CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

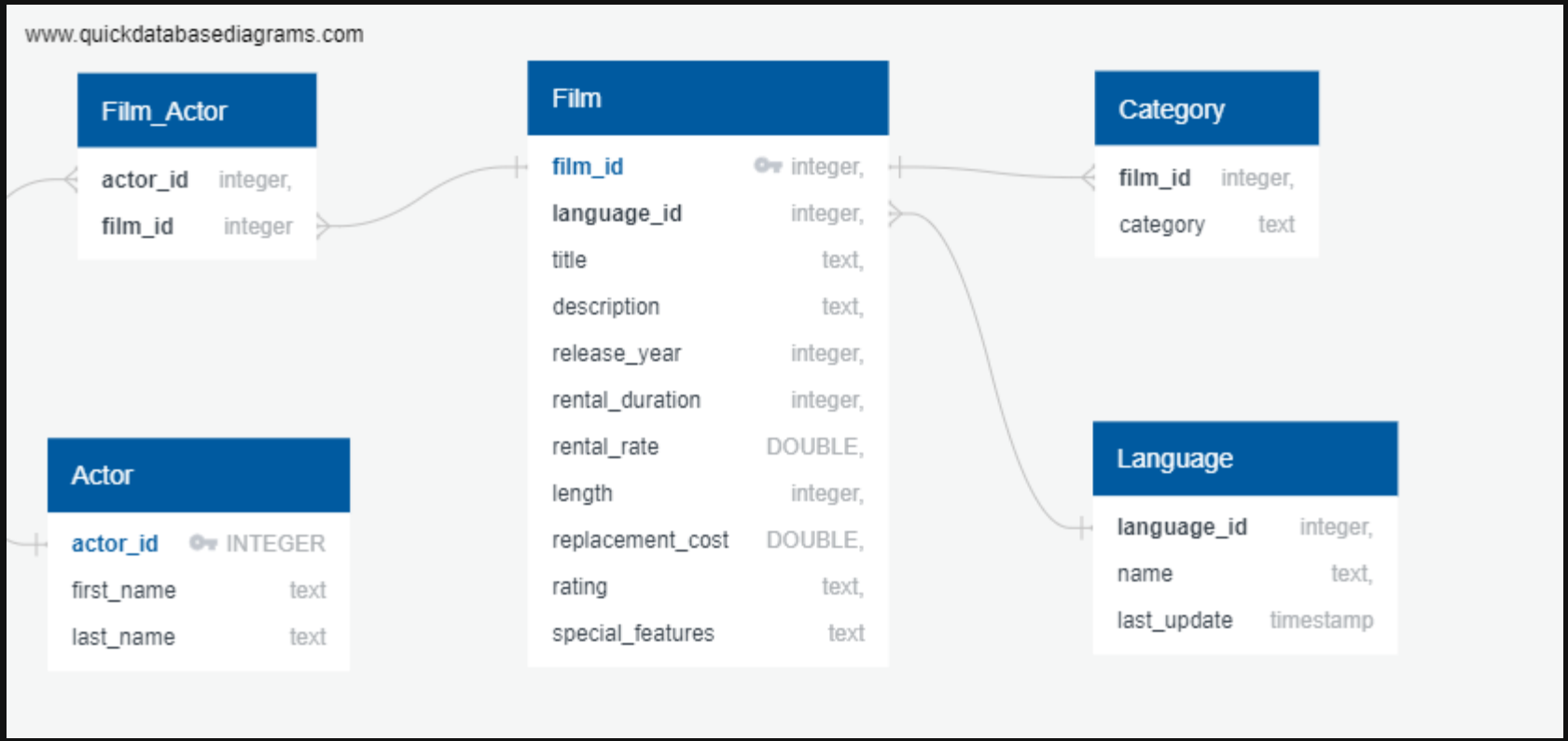
Basics of SQL Querying

Goals

- Understand the basics of building SQL queries
- Get ready for SQL interview

Sections

1. Understanding the `SELECT` statement
2. Aggregation
3. Joins


www.quickdatabasediagrams.com
Film_Actor
actor_id integer,
film_id integer
Film
film_id integer,
language_id integer,
title text,
description text,
release_year integer,
rental_duration integer,
rental_rate DOUBLE,
length integer,
replacement_cost DOUBLE,
rating text,
special_features text
Category
film_id integer,
category text
Actor
actor_id INTEGER
first_name text
last_name text
Language
language_id integer,
name text,
last_update timestamp

