Solution: Setting Up Lab Environment

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 you will SSH into.

2. SSH to the cloud instance using the IP address and key provided.

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

Run cqlsh

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'.

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

2. 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.

3. Type 'exit' to quit 'cqlsh'.

Solution: Basic CQL Fundamentals

General setup

Ensure you are are in the /labwork/cql directory and you have started 'cqlsh'.

Creating Keyspace and USE

```
CREATE KEYSPACE killrvideo
WITH replication = {'class':'SimpleStrategy','replication_factor': 1};

USE killrvideo;
```

Create videos table

```
CREATE TABLE videos (
  video_id TIMEUUID,
  added_date TIMESTAMP,
  description TEXT,
  title TEXT,
  user_id UUID,
  PRIMARY KEY (video_id)
);
```

COPY data into videos table

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

Verify with SELECT

```
SELECT * FROM videos LIMIT 10;
SELECT COUNT(*) FROM videos;
SELECT * FROM videos WHERE video_id = 6c4cffb9-0dc4-1d59-af24-
c960b5fc3652;
```

Remove data from table and exit 'cqlsh'

```
truncate videos;
exit
```

Solution: Working with Partitions

General setup

Ensure you are in /labwork/partitions/ directory and have started cqlsh. Ensure you have switched to the killrvideo keyspace.

Creating table videos_by_title_year

```
CREATE TABLE videos_by_title_year (
        title TEXT,
        added_year INT,
        added_date TIMESTAMP,
        description TEXT,
        user_id UUID,
        video_id TIMEUUID,
        PRIMARY KEY ((title, added_year))
);
```

Loading data into table

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

Querying the table

```
SELECT * FROM videos_by_title_year WHERE title = 'Introduction To
Apache Cassandra' AND added_year = 2014;
SELECT * FROM videos_by_title_year WHERE title = 'Sleepy Grumpy
Cat' AND added_year = 2015;
SELECT * FROM videos_by_title_year WHERE title = 'Grumpy Cat:
Slow Motion' AND added_year = 2015;
SELECT * FROM videos_by_title_year WHERE title = 'AzureDev' AND
added_year = 2015;
```

All of these above queries will work for the students.

Querying table with bad partition key

```
SELECT * FROM videos_by_title_year WHERE title = 'Introduction To
Apache Cassandra';
```

```
SELECT * FROM videos_by_title_year WHERE added_year = 2015;
```

```
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you want to execute this query despite the performance
unpredictability, use ALLOW FILTERING"
```

This query fails because of not using the whole partition key.
Since both `title` and `added_year` are part of the partition key, both are required to query.

Solution: Clustering Columns

General setup

Ensure you are in /labwork/clustering/ directory, that you've started cqlsh and you are using the killrvideo keyspace.

Create upserts

```
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))
);
```

DESCRIBE bad_videos_by_tag_year ;

Load bad table

```
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.

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.

Drop bad table

```
DROP TABLE bad_videos_by_tag_year;
```

Create table videos_by_tag_year

```
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 ((tag), added_year, video_id)
) WITH CLUSTERING ORDER BY (added_year desc, video_id asc);
```

Load data

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

Query table

```
SELECT * FROM videos_by_tag_year WHERE tag = 'trailer' AND
added_year = 2015;
SELECT * FROM videos_by_tag_year WHERE tag = 'cql' AND added_year
= 2014;
SELECT * FROM videos_by_tag_year WHERE tag = 'spark' AND
added_year = 2014;
```

Count rows

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

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.

Query all videos before 2015

```
SELECT * FROM videos_by_tag_year WHERE added_year < 2015;
```

This fails because the partition column `tag` was not queried, which would mean all of the partitions in the table would have to be filtered to get the results for the query.

Solution: Denormalizing

General setup

Ensure you are in /labwork/denormalization/, you've started cqlsh and you have switched to the killrvideo keyspace.

Creating UDT for encoding

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

Create table videos_by_actor

