Exercise 1.1: Setting Up Lab Environment

In this exercise, you will:

- Setup the virtual machine for this course
- Familiarize yourself with cqlsh

Steps

Connect to the cloud instance

1. If you are taking this course in a class environment, your instructor will assign you an IP address of a cloud instance that you will SSH into.
2. SSH to the cloud instance using the IP address and key provided.

   ```
   ssh -i <classkey> ubuntu@<ipaddress>
   ```

Run cqlsh

3. Start the Cassandra tool'cqlsh'. You can run 'cqlsh' or any Cassandra / DSE tool from any directory because they are in your '$PATH'.

   ```
   cqlsh
   ```

IMPORTANT: If 'cqlsh' is not able to connect to DataStax Enterprise, use the command 'service dse status' to see if DSE is running. If for some reason DSE is not running, you can start it with 'service dse start'.

In 'cqlsh', note the prompt header changed to `cqlsh>` indicating further commands are for 'cqlsh' instead of the Linux shell.

4. Execute the following query:

   ```
   SELECT * FROM system_schema.keyspaces;
   ```

'cqlsh' displays the results immediately. Notice the built-in keyspaces. Your cqlsh environment is working correctly.

5. Type 'exit' to quit 'cqlsh'.

Exercise 1.2: Basic CQL Fundamentals

- Create a Keyspace for KillrVideo
- Create a table to store video metadata
- Load the data for the video table from a CSV file

Background

Welcome to the KillrVideo company! KillrVideo hired you to build the latest and greatest video sharing application on the Internet. Your task is to ramp up on the domain and become acquainted with Cassandra. To start, you decided to look into creating a table schema and to load some video data.

The video metadata is made up of:

Column Name	Data Type
video_id	timeuuid
added_date	timestamp
description	text
title	text
user_id	uuid

Steps

1. If necessary, SSH to cloud environment provided by your instructor.
2. Open '/home/ubuntu/labwork/cql/videos.csv' with a text editor and review the contents of the file.

IMPORTANT: Notice the order of the columns matches the order shown above.

3. Navigate to the '/home/ubuntu/labwork/cql' directory.

   ```
   cd /home/ubuntu/labwork/cql
   ```

4. At the prompt, start 'cqlsh'.
5. In 'cqlsh', create a keyspace called 'killrvideo' and switch to that keyspace. Use 'SimpleStrategy' for the replication class with a replication factor of one. Remember the 'use' command switches keyspaces.

NOTE: You can press the tab key within the CREATE KEYSPACE command to have 'cqlsh' autocomplete the replication parameters.

6. Create a single table called 'videos' with the same structure as shown in table above. 'video_id' is the primary key.

7. Load the newly created table with the 'videos.csv' file using the 'COPY' command.

   COPY videos (video_id,added_date,description,title,user_id) FROM 'videos.csv' WITH HEADER=true;

8. Use SELECT to verify the data loaded correctly. Include LIMIT to retrieve only the first 10 rows.

9. Use SELECT to COUNT(*) the number of imported rows. It should match the number of rows COPY reported as imported.

10. Use SELECT to find a row where the video_id = 6c4cffb9-0dc4-1d59-af24-c960b5fc3652.

Next we will explore some other CQL commands that will come in handy, like TRUNCATE in a later exercise, we will show you how to add/remove (non-primary key) columns.

11. Let's remove the data from our table using TRUNCATE.

    ```
    truncate videos;
    ```

12. Exit 'cqlsh'.

Exercise 2.1: Working with Partitions

In this exercise, you will:

- Create a new table that allows querying videos by title and year using a composite partition key

Background

Your peers need to query videos based on 'title' and 'added_year' The new columns for this table are:

Column Name	Data Type
title	text
added_year	int
added_date	timestamp
description	text
user_id	uuid
video_id	uuid

Steps

1. Use a text editor to view the '/home/ubuntu/labwork/partitions/videos_by_title_year.csv' file.
2. At prompt, navigate to '/home/ubuntu/labwork/partitions/'.
3. Start 'cqlsh'
4. Create a second table in the 'killrvideo' keyspace called 'videos_by_title_year' with the structure shown in above table. Be sure users can query this table on both 'title' and 'added_year' by combining them into the partition key.
5. Load the data from the 'videos_by_title_year.csv' file using the `COPY` command.
6. COPY videos_by_title_year (title, added_year, added_date, description, user_id, video_id) **FROM** 'videos_by_title_year.csv' WITH HEADER=true;
7. Try running queries on the 'videos_by_title_year' table to query on a specific 'title' and 'added_year'.

Example queries:

title	added_year
Introduction to Apache Cassandra	2014
Grumpy Cat: Slow Motion	2015
Grumpy Cat: Slow Motion	2015
AzureDev	2015

8. What error does Cassandra return when you try to query on just title or just year? Why?

9. Exit cqlsh.

Exercise 2.2: Clustering Columns

In this exercise, you will:

- Create a 'videos_by_tag_year' table that allows range scans and ordering by year

Background

There have been some interesting wrinkles in your quest to understand how Cassandra and CQL work. Although you have been able to complete your tasks to the letter, your team cannot query based on tag and year. Fortunately, your new understanding of clustering columns will help to improve your design. You decide to build a table that allows querying by 'tag' and 'added_year'.