Section 1: SELECT Statment

```
SELECT title, rental_rate, replacement_cost
FROM film
WHERE release_year BETWEEN 2004 AND 2010
ORDER BY rental_year DESC, replacement_cost
```

What is a SELECT statement

- Used when we want to get data
- Alternative SQL statements would be
  - `CREATE` , `DELETE` , `UPDATE` , `ALTER`
- Returns a table with out specified columns and filtered records, from our chosen table/s

Section 1: SELECT Clause

```
SELECT *
FROM film
```

```
SELECT title, release_year
FROM film
```

Notes

- Used to specify the columns we would like to return
- We can use `*` to return all of the columns from a table
- We can specify 1 or more columns by name separated by a comma

Section 1: FROM Clause

```
SELECT title, release_year
FROM film
```

```
SELECT *
FROM film, category
```

Notes

- Use is most SQL queries to state the table/s you are working with
- If you don't specify the table, the operation can not be performed
- Should be written as the second clause in the `SELECT` statement
- Can work with a single more multiple tables (`CROSS JOIN`)

Section 1: WHERE Clause

```
SELECT title, release_year
FROM film
WHERE release_year = 2010
```

```
film_table = [{ ... }] # List of dictinoaries as our table
query_results = []

select_columns = ['title', 'release_year'] # SELECT

for row in film_table: # FROM
    if (row["release_year"] == 2010): # WHERE
        query_results.append({
            key: row[key]
            for key in selected_columns # IMPLEMENT SELECT
        })

print(output)
```

Notes

- Used to filter values in our query
- Mostly uses the same comparison operators as Python
  - <, >, =, <>
- Can be thought of like an IF statement in Python
- We can look at the entire `SELECT` statement through Python

Section 1: Logical Operators

```
...
WHERE release_year > 2005 AND rental_rate < 3
```

```
...
WHERE release_year <> 2005 OR release_year = 2010
```

```
...
WHERE release_year BETWEEN 2005 AND 2010
```

```
...
WHERE release_year IN (2005, 2008, 2010)
```

```
...
WHERE release_year NOT IN (2005, 2008, 2010)
```

Notes

- Use the same as you would in Python
- `BETWEEN` is inclusive of the lower and upper bound
- `NOT` can be used to check for inequality
  - Works with any logical operation
- Keep in mind the Python translation, you can chain conditions in the `WHERE` the same way you can in an `IF`

Section 1: Comparing Strings

```sql
SELECT title, release_year
FROM film
WHERE title LIKE '%ANNIE'
```

```sql
SELECT title, release_year
FROM film
WHERE LENGTH(title) < 20
```

Notes

- Similar to Python, all of the other comparison and logical operators will with with strings
- The `LIKE` operator lets you find sub-strings,
  - Uses `%` in the string
  - `%value` - where the string ends with `value`
  - `value%` - Where the string starts with `value`
  - `%value%` - Where `value` appears in the string

Section 1: Comparing Dates

```
...
WHERE payment_date > '2005-01-01'
```

```
SELECT payment_id AS over_due_id
FROM payment
--5 days ago from today
WHERE payment_date > DATEADD(day, -5, GETDATE())
```

Notes

- We can use the same comparison and logical operators on dates are we can on other variables
- To compare dates, we use a string literal with the format `YYYY-MM-DD`
- There are a lot of custom date functions, feel free to look into them

Section 1: ORDER BY

```sql
SELECT release_year, rental_rate
FROM film
ORDER BY release_year
```

```sql
SELECT release_year, rental_rate
FROM film
ORDER BY release_year DESC
```

```sql
SELECT release_year, rental_rate
FROM film
ORDER BY release_year DESC, rental_rate
```

Notes

- Choose the column that you want to order by
- Set how you want to order
  - `DESC` - Descending Order
  - `ASC` - Ascending order (implicit)
- You can order by more than one column

Section 1: Recap

Clauses

- SELECT
- FROM
- WHERE
- ORDER BY

Points

- SELECT
  - Takes in a column, or multiple columns that we would like to display
- FROM
  - Tells SQL which table/s we are working with
- WHERE
  - Allows us to filter our query
- ORDER BY
  - Sorts the results of our query

QUESTIONS ?