```
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 ((actor), added_date, video_id, character_name)
) WITH CLUSTERING ORDER BY (added_date DESC, video_id ASC,
character_name ASC);
```

Load 'videos_by_actor.csv'

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

Query videos_by_actor table

```
SELECT * FROM videos_by_actor WHERE actor = 'Leonardo DiCaprio'
LIMIT 10;
```

```
SELECT actor, added_date FROM videos_by_actor WHERE actor =
'Leonardo DiCaprio' LIMIT 10;
```

Create a videos_by_genre table

```
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 ((genre), added_date, video_id)
) WITH CLUSTERING ORDER BY (added_date DESC, video_id ASC);
```

Load 'videos_by_genre.csv'

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

Query a genre

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

Solution: User Defined Types (UDTs)

General setup

Ensure you are in /labwork/udts/, you have started cqlsh and you are in the killrvideo keyspace.

Truncate videos table

```
truncate videos;
```

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

```
ALTER TABLE videos ADD tags SET<TEXT>;
```

Load data

```
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,
);
```

Add an 'encoding' column of the 'video_encoding' type

```
ALTER TABLE videos ADD encoding FROZEN<video_encoding>;
```

Load the data from the 'videos_encoding.csv'

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

Query first 10 rows

```
SELECT * FROM videos LIMIT 10;
```

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

Solution: Using Counters in CQL

General steps

Ensure you are in /labwork/counters/, you've started cqlsh and you are in the killvideo keyspace.

Create table 'videos_count_by_tag' with a column 'video_count'

```
CREATE TABLE videos_count_by_tag (
   tag TEXT,
   added_year INT,
   video_count counter,
   PRIMARY KEY (tag, added_year)
);
```

Load 'videos_count_by_tag.cql' file

```
SOURCE 'videos_count_by_tag.cql'
```

Query table

```
SELECT * FROM videos_count_by_tag;
```

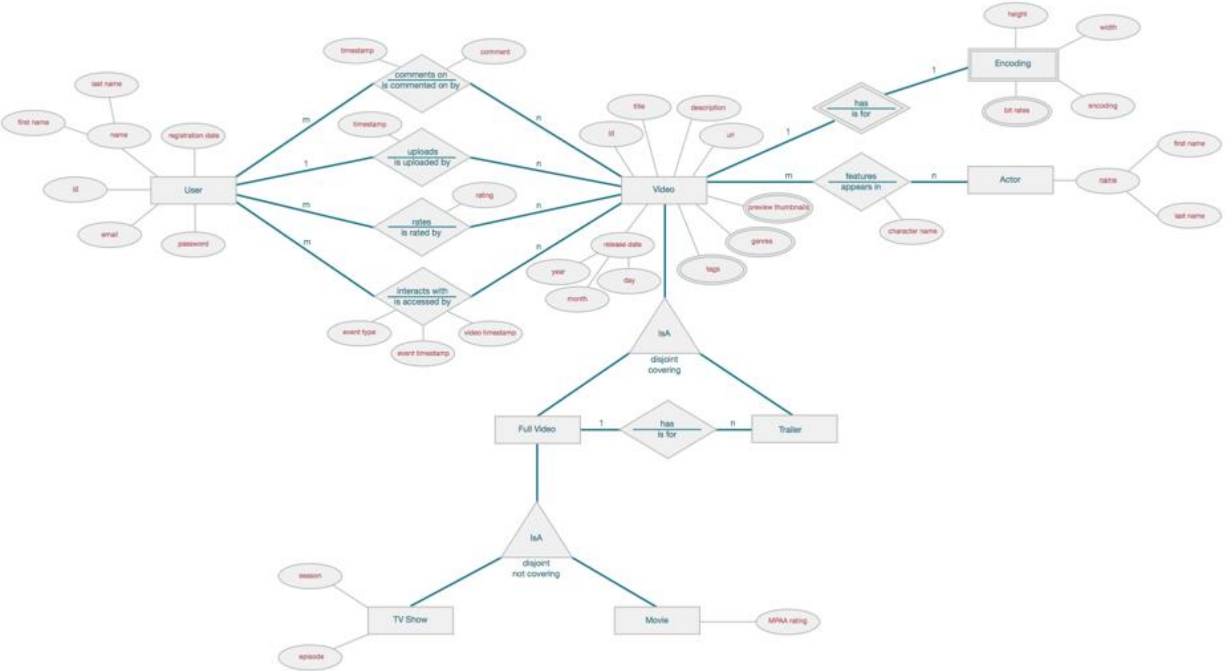
Update table to increment video