The columns are as follows:

Column Name	Data Type
tag	text
added_year	int
video_id	uuid
added_date	timestamp
title	text
user_id	uuid

Steps

1. At the prompt, navigate to '/home/ubuntu/labwork/clustering' directory. Launch 'cqlsh' and switch to the 'killrvideo' keyspace.

In order to demonstrate a little bit more about clustering columns, we are first going to show you upserts.

2. To become an upsert ninja, create the following (bad) table with the (crummy) primary key:

```
CREATE TABLE bad_videos_by_tag_year (
  tag text,
  added_year int,
  added_date timestamp,
  title text,
  description text,
  user_id uuid,
  video_id timeuuid,
  PRIMARY KEY ((video_id))
);
```

3. As an aside, use DESCRIBE TABLE to view the structure of your 'bad_videos_by_tag_year' table.

NOTE: Notice the column order differs from the CREATE TABLE statement. Cassandra orders columns by partition key, clustering columns (shown later), and then alphabetical order of the remaining columns.

4. Execute the following COPY command to import `videos_by_tag_year.csv` file.

```
COPY bad_videos_by_tag_year (tag, added_year, video_id,
added_date, description, title, user_id) FROM
'videos_by_tag_year.csv' WITH HEADER=true;
```

NOTE: We must explicitly list the column names because this table schema no longer matches the CSV structure.

NOTE: Notice the number of imported rows.

5. Now COUNT() the number of rows in the `bad_videos_by_tag_year`.

```
SELECT COUNT(*)
FROM bad_videos_by_tag_year;
```

Notice the number of rows in the 'bad_videos_by_tag_year' does not match the number of rows imported from 'videos_by_tag_year.csv'. Since 'videos_by_tag_year.csv' duplicates 'video_id' for each unique 'tag' and 'year' per video, Cassandra upserted several records during the COPY. `video_id` is not a proper partition key for this scenario.

6. Drop your nasty table.

```
DROP TABLE bad_videos_by_tag_year;
```

Your mission is to restructure your table and allow users to query on 'tag' and possible 'added_year' ranges while avoiding upserts on import. You must also return your results in descending order of year.

Steps

7. Create a table with the columns above to facilitate querying for videos by tag within a given year range returning the results in descending order by year.

8. We wrote most of the CREATE TABLE for you. Fill in the PRIMARY KEY and CLUSTERING ORDER BY.

```
CREATE TABLE videos_by_tag_year (
  tag text,
  added_year int,
  video_id timeuuid,
  added_date timestamp,
  description text,
  title text,
  user_id uuid,
  PRIMARY KEY ( )
) WITH CLUSTERING ORDER BY ( );
```

9. Load the data from the 'videos_by_tag_year.csv' file in the provided 'exercise=4' directory using the COPY command.

   ```
   COPY videos_by_tag_year FROM 'videos_by_tag_year.csv' WITH
   HEADER=true;
   ```

10. Check the number of rows in the 'videos_by_tag_year' table.

NOTE: The number of rows should match the number of rows imported by the COPY command. If not, you had upserts again and will need to adjust your PRIMARY KEY. Ask your instructor for help if necessary.

11. Try running queries on the 'videos_by_tag_year' table to query on a specific tag and added year.

Example queries:

tag	added_year
trailer	2015
cql	2014
spark	2014

12. Try querying for all videos with tag "cql" added before the year 2015. Notice you can do range queries on clustering columns.

13. Try querying for all videos added before 2015. The query will fail. What error message does cqlsh report? Why did the query fail whereas the previous query worked?

14. Exit cqlsh.

Exercise 2.3: Denormalizing

In this exercise, you will:

- Create tables to support querying for videos by actor or genre

Background

With all of the success you've been having on the video sharing development team, you have been promoted and assigned to work on a high-priority project to incorporate movie content into the KillrVideo application.

Your new team is normalizing their video and actor metadata into separate tables and currently are stuck figuring out how to join tables in Cassandra. Having been around the Cassandra block a few times, you know that JOINs are expensive and not supported. It is up to you to show your team the optimal way of performing these queries.

The video metadata is similar to what was in the video sharing domain:

Column Name	Data Type
video_id	timeuuid
added_date	timestamp
description	text
encoding	video_encoding
tags	set<text>
title	text
user_id	uuid

There is also the additional following metadata:

Column Name	Data Type
actor	text
character	text
genre	text

With this metadata, the data model must support the following queries:

- Q1: Retrieve videos an actor has appeared in (newest first).
- Q2: Retrieve videos within a particular genre (newest first).

Creating a new videos_by_actor table

1. Navigate to /labwork/denormalization.

2. In 'cqlsh', create a new table called 'videos_by_actor' which will support query Q1.

Before we do that though, look a little more closely at the table above. Our 'encoding' column is actually something called a User Defined Type (or UDT for short). Fear not! We will be talking about these in the next exercise. For now, copy and paste this code to create the UDT so that our create table works correctly.