Section 1: Order of Execution

How we write the code

1. SELECT
2. FROM
3. WHERE
4. ORDER BY

How SQL Runs the Code

1. FROM
2. WHERE
3. SELECT
4. ORDER BY

Section 1: Execution Order

Code Example

```
...

for row in film_table: # FROM
    if (row["release_year"] == 2010): # WHERE
        query_results.append({
            key: row[key]
            for key in selected_columns # SELECT
        })

query_results = sorted(query_result, key=lambda item: item["release_year"]) # ORDER BY

print(output)
```

Section 1.5: Aliasing

```
SELECT f.title, l.name
FROM film AS f, language AS l
WHERE f.language_id = l.language_id
```

```
SELECT f.title AS 'film title', l.name as film_language
FROM film AS f, language AS l
WHERE f.language_id = l.language_id
AND l.name = 'English'
ORDER BY film_language
```

Notes

- Aliases give us a short name to call when writing our queries
- We are able to access the table aliases on the `SELECT` even though they are only defined a line later
- We can add aliases to columns in the `SELECT` clause
- We can not use the aliases on the `WHERE` clause because `SELECT` comes after `WHERE` in the order of execution
- We can use the aliases in the `ORDER BY` clause

QUESTIONS ?

Section 2: Aggregation

Functions

- `COUNT`
- `MIN`
- `MAX`
- `SUM`
- `AVG`

```sql
-- Get total number of records
SELECT COUNT(*) AS total_films
FROM film

-- Sum of all rental_rates
SELECT SUM(rental_rate) AS total_rental_rate
FROM film
```

Notes

- Used to summarize data
- `COUNT` can take in `*` to return a total number of records
- all aggregation functions take in a column name
- They return a single value
- Important that you add an alias as the disply name will be the function name which will make undertanding the value hard

Section 2: GROUP BY

```
SELECT release_year, SUM(rental_rate) AS total_rental_rate
FROM film
GROUP BY release_year
ORDER BY total_rental_rate
```

Without GROUP BY

```
-- For 2005
SELECT SUM(rental_rate) AS total_rental_rate_2005
FROM film
WHERE release_year = 2005

-- For 2006
SELECT SUM(rental_rate) AS total_rental_rate_2005
FROM film
WHERE release_year = 2006
```

Notes

- Used when we want to select an aggregation function and another column
- Used `GROUP BY` on the table that is not performing the aggregation
- Outputs the aggregate for each distinct value in the column we are grouping by
- Alternative to `GROUP BY` is to run a `SELECT` statement for every year (could get time consuming)

Section 2: Order of Execution

Full Order of Execution

1. FROM / JOIN
2. WHERE
3. `GROUP BY`
4. HAVING
5. SELECT
6. ORDER BY

Notes

- `GROUP BY` comes after `WHERE`
  - We can not use `WHERE` to filter the aggregated values
- `GROUP BY` comes before `SELECT`
  - We can not use aliases from the `SELECT` clause to group

Section 2: Order of Execution

```
SELECT release_year AS year, SUM(rental_rate) AS total_rental_rate
FROM film
GROUP BY release_year -- We cannot use the alias `year`
ORDER BY year -- We can use the alias year
```