```
UPDATE videos_count_by_tag SET video_count = video_count + 1
WHERE tag = 'cassandra' AND added_year = 2015;

SELECT * FROM videos_count_by_tag WHERE tag = 'cassandra' AND
added_year = 2015;
```

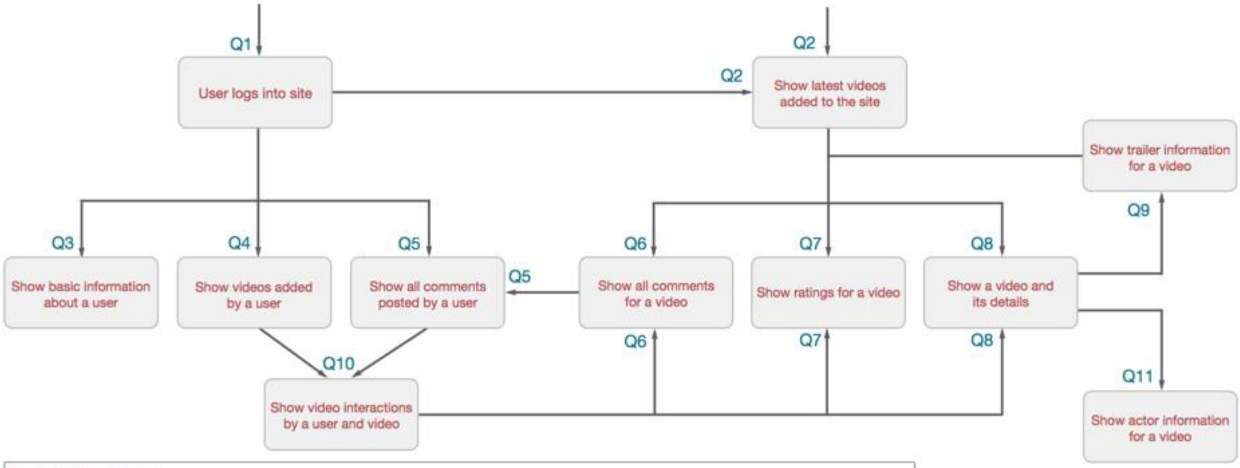
Solution: Finish a conceptual data model

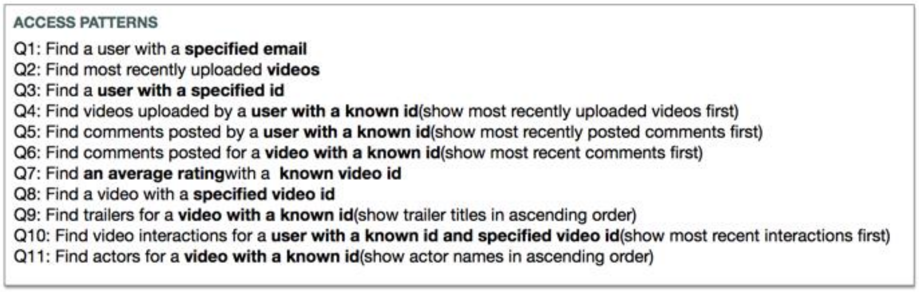
Diagram KillrVideo entities and their attributes



Solution: Application Workflow and Access Patterns

Finalized workflow and access patterns


Q1
User logs into site
Q2
Q2
Show latest videos added to the site
Show trailer information for a video
Q9
Q3
Q4
Q5
Q6
Q7
Q8
Show basic information about a user
Show videos added by a user
Show all comments posted by a user
Q5
Show all comments for a video
Show ratings for a video
Show a video and its details
Q10
Q6
Q7
Q8
Q11
Show video interactions by a user and video
Show actor information for a video



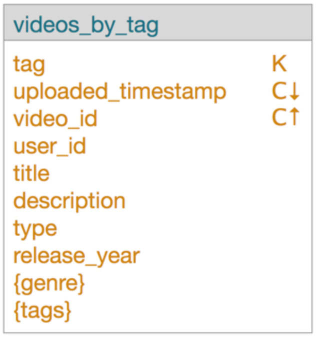
ACCESS PATTERNS

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 comments posted by a **user with a known id**(show most recently posted comments first)
Q6: Find comments posted for a **video with a known id**(show most recent comments first)
Q7: Find **an average rating**with a **known video id**
Q8: Find a video with a **specified video id**
Q9: Find trailers for a **video with a known id**(show trailer titles in ascending order)
Q10: Find video interactions for a **user with a known id and specified video id**(show most recent interactions first)
Q11: Find actors for a **video with a known id**(show actor names in ascending order)

Solution: Extend The KillrVideo Logical Model

- **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)

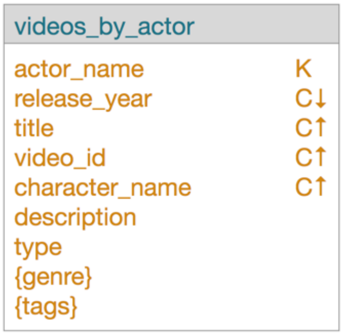
Q1



videos_by_tag	
tag	K
uploaded_timestamp	C↓
video_id	C↑
user_id	
title	
description	
type	
release_year	
{genre}	
{tags}	

- What entity type is being stored in a partition or row?
  - Each partition holds the `User posts Video` relationship.
  - Each row represents the video and user information for a specific tag.
- What are the key attribute(s) for this table?
  - The key attribute is `video_id`.
- What attribute is used for the partition key(s) that enables the query?
  - The equality attribute is `tag`.
- What are the clustering column(s) and ordering that support the required results?
  - The clustering column that support ordering is *`uploaded_timestamp`* *(DESC)*.

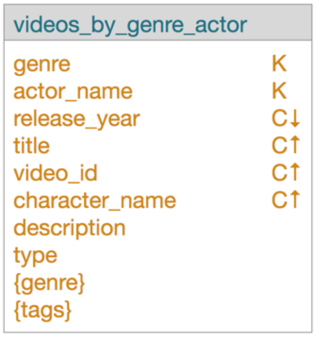
Q2



videos_by_actor	
actor_name	K
release_year	C↓
title	C↑
video_id	C↑
character_name	C↑
description	
type	
{genre}	
{tags}	

- What entity or relationship type is being stored in a partition or row?
  - Each partition represents a `Video features Actor` relationship.
  - Each row represents information for a video the actor was featured in.
- What are the key attribute(s) for this table?
  - The key attributes are `actor_name` and `video_id`.