3. Creating the 'encoding' UDT.

```
CREATE TYPE IF NOT EXISTS video_encoding (
   encoding TEXT,
   height INT,
   width INT,
   bit_rates SET<TEXT>
);
```

4. We provided most of the CREATE TABLE for you except the PRIMARY KEY.

```
CREATE TABLE videos_by_actor (
  actor text,
  added_date timestamp,
  video_id timeuuid,
  character_name text,
  description text,
  encoding frozen<video_encoding>,
  tags set<text>,
  title text,
  user_id uuid,
  PRIMARY KEY ( )
) WITH CLUSTERING ORDER BY ( );
```

5. Load 'videos_by_actor.csv' into the 'videos_by_actor' table using the COPY command.

```
COPY videos_by_actor
(actor,added_date,video_id,character_name,description,encoding,ta
gs,title,user_id) FROM 'videos_by_actor.csv' WITH HEADER = true;
```

6. Run a query to retrieve the video information for a particular actor (Tom Hanks, Denzel Washington, or see if your favorite actor is in there).

7. Try SELECTing just the actor and the added_date columns. Notice the order of added_dates.

Create a videos_by_genre table

8. In 'cqlsh', create a new table called 'videos_by_genre' which will support query Q2. We provided most of the CREATE TABLE for you except the PRIMARY KEY.

```
CREATE TABLE videos_by_genre (
  genre text,
  added_date timestamp,
  video_id timeuuid,
  description text,
  encoding frozen<video_encoding>,
  tags set<text>,
  title text,
  user_id uuid,
  PRIMARY KEY ( )
) WITH CLUSTERING ORDER BY ( );
```

9. Load 'videos_by_genre.csv' into the 'videos_by_genre' table using the COPY command.

```
COPY videos_by_genre
(genre,added_date,video_id,description,encoding,tags,title,
user_id) FROM 'videos_by_genre.csv' WITH HEADER = true;
```

10. Run a query to retrieve the video information for a particular genre (Future noir, Time travel).

```
SELECT * FROM videos_by_genre WHERE genre = 'Musical' LIMIT 10;
```

11. Exit cqlsh.

Exercise 3.1: User Defined Types (UDTs)

In this exercise, you will:

- Create a user defined type
- Alter an existing table and add additional columns

Background

After reviewing your design of the tables that support tag and year queries, your manager happened to remember that there isn't a 'tags' column in the original table that stores video metadata. You have now been asked to include that in the videos table schema and also to add another column to store video encoding information.

The revised 'videos' table schema:

Column Name	Data Type
video_id	timeuuid
added_date	timestamp
description	text
encoding	video_encoding
tags	set<text>
title	text
user_id	uuid

The encoding data structure:

field Name	Data Type
Bit_rates	set<text>
encoding	text
height	int
width	int

Steps

1. Use a text editor to open and review the 'videos.csv' file in the 'labwork/udts' directory. Notice the addition of the 'tags' column.

2. Also open and review the 'videos_encoding.csv' file

3. At the prompt, navigate to '/home/ubuntu/labwork/udts'. Launch 'cqlsh' and switch to the 'killrvideo' keyspace.

4. Run the TRUNCATE command to erase the data from the 'videos' table.

5. Alter the 'videos' table to add a 'tags' column.

6. Load the data from the 'videos.csv' file using the COPY command.

```
COPY videos FROM 'videos.csv' WITH HEADER=true;
```

Remember, we do not need to create the user defined type called 'video_encoding' because we did so in the previous exercise. However, take a look at the code below as a refresher. **Do not** run it again or you will get an error!

```
CREATE TYPE video_encoding (
   bit_rates SET<TEXT>,
   encoding TEXT,
   height INT,
   width INT,
);
```

7. Alter your table to add an 'encoding' column of the 'video_encoding' type.

8. Load the data from the 'videos_encoding.csv' file using the COPY command.

```
COPY videos (video_id, encoding) FROM 'videos_encoding.csv' WITH
HEADER=true;
```

9. Run a query to retrieve the first 10 rows of the 'videos' table.

Notice the altered table contains data for the new 'tags' and 'encoding' column.

10. Exit cqlsh.

Exercise 3.2: Using Counters in CQL

In this exercise, you will:

- Create a new table that makes use of the counter type
- Load the newly created table with data
- Run queries against the table to test counter functionality

Background

You have been on a roll lately designing tables, and that hasn't gone unnoticed. Since no good deed goes unpunished, your team has unanimously agreed to let you tackle one of the trickier problems they encountered learning Cassandra.

The problem is with keeping track of the number of videos in a particular category. Whenever a tag is added to a video, a transaction is done which reads the number of videos for that video category, increments it by one, and updates it in the database. However, this technique is susceptible to race conditions. Cassandra is a distributed database that allows for concurrent, simultaneous operations, and it does not allow transactions. Your team has been stumped on how to get this functionality working correctly in Cassandra. Fortunately, you have just read up on the counter type and know that it is the perfect way to solve this problem.

The requirement itself is to be able to support a query that can retrieve the number of videos for a particular category, defined as a specific tag and year added. The query allows querying on a tag, and optionally, on a range for the added year.

Steps

1. Navigate to '/home/ubuntu/labwork/counters' and open the 'videos_count_by_tag.cql' file and review its contents. Notice this is a CQL script rather than a CSV data file.

2. Launch 'cqlsh' and switch to 'killrvideo' keyspace.

3. Create a new table called 'videos_count_by_tag' with a column 'video_count' which makes use of a counter type to store the video count. Structure your table to work correctly with the CQL in 'videos_count_by_tag.cql'.