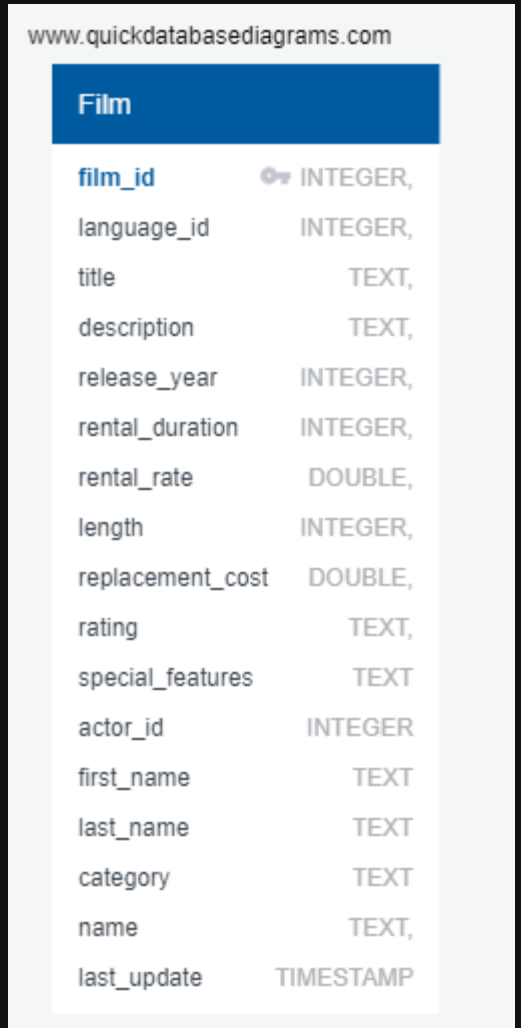
Section 2: HAVING

```sql
SELECT release_year AS year, SUM(rental_rate) AS total_rental_rate
FROM film
GROUP BY release_year
HAVING SUM(rental_rate) < 50 -- Comes before SELECT in the order of execution
ORDER BY year
```

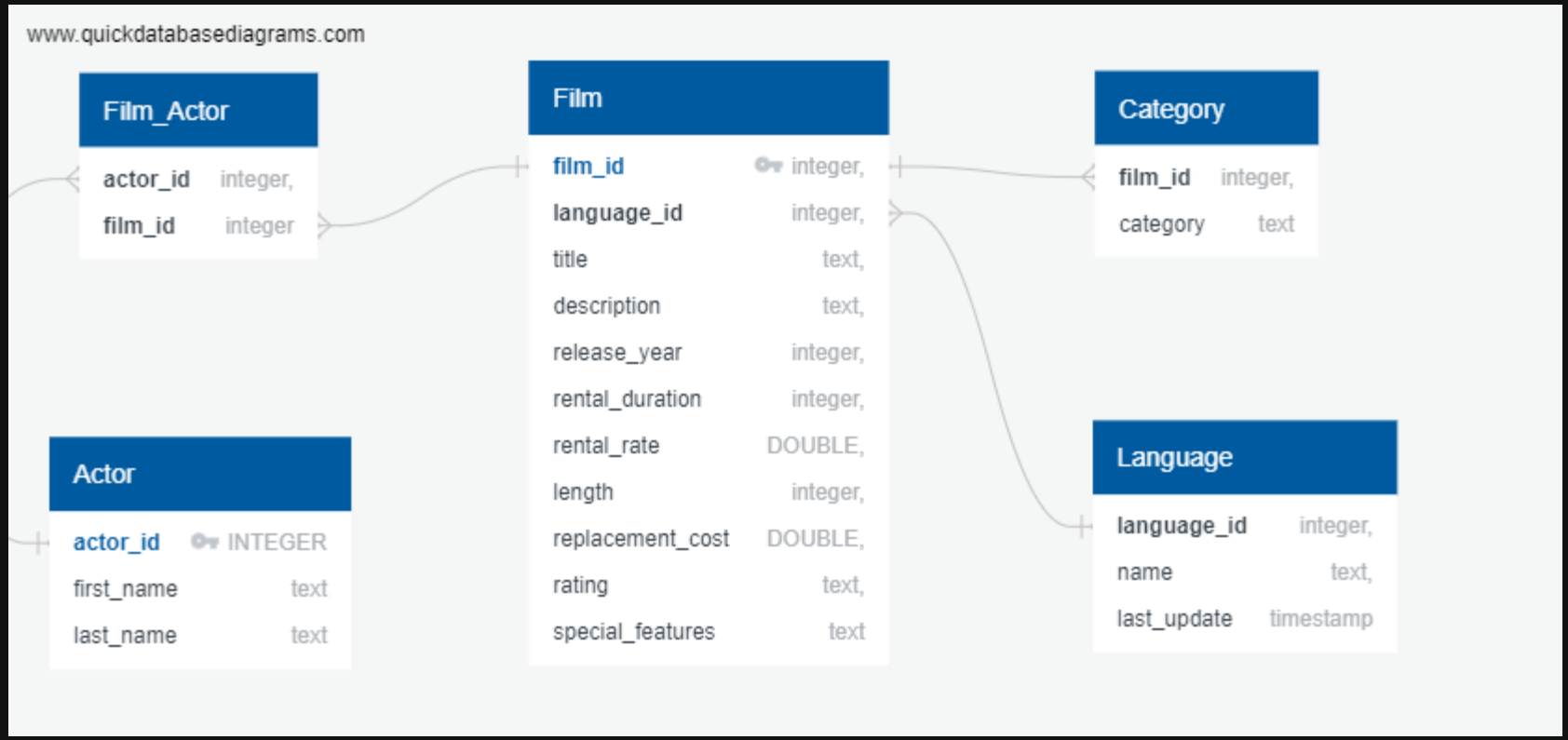
- Lets use filter our aggregation (a `WHERE` clause for `GROUP BY`)
- We need to call the `SUM` function again at `HAVING` because we can't use the alias due to the order of execution