- What attribute is used for the partition key(s) that enabled the query?
  - The equality attribute is `actor_name`.
- What attribute is used for the cluster column(s) that enables the inequality / range scan?
  - The inequality attribute is `release_year`.
- What are the clustering column(s) and ordering that support the required results?
  - The clustering columns that support ordering is *`release_year`* *(DESC)* and *`title`* *(ASC)*.

Q3



videos_by_genre_actor	
genre	K
actor_name	K
release_year	C↓
title	C↑
video_id	C↑
character_name	C↑
description	
type	
{genre}	
{tags}	

- What entity or relationship type is being stored in a partition or row?
    - Each partition represents a `Video features Actor` relationship.
    - Each row represents video information for a specific genre and actor.
- What are the key attribute(s) for this table?
    - The key attributes are `actor_name` and `video_id`.
- What attribute is used for the partition key(s) that enabled the query?
    - The equality attributes are `genre` and `actor_name`.
- What attribute is used for the cluster column(s) that enables the inequality / range scan?
    - The inequality attribute is `release_year`.
- What are the clustering column(s) and ordering that support the required results?
    - The clustering columns that support ordering is *`release_year`* *(DESC)* and *`title`* *(ASC)*.

Exercise: 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

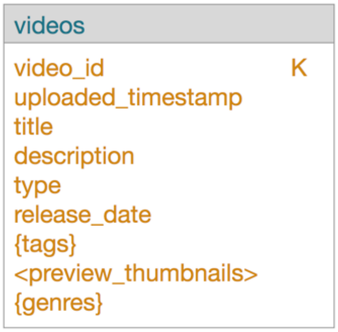
During a peer review of an intern's work, your team found that the intern created a schema with the assumption that the queries could use aggregate functions.

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:



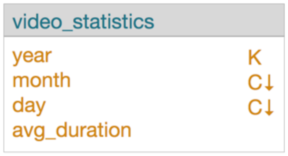
videos

video_id K
uploaded_timestamp
title
description
type
release_date
{tags}
<preview_thumbnails>
{genres}

- Review the above requirement #1.
- 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?
- Come up with an optimal table schema that meets this requirement using Cassandra.
    - What would be the query that can retrieve the count of the number of views for a video for a particular year and month?

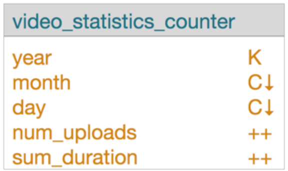
- Is it possible to retrieve the all-time number of views for a video? If possible, what would be the query?
- Is there anything that needs to be done from the application side?
- How would your design increment a video's view count? Provide the queries and/or write statements needed to do this.

- 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?
- Review the above requirement #2.
- Modify the existing Videos table and/or come up with your own table schema that will meet these requirements.



video_statistics

year K
month C↓
day C↓
avg_duration



video_statistics_counter

year K
month C↓
day C↓
num_uploads ++
sum_duration ++

- What is the query to retrieve the total number of uploaded videos?