4. Load the number of counts from the 'videos_count_by_tag.cql' file into the 'videos_count_by_tag' table using the SOURCE command.

5. Run a query to display each category of tag and added year, along with the count of videos for each.

6. Simulate adding another a tag for another video by incrementing the video count for a category, and then querying the new count from the 'videos_count_by_tag' table.

7. Exit cqlsh.

Exercise 4.1: Finish a conceptual data model

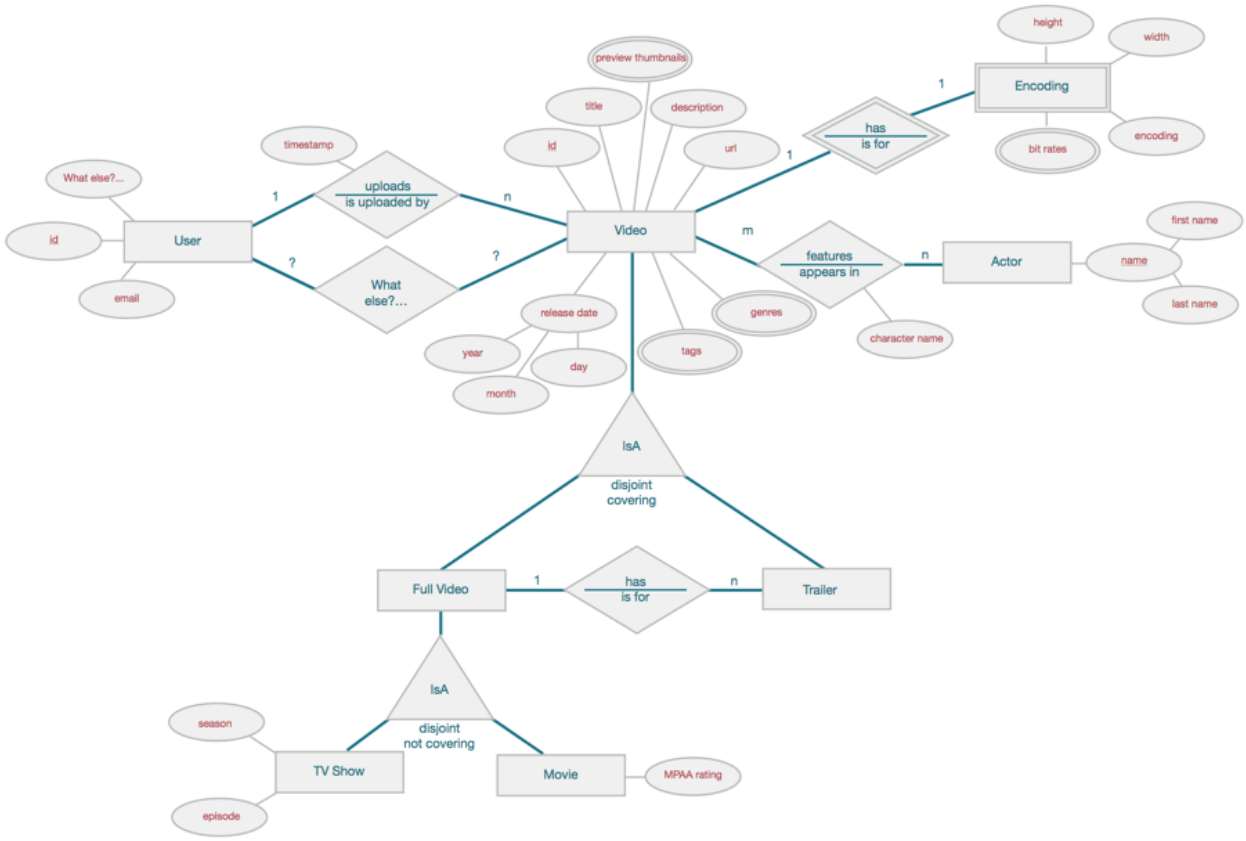
In this exercise, you will:

- Model the KillrVideo entities
- Identify the entity types, relationship types, and attribute types

Steps

Diagram KillrVideo entities and their attributes

Here is an almost complete conceptual model of the KillrVideo domain:



Note the "What else?" areas for both attributes and relationships for User.

1. Using pen and paper or a whiteboard if you are working in groups, identify what further attributes would also describe the User entity type.

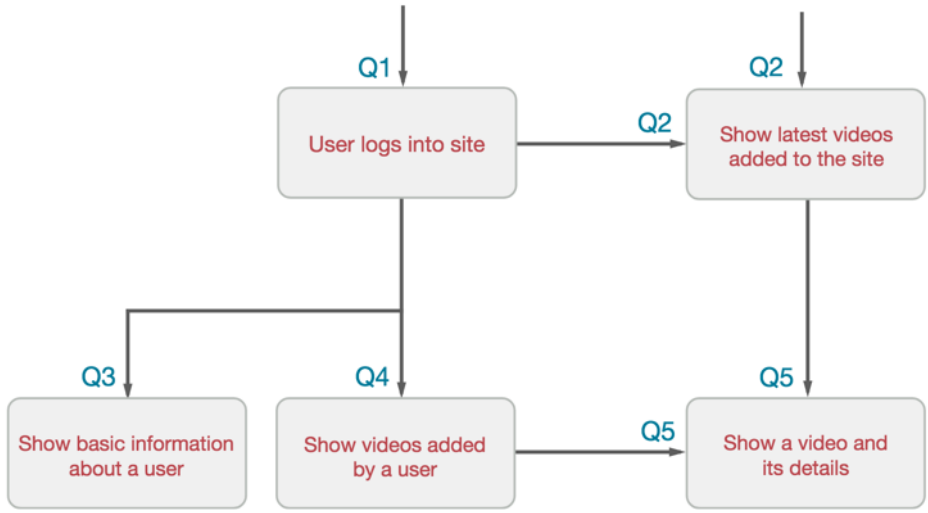
2. Determine what other relationships are necessary between User and Video.

