create the `user_credentials` table as discussed and shown above. READY

*Need a hint? Click [here](#).*

## 4 \_Data\_Modeling

```
CREATE TABLE killrvideo.user_credentials (
  // Fill in the next line with the first column name (hint: rhymes with "flea-ale")
  ,
  // Fill in the next line with the second column name (hint: rhymes with "bass-bird")
  ,
  // Fill in the next line with the third column name (hint: rhymes with "loser-by-me")
  ,
  PRIMARY KEY(
    // Fill in the next line with the partition key column name
  )
);
```

Want to see the solution? [Click here.](#)

```
CREATE TABLE killrvideo.user_credentials (
  email TEXT,
  password TEXT,
  userid UUID,
  PRIMARY KEY(email)
);
```

READY

```
// Write the CQL to create the user_credentials table here:

// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

Let's look at the users in the `users` table.

READY

**Step 6:** Execute the following cell and check out the results.

**Note:** You don't see the CQL script because the code editor is hidden, but you can still run the cell.

READY

The users in the `users` table also need to be in the `user_credentials` table - that's how

## 4 \_Data\_Modeling

**Step 7:** For each row in the `users` table, create a row in the `user_credentials` table.

- The two tables contain denormalized data, so make sure the denormalized columns are consistent
- Do *NOT* use the `uuid()` method for the `userid` column in the `user_credentials` because `uuid()` would generate a different ID and the `userid`s in the two tables need to match

Here is a list of user data for reference:

First Name	Last Name	Email	Password	userid
Jeff	Carpenter	jc@datastax.com	J3ffL0v3\$C@ss@ndr@	11111111-1111-1111-1111-111111111111
Eric	Zietlow	ez@datastax.com	C@ss@ndr@R0ck\$	22222222-2222-2222-2222-222222222222
Cedric	Lunven	cl@datastax.com	Fr@nc3L0v3\$C@ss@ndr@	33333333-3333-3333-3333-333333333333
David	Gilardi	dg@datastax.com	H@t\$0ff2C@ss@ndr@	44444444-4444-4444-4444-444444444444
Cristina	Veale	cv@datastax.com	3@\$tC0@\$tC@ss@ndr@	55555555-5555-5555-5555-555555555555
Aleks	Volochnev	av@datastax.com	C@ss@ndr@43v3r	66666666-6666-6666-6666-666666666666
Ardon	Hall	ah@datastax.com	C@ss@ndr@3v3rywh3r3	77777777-7777-7777-7777-777777777777

- ▶ \*Want a hint? Click here.\*
- ▶ \*Want the answer? Click here.\*

```
// Write the CQL to insert the seven (advocates) users into user_credentials here:      READY
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the Run but
```

**Step 8: Execute the next *two* cells to confirm the contents of the `user_credentials` and `users` tables.**

**Note: You don't see the CQL script because the code editor is hidden, but you can still run the cell.**

READY

**4 \_Data\_Modeling**

```
// See the contents of the entire users table      READY
// Note: this query requires a full table scan - we would never run this type of query on a prod
// Execute this cell by clicking on the cell and pressing SHIFT+ENTER or clicking on the CLONE
SELECT * FROM killrvideo.users;
```



**Step 9: Review your work.**

READY

- Compare the `email` and `userid` column values of the two tables to make sure the values are consistent and correct

Congratulations!!!!

**If you have made it to the end of this notebook successfully, you are armed with the essential Cassandra data modeling skills!**

READY

Bonus Challenge: Model the KillrVideo comments domain

If you got done early and want something to do while you wait for others, here's a bonus challenge

In this section, you will do the following things:

- Create a data model for the "comments" domain within KillrVideo

Since this is a bonus exercise, we won't hold your hand. If you need help, refer back to the previous exercise as a pattern.

**Here's the pitch:**

#In this bonus challenge, we want to try and work through a very simple data model. Let's limit ourselves to the section of KillrVideo that deals with comments about videos. In this domain, users make comments about various videos. Of course, other users like to peruse the comments about a video for sundry reasons. Also, any user may want to review the comments they have made.

**This is a simple domain. Let's start by identifying the data items in this domain.**

**Step 1: List the names and meanings of the data items for the video comments domain.**

*Want to compare your answers to ours? [Click here.](#)*

**videoid** – The ID of the video that is the subject of the comment

**userid** – The ID of the user who made the comment

**comment** – The text of the comment

## 4 \_Data\_Modeling

**Step 2: List the use-cases associated with this domain.**

READY

*Want to compare your answers to ours? [Click here](#).*

Your answers may vary based on your understanding of the problem and the domain. The following represents our perspective:

**Create a comment** - A user creates a comment on a video

**See video comments** - A user wants to see all comments associate with a video

**Review my comments** - A user wants to review the comments he/she has created

**Review other user's comments** - A user wants to see all comments created by some other user

**Step 3: Create database access pattern(s), including identifying the inputs and outputs, necessary to support all the use-cases.** READY

*Want to compare your answers to ours? [Click here](#).*

**CreateComment: (use-case: Create a comment)**

- **Input:** videoid, userid, comment
- **Output:** none

**CommentsByVideo: (use-case: See video comments)**

- **Input:** videoid
- **Output:** comments, userids

**CommentsByUser: (use-case: Review my comments and Review other user's comments)**

- **Input:** userid
- **Output:** videoids, comments

**Step 4: For each database access pattern that has an output, define the tables.** READY

- Name the table
- Identify the column names and their associated data types

**4 \_Data\_Modeling** Determining the partition key (and its data type) as defined by the access pattern input

- Determine any ordering for the output (probably most recent comments first)

- Make sure the primary key is unique for each row

*Want to compare your answers to ours? [Click here.](#)*

**CommentsByVideo: // Note that the commentid makes the row unique and provides necessary ordering**

- **Table name:** comments\_by\_video
- **Column:** comment TEXT
- **Column:** userid UUID
- **Column:** videoid UUID
- **Column:** commentid TIMEUUID
- **Primary key:** (videoid, commentid)

**CommentsByUser: // Note that the commentid makes the row unique and provides necessary ordering**

- **Table name:** comments\_by\_user
- **Column:** comment TEXT
- **Column:** videoid UUID
- **Column:** userid UUID
- **Column:** commentid TIMEUUID
- **Primary key:** (userid, commentid)

**Step 5: For each table definition, write the CQL to create the table.**

FINISHED

*Want to compare your answers to ours? [Click here.](#)*

## 4 \_Data\_Modeling

```
CREATE TABLE killrvideo.comments_by_video (  
  videoid UUID,  
  commentid TIMEUUID,  
  comment TEXT,  
  userid UUID,  
  PRIMARY KEY(videoid, commentid));
```

```
CREATE TABLE killrvideo.comments_by_user (  
  userid UUID,  
  commentid TIMEUUID,  
  comment TEXT,  
  videoid UUID,  
  PRIMARY KEY(userid, commentid));
```

Took 2 sec. Last updated by anonymous at July 01 2020, 5:14:34 PM.

READY

**Step 6: For each query access pattern, write the CQL for the query.**

READY

*Want to compare your answers to ours? [Click here.](#)*

```
SELECT * FROM killrvideo.comments_by_video WHERE videoid = SomeVideoID;  
SELECT * FROM killrvideo.comments_by_user WHERE userid = SomeUserID;
```

```
// drop tables if they exist already.
```

READY

```
DROP TABLE killrvideo.comments_by_video;  
DROP TABLE killrvideo.comments_by_user;
```

READY

READY