```
SELECT num_uploads FROM video_statistics_counter WHERE year = ?
AND month = ? AND day = ?;
```

- What is the query to retrieve the combined duration of all uploaded videos?

```
SELECT sum_duration FROM video_statistics_counter WHERE year = ?
AND month = ? AND day = ?;
```

- What is the query to get the average duration of an uploaded video?

```
SELECT avg_duration FROM video_statistics WHERE year = ? AND
month = ? AND day = ?;
```

Explain how these values are updated whenever a new video is uploaded. Retrieve the duration (in seconds) of the uploaded video.

- The columns in `video-statistics-counter` would be updated first, using the current date:

  ```
  UPDATE video-statistics-counter SET num_uploads = num_uploads +
  1, sum_duration = sum_duration + ?
  WHERE year = ? AND month = ? AND day = ?;
  ```

- Retrieve the new number of uploaded videos and total duration:

  ```
  SELECT num_videos, sum_duration FROM video_statistics_counter
  WHERE YEAR = ? AND month = ? AND day = ?;
  ```

- Calculate the average duration on the application side using the formula `sum_duration` / `num_videos`.
  Store the result in `avg_duration` in `video-statistics`:

  ```
  UPDATE video-statistics SET avg_duration = ?
  WHERE year = ? AND month = ? AND day = ?;
  ```

Solution: Finalizing Physical Data Modeling

General setup

Ensure you are in /labwork/final/ directory, cqlsh is running and you are in the killrvideo keyspace.

Create the tables

Instructor note: The students will be using the slides as guidance to finish filling out the physical data model.