Exercise 4.2: Application Workflow And Access Patterns

In this exercise, you will:

- Finish the application workflow based KillrVideo query requirements
- Add the remaining access patterns based on KillrVideo query requirements

Steps



ACCESS PATTERNS

Q1: Find a user with a **specified email**
Q2: Find most recently uploaded **videos**
Q3: Find a **user with a specified id**
Q4: Find videos uploaded by a **user with a known id**(show most recently uploaded videos first)
Q5: Find a video with a **specified video id**

Using the following KillrVideo tasks, finish filling out the application workflow diagram and access patterns.

- Find comments posted by a user with a known id (show most recently posted comments first)
- Find comments posted for a video with a known id (show most recent comments first)
- Find an average rating with a known video id
- Find trailers for a video with a known id
- Find video interactions for a user with a known id and specified video id (show most recent interactions first)
- Find actors for a video with a known id

Exercise 4.3: Extend The KillrVideo Logical Model

In this exercise, you will:

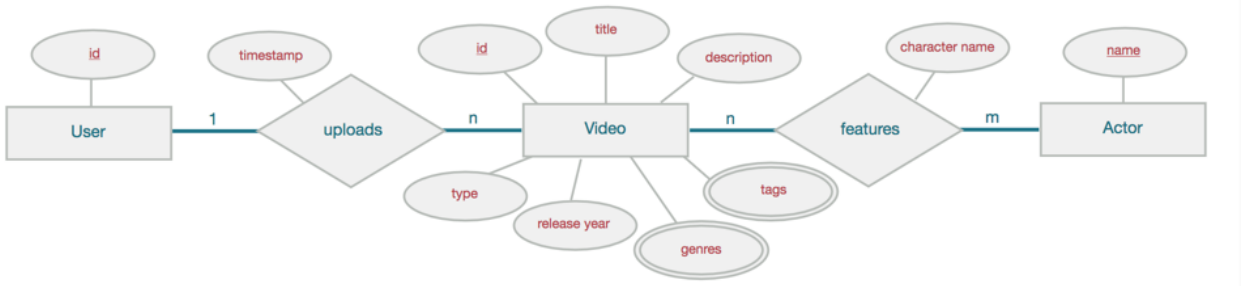
- Add tables to the logical model to support additional queries
- Ensure that the mapping rules are applied appropriately

Background

The KillrVideo product team wants to add additional query capabilities to the application. Specifically the following queries will need to be supported:

- **Q1.** Find all user videos that match a specific tag (show the most recent uploaded videos first)
- **Q2.** Find all movies that features a specific actor and release year range (show the most recent videos first, and then sorted by title)
- **Q3.** Find all movies that features a specific actor, genre and release year range (show the most recent videos first, and then sorted by title)

The conceptual model outlines the available attributes for the appropriate entities and relationships.



Steps

1. On paper or in a text editor, create a logical Chebotko diagram that can support Q1.
2. On paper or in a text editor, create a logical Chebotko diagram that can support Q2.
3. On paper or in a text editor, create a logical Chebotko diagram that can support Q3.
4. For each query, answer the following questions:
    a. What entity or relationship type is being stored in a partition or row?
    b. What are the key attribute(s) for this table?
    c. What attribute is used for the partition key(s) that enabled the equality query?

d. What attribute is used for the cluster column(s) that enables the inequality / range scan?
e. What are the clustering column(s) and ordering that support the required results?

Exercise 4.4: Finalizing Physical Data Modeling

In this exercise, you will:

- Add data types to the physical data model
- Run the CQL CREATE TABLE statements for each table in physical model
- Load data and run some queries to test the physical data model

Background

In this exercise, you will build on your logical data model and finalize your physical data model. Not all your tables have data types for columns.

You will also need to build and run the CREATE TABLE statements for each of the tables, and run some queries to verify that everything is working correctly.

Steps

1. Use a text editor to open and review the `killrvideo.cql` file in the `labwork/final` directory.
2. There are several tables with columns marked with `*CQL Type*`. Fill in the appropriate data type for the columns in `users_by_email`, `users`, `videos_by_user`, and `comments_by_user`.
3. In the `labwork/final` directory, start up `cqlsh`.
4. Run the SOURCE command on the `killrvideo.cql` file to execute the CREATE TABLE statements.

If the SOURCE command was successful, you should now have a new keyspace called `killr_video`.