QuEsTiOnS ?

Denormalized table



Normalized Tables



Normalized

Query to get all records

```
SELECT f.*, a.*, c.*,l.*
FROM film f
INNER JOIN film_actor fa ON fa.film_id = f.film_id
INNER JOIN actor a ON fa.actor_id = a.actor_id
INNER JOIN category c ON f.film_id = c.film_id
INNER JOIN language l on f.language_id = l.language_id
```

Unnormalized

Query to get all records

```
SELECT *
FROM film
```

Section 3: CROSS JOIN

```
SELECT f.title as film_title, l.name AS language
FROM film f, language l
```

```
SELECT f.title as film_title, l.name AS language
FROM film f, language l
WHERE f.language_id = l.language_id
```

Notes

- Creates every possible combination for the records in each table
- We can use a `WHERE` clause to make it simulate an `INNER JOIN`
- AVOID USING THIS!!

Section 3: INNER JOIN

```
SELECT f.title, l.name AS language
FROM film AS f
INNER JOIN language l ON f.language_id = l.language_id
```

Notes

- Most common JOIN
- Only joins records with matching values
  - `ON table1.column = table2.column`
- Does not return any null values for missing relationships

Section 3: OUTER JOIN

LEFT JOIN

```
SELECT l.*, f.title
FROM language as l
LEFT JOIN film as f
ON l.language_id = f.language_id
```

Notes

- Left and Right table refers to the order that the tables were called
  - First table (`FROM` clause) will be the left table for example
- `LEFT JOIN` will return all of the records from the left table
- If a record in the left table is missing a relationship in the right table, the missing values will be shown as null
- More commonly used than `RIGHT JOIN`

Section 3: OUTER JOIN

RIGHT JOIN

```
SELECT l.*, f.title
FROM language as l
RIGHT JOIN film as f
ON l.language_id = f.language_id
```

FULL JOIN

```
SELECT l.*, f.title
FROM language as l
FULL JOIN film as f
ON l.language_id = f.language_id
```

Notes

- Left and Right table refers to the order that the tables were called
  - First table (`FROM` clause) will be the left table for example
- Works the exact same way as the `LEFT JOIN` except we will be getting all of the values in the right table and attacting records from the left table
- Not commonly used, a `LEFT JOIN` is usually used by just moving the right table to the left

Section 3: OUTER JOIN

FULL JOIN

```
SELECT l.*, f.title
FROM language as l
FULL JOIN film as f
ON l.language_id = f.language_id
```

Notes

- Takes all of the records from the left and right tables and joins them
- If there are missing relationships in either table, they will be shown as null

Section 3: SELF JOIN

```
SELECT f1.title, f1.length, f1.rental_rate
FROM film f1
INNER JOIN film f2 ON f1.rental_rate = f2.rental_rate
```

Notes

- Used when joining a table to itself
- Useful for looking at relationships within a single table
- Not really effective when the database is normalized

TEST YOUR MIGHT

LEETCODE SQL CHALLENGES