```
CREATE TABLE users_by_email (
   email TEXT,
   password TEXT,
   user_id UUID,
   PRIMARY KEY ((email))
);

CREATE TABLE users (
   user_id UUID,
   email TEXT,
   first_name TEXT,
   last_name TEXT,
   registration_date TIMESTAMP,
   PRIMARY KEY((user_id))
);

CREATE TABLE videos_by_user (
   user_id UUID,
   video_id TIMEUUID,
   title TEXT,
   type TEXT,
   tags SET<TEXT>,
   preview_thumbnails MAP<INT,BLOB>,
   PRIMARY KEY ((user_id), video_id))
WITH CLUSTERING ORDER BY (video_id DESC);

CREATE TABLE comments_by_user (
   user_id UUID,
   posted_timestamp TIMESTAMP,
   video_id TIMEUUID,
   comment TEXT,
   title TEXT,
   type TEXT,
   tags SET<TEXT>,
   preview_thumbnails MAP<INT,BLOB>,
   PRIMARY KEY ((user_id), posted_timestamp, video_id))
WITH CLUSTERING ORDER BY (posted_timestamp DESC, video_id ASC);
```

Instructor note: You can suggest taking a look at some of the other tables that already have data types if students are having trouble selecting data types to columns.

Load the schema

```
SOURCE 'killrvideo.cql';
```

Look at schema

```
DESCRIBE KEYSPACE killr_video;
```

Switch to new killr_video keyspace

```
USE killr_video;
```

Copy data into tables from csv files

```
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;
```

Query to test funtionality

```
SELECT * FROM latest_videos LIMIT 50;
```

Yes, the movie `Gone Girl` exists and was uploaded with the `video_id` 8a657435-0ef2-11e5-91b1-8438355b7e3a.

```
SELECT * FROM videos WHERE video_id = 8a657435-0ef2-11e5-91b1-
8438355b7e3a;
```

The movie was first released on September 26, 2014, and is categorized as *Drama*, *Mystery*, and *Thriller*.

```
SELECT * FROM actors_by_video WHERE video_id = 8a657435-0ef2-
11e5-91b1-8438355b7e3a;
```

The character Desi Collings was played by the actor Neil Patrick Harris.

```
SELECT * FROM trailers_by_video WHERE video_id = 8a657435-0ef2-
11e5-91b1-8438355b7e3a;
```

There are three trailers for this movie, and the `trailer_id` of the first one is 8a65751c-0ef2-11e5-9cac-8438355b7e3a.