5. Run the DESCRIBE KEYSPACE command on the `killr_video` keyspace to review the keyspace and table schema.
6. Set the default keyspace to `killr_video` with the USE command.
7. You can now load video data into your new keyspace. Use the COPY command to load data into the following tables:

```
COPY videos FROM 'videos.csv' WITH HEADER=true;COPY latest_videos
FROM 'latest_videos.csv' WITH HEADER=true;COPY trailers_by_video
FROM 'trailers_by_video.csv' WITH HEADER=true;COPY
actors_by_video FROM 'actors_by_video.csv' WITH HEADER=true;
```

If everything is successful, you should now have video data available for reading.

8. Query the latest_videos table to find the most recent 50 videos that was uploaded.
    - Is there a video uploaded for the movie Gone Girl? What is the `video_id` for that movie?
9. Let's find out some more information about this movie. Query the `videos` table using the previously found `video_id`.
    - When was this movie released? What are the genres for this movie?
10. We can also find the actors that were in the movie and the characters they played. Go ahead and query the `actors_by_video` table using the `video_id` for Gone Girl.
    - Who was the actor that played the character Desi Collings?
11. At this point, an interested user might want to watch a trailer for this movie. Query the `trailers_by_video` table to check if there are any trailers available for this movie.
12. If there is a trailer available, make note of the `trailer_id` and then query the `videos` table again using the `trailer_id` value as the equality condition for the `video_id` column.
    - What is the URL for the trailer?

Copy the URL into a web browser and enjoy!

Exercise 5.1: Creating Secondary Indexes

In this exercise, you will:

- Create a secondary index on a table

Background

Based on surveys, focus groups and other user feedback, Killrvideo has learned that users want to be able to determine an actor's name based on the video the actor appeared in and the name of the character that actor played. Also, sometimes they want to determine the actor's name based on the character's name alone because they don't recall the name of the movie. Although secondary indexes aren't a best practice, you have been studying the topic and know that this is one instance where a secondary index could make sense for this need.

Steps

1. Navigate to '/home/ubuntu/labwork/final'.

2. Launch 'cqlsh' and switch to 'killr_video' keyspace.

3. Create a secondary index on the actors_by_video table using the following syntax:

   - CREATE INDEX [index name] ON [table name]([field name])

4. Query the table as follows to validate your work:

   - Search for all rows where the video_id = 87c645e8-0ef2-11e5-98f3-8438355b7e3a AND the character is named Kelly La Fonda.

   - Search for all rows where the character name is George McFly.

Exercise 5.2: Creating Materialized Views

In this exercise, you will:

- Create a materialized view for an existing table.
- Execute a search to validate the design of the materialized view.

Scenario

You have been asked to extend the data model so that the application enables users to see a list of videos uploaded by a specific user. Although the videos table contains both the video details as well as the user that uploaded each video, the primary key only contains the video id. You have read up on materialized views and know you can use a materialized view to fulfill this requirement.

Steps

1. Navigate to '/home/ubuntu/labwork/final'.
2. Launch 'cqlsh' and switch to 'killr_video' keyspace.
3. Create the materialized view per the requirements in the scenario above.
4. Validate the design of your materialized view using user_id `723f6f5f-3658-4449-90d0-4391d63e50a8`. You should retrieve 13 rows.

Exercise 5.3: Implementing Aggregation In Your Data Model

In this exercise, you will:

- Design a logical data model that will be able to store and retrieve aggregated values

Background

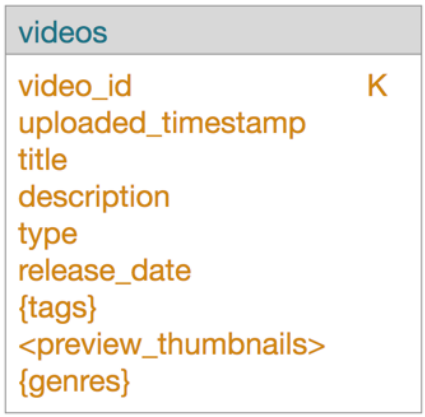
During a peer review of an intern's work, your team found that the intern created a schema with the assumption that the queries could use aggregate functions.

It's back to the drawing board, but this time with you working with the hapless intern to come up with a proper design.

- Requirement #1: KillrVideo must be able to count the number of views for each video in a particular month and year. The count does not have to be 100% accurate.
- Requirement #2: KillrVideo must be able to keep track of the total number of uploaded videos, the combined duration (in seconds) of all of these videos, and the average duration (in seconds) of an uploaded video. These statistics must be stored per day.

They do not have to be 100% accurate.

A logical table schema for the `Video` entity has already been created and looks like:



Steps

1. Review the above requirement #1.
2. The intern made a table to log an entry into whenever a user views a video. The original query would have then used the COUNT aggregate to retrieve the number of views for a particular video.
    - As far as Cassandra is concerned, what is inefficient about the intern's design?

3. Come up with an optimal table schema that meets this requirement using Cassandra.
    - What would be the query that can retrieve the daily count of the number of views for a video for a particular year and month?
    - Is it possible to retrieve the all-time number of views for a video? If possible, what would be the query?
    - Is there anything that needs to be done from the application side?
    - How would your design increment a video's view count? Provide the queries and/or write statements needed to do this.
4. The intern, being ever helpful, suggested that it would be useful to also display the top 10 videos for each month, based on the number of views.
    - Would it be possible to query this information in Cassandra? Why or why not?
    - If not, is there a way to do this outside of Cassandra using your schema?
5. Review the above requirement #2.
6. Modify the existing Videos table and/or come up with your own table schema that will meet these requirements.
    - What is the query to retrieve the total number of videos uploaded on a specific day?
    - What is the query to retrieve the combined duration of all videos uploaded on a specific day?
    - What is the query to get the average duration of all videos uploaded on a specific day?
    - Explain how these values are updated whenever a new video is uploaded.

