</talentlabs>

CHAPTER 6

Complex SELECT Statements

</talentlabs>

AGENDA

- Subquery
- SQL CASE
- Basic Aggregations
- Advanced Aggregation
  - GROUP BY
  - HAVING

Subqueries

When queries getting complicated

- We may need to use the data of one table to query another table
- We may also need to filter or query a table twice to get the results we want
- Let’s assume we have the below table and columns

Movies

id	title
1	Movie A
2	Movie B
3	Movie C
4	Movie D
5	Movie E

Ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Table: movies

id	title
1	Toy Story
2	Toy Story 2
3	Toy Story 3
4	Star Wars 1
5	Star Wars 2

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Subquery

- From the sample tables, let's say we want to extract the movie rating of Toy Story

```
SELECT rating FROM ratings
WHERE movie_id = (
    SELECT id FROM movies WHERE title='Toy Story'
)
```

- We have a query to movies table for the movie id of "Toy Story" before we query the ratings table
- In this example, the query to movies table is a subquery
- The id data queried (id=1) from movies table is passed to the main query as a WHERE clause condition (movie_id=1)
- The result will be 5.7

Subquery

Table: movies

id	title
1	Toy Story
2	Toy Story 2
3	Toy Story 3
4	Star Wars 1
5	Star Wars 2

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Results

movie_id	rating
1	5.7
2	3.0
3	9.3

- If we need to get the ratings of Toy Story Series (i.e. all three episodes.)

```
SELECT movie_id, rating FROM ratings
WHERE movie_id IN (
    SELECT id
    FROM movies
    WHERE title LIKE 'Toy Story%'
)
```

- We can use IN keyword to pick up multiple result of the subquery

Organizing Subqueries

- Assume we have a years table for each movie_id
- Read the below query and try to tell what it is trying to accomplish.

Table: years

movie_id	year
1	2000
2	2009
3	2013
4	1980
5	1983

```
SELECT rating FROM ratings
WHERE movie_id IN (
    SELECT id FROM movies
    WHERE
        id IN (
            SELECT movie_id FROM years
            WHERE year > 2010
        )
        AND title LIKE 'Toy Story%')
)
```

- The query is getting difficult to read as the subqueries are nested together

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.

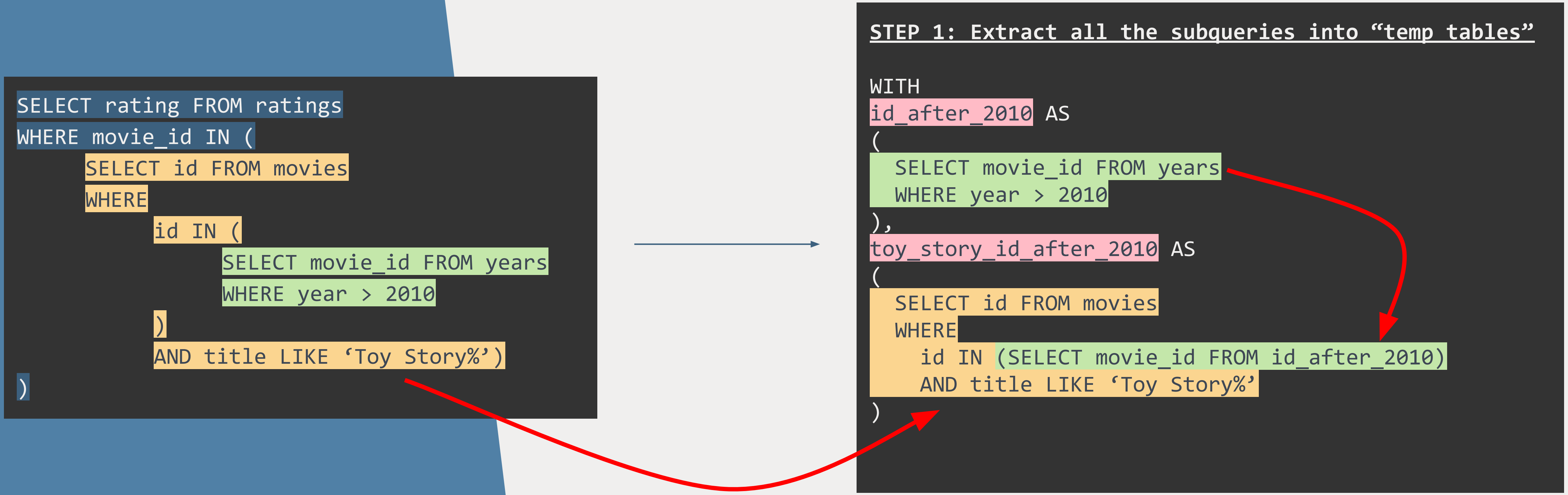
```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

```
STEP 1: Extract all the subqueries into “temp tables”

WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
```

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.



```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

```
STEP 1: Extract all the subqueries into “temp tables”

WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
toy_story_id_after_2010 AS
(
  SELECT id FROM movies
  WHERE
    id IN (SELECT movie_id FROM id_after_2010)
    AND title LIKE ‘Toy Story%’
)
```

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.

```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

```
STEP 2: Build the outermost query

WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
toy_story_id_after_2010 AS
(
  SELECT id FROM movies
  WHERE
    id IN (SELECT movie_id FROM mv_id_after_2010)
    AND title LIKE ‘Toy Story%’
)
SELECT rating FROM ratings
WHERE
  movie_id IN (SELECT id FROM toy_story_id_after_2010)
```

SQL CASE

</talentlabs>

Table: movies

id	title	year
1	Toy Story	2000
2	Toy Story 2	2009
3	Toy Story 3	2013
4	Star Wars 1	1980
5	Star Wars 2	1983

Query Result

title	movie_period
Toy Story	Released after 2000
Toy Story 2	Released after 2000
Toy Story 3