```
SELECT * FROM videos WHERE video_id = 8a65751c-0ef2-11e5-9cac-
8438355b7e3a;
```

Exercise: Table Optimizations

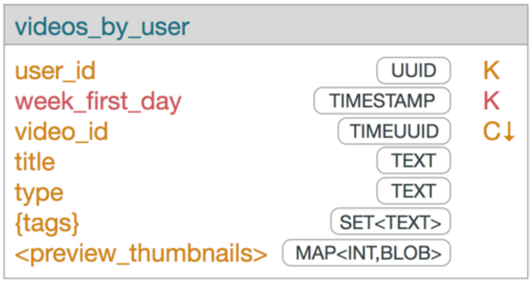
Given the worst case scenario, the `preview_thumbnails` column would be: 6 hours * 60 minutes per hour * 60 seconds per minute / 20 seconds per image * 20 KB per image = 21,600 KB or ~21 MB

Since there is only one video stored per partition in the `videos` table, these partitions should be manageable.

Using the column size of 21 MB that we just calculated, we can multiply by 500 to get the total size of 10,500 MB or 10 GB.

The size of this partition may make it unmanageable, performance-wise, and would benefit from splitting the partition in some way.

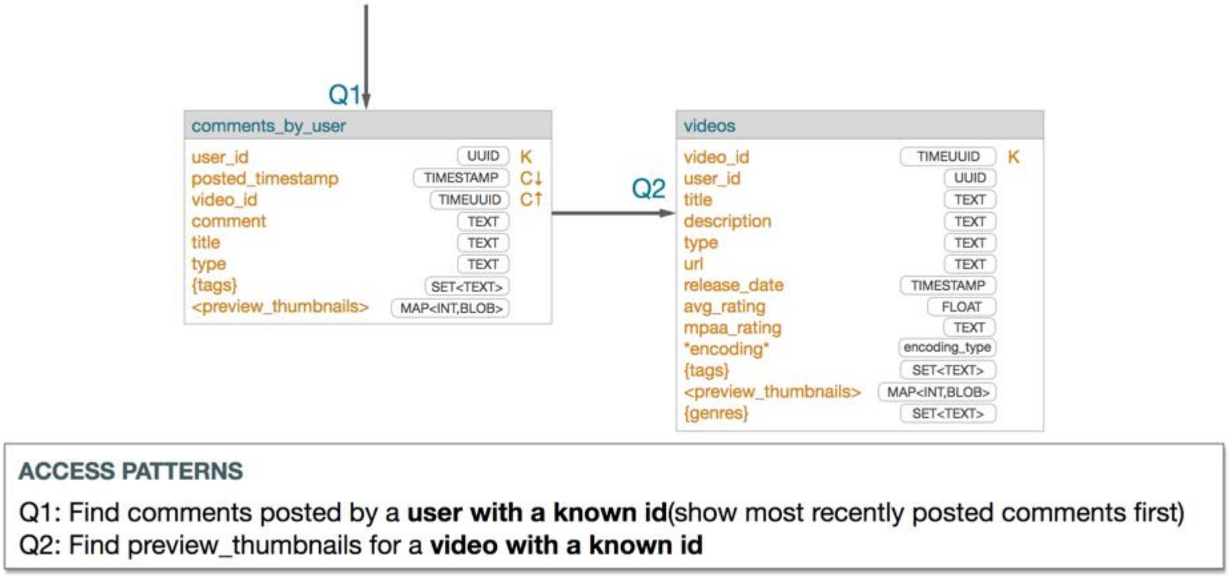
One possibility would be to add a part of uploaded timestamp as the partition key, which will only store in one partition the videos uploaded by the user during a certain time range. Reasonable durations may include all videos uploaded within an hour period, a day, or even a week.



videos_by_user		
user_id	UUID	K
week_first_day	TIMESTAMP	K
video_id	TIMEUUID	C↓
title	TEXT	
type	TEXT	
{tags}	SET<TEXT>	
<preview_thumbnails>	MAP<INT,BLOB>	

The column that may make the most sense would be the `posted_timestamp` column. However since it may be possible for users to post numerous comments in a short amount of time, finding an appropriate duration may be a challenge.

At a minimum, it would be very beneficial to split the table so that the `preview_thumbnails` column is centralized in a different table. If we take a look at the `videos` table, it already contains the `preview_thumbnails` column. After splitting the `comments_by_user` table, a query will first be directed to that table, and after retrieving the video_id(s), the `video`table can then be queried to retrieve the `preview_thumbnails` data for those videos.



comments_by_user		
user_id	UUID	K
posted_timestamp	TIMESTAMP	C↓
video_id	TIMEUUID	C↑
comment	TEXT	
title	TEXT	
type	TEXT	
{tags}	SET<TEXT>	
<preview_thumbnails>	MAP<INT,BLOB>	

videos		
video_id	TIMEUUID	K
user_id	UUID	
title	TEXT	
description	TEXT	
type	TEXT	
url	TEXT	
release_date	TIMESTAMP	
avg_rating	FLOAT	
mpaa_rating	TEXT	
encoding	encoding_type	
{tags}	SET<TEXT>	
<preview_thumbnails>	MAP<INT,BLOB>	
{genres}	SET<TEXT>	

ACCESS PATTERNS

Q1: Find comments posted by a **user with a known id**(show most recently posted comments first)
Q2: Find preview_thumbnails for a **video with a known id**