Exercise 5.4: Using the Cast Function

In this exercise, you will:

- Generate a list of videos along with their release date.

Background

Killrvideo wants to analyze how long films in certain genres remain popular from the release date until present. You've been asked to provide a solution that will enable analysts to view the title along with the release date in coordinated universal time (UTC) format.

Steps

1. First consider the following questions regarding the killr_video keyspace:

    - What table would you use? Why?
    - What filter will you have to apply?
    - How will you filter on the column you decided to use?

2. Navigate to the killr_video keyspace.

3. Launch 'cqlsh'.

4. Write the required cql code to satisfy the requirements and validate your work.

Exercise 5.5: Table Optimizations

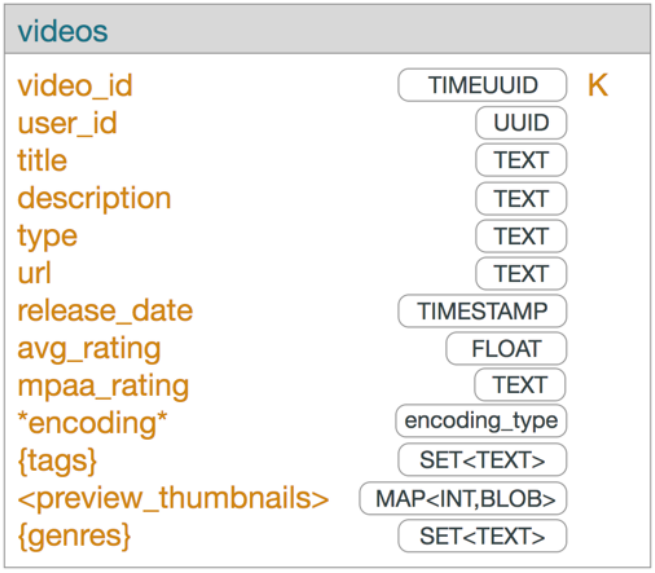
In this exercise, you will:

- Determine what tables might benefit from splitting partitions
- Determine what tables might benefit from splitting tables

Background

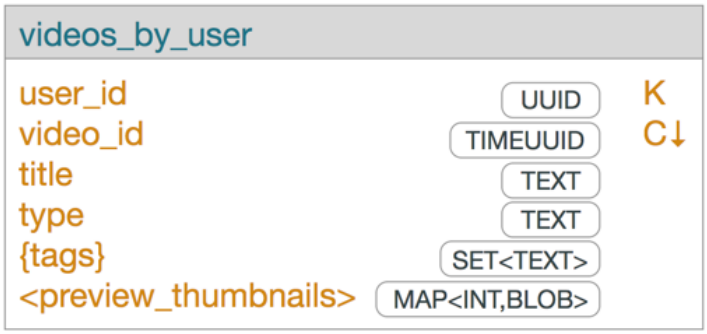
Even though you've already created your data model, there is some room for improvement. You have been assigned to take a look at some of the table diagrams and to analyze whether any further optimization is necessary.

Steps



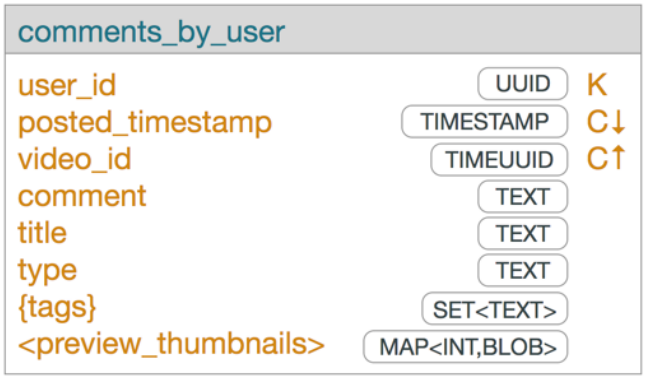
In the `videos` table, there is some concern that the `preview_thumbnails` column might make partitions too large to be manageable. The column contains screen capture images from videos, with one screenshot for every 20 seconds in the video. The average screenshot is about 20 KB and the longest expected video would be 6 hours.

- What would be the estimated size of the column for the worst case of having a 6 hour video?
- Is this table a good candidate for splitting partitions?
- If the table is a candidate, what would be a viable way to split the partitions?



The `videos_by_user` table also has the same `preview_thumbnails` column, but in this case there may be multiple videos nested within a user partition. A user may upload several to hundreds of videos, which means that the thumbnail images would all need to be stored in that one partition.

- Assume that in the worst case, any individual user may upload 500 videos. What would the total size of the `preview_thumbnails` column by for that partition?
- Is this table a good candidate for splitting partitions?
- If the table is a candidate, what would be a viable way to split the partitions?



The last table to consider is the `comments_by_user` table. As before, there is a `preview_thumbnails` column. However this table stores all of the comment and video information for a user in a partition. A user may foreseeably comment on hundreds of videos, which gets stored in one partition along with those preview thumbnail images.

- The partitions will probably be too large, we will consider splitting the data up in some way.
- Are there any viable ways to split the partitions in the `comments_by_user` table?

One other possibility may be to split the table itself.

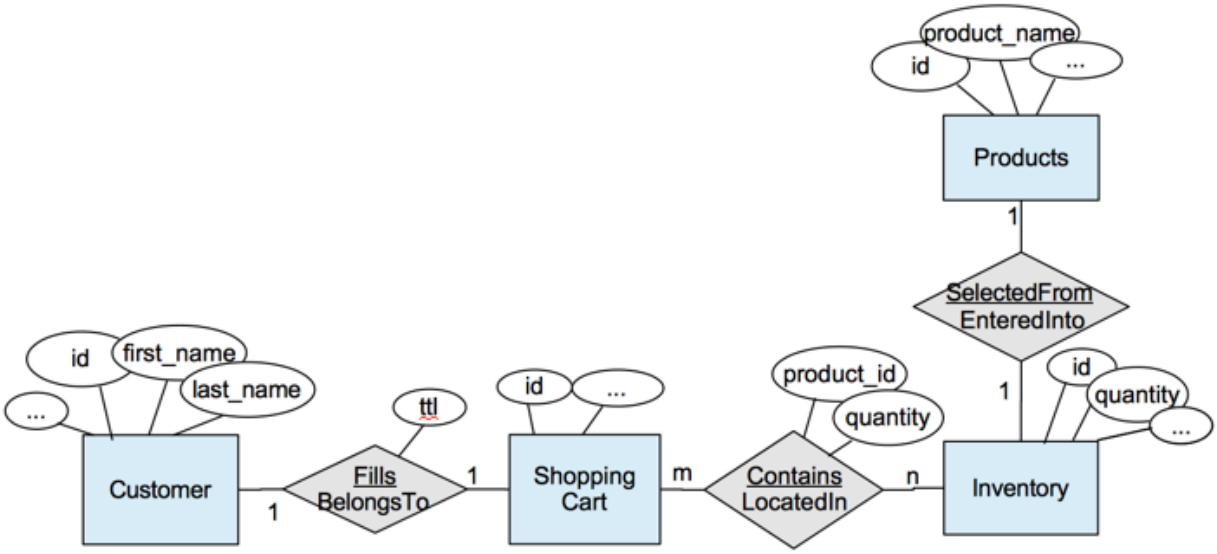
- What columns would remain in the `comments_by_user` table, and what columns will move to a different table?

- What would be the queries that can be used to access the columns in each of the tables?

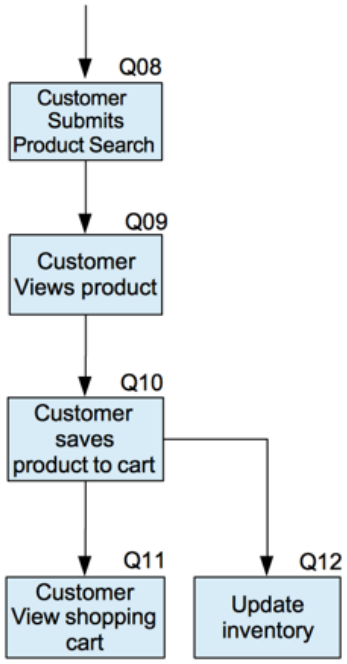
Exercise 6.1: Use Case 1 - Shopping Cart

You are an application developer for Widgets International, a company that manufactures widgets and sells them exclusively via their online storefront. Your company has made the decision to migrate its online storefront to a DSE based application. Analyze the conceptual design and application workflow fragment for this exercise and write the CQL to implement the shopping cart and the product catalog.

Conceptual Design



Application Workflow



Q08. Retrieve a list of products based on product description

Q09. Retrieve product details based on product id

Q10. Insert item into cart based on the customer id and product id of the product selected.

Q11. Retrieve a list of products in the cart based on the customer id.

Q12. Increment or decrement the quantity of a product in inventory based on specific product id and quantity to increment or decrement.

Exercise 6.2: Use Case 2 - Customer Profile

Widgets International loves the shopping cart design for the online storefront scenario presented in the previous exercise and has brought you some new requirements. In conversations with the business, you learn that it will be important to store multiple phone numbers, addresses and email addresses for each customer. This data will be read/write intensive and it is important to keep latency to 50ms or less. Also, since passwords will be stored, encryption is mission-critical. Analyze the same conceptual design and application workflow from the previous exercise and write the CQL to implement these customer profile requirements.

Exercise 6.3: Use Case 3 - Sensor Event Tracking

Widgets International is so pleased with the work that you have done in the previous two exercises that they are tasking you to solve a new problem related to their manufacturing facilities.

The widgets made by your company are sensitive, high tech devices, and each machine used in the manufacturing process contains one or more sensors that captures data such as temperature, speed of a moving part, etc. This data is now going to be stored in a DSE cluster that you must design and maintain. Once the application is up and running, the data it captures will be provided to analytics tool chains.

The application will definitely be write heavy; tens of millions of events will be generated per second, and your company has an SLA requiring that all events be persisted across multiple nodes in your DSE cluster in less than 10ms.

Analyze the conceptual design below and write the CQL to implement the table(s) needed to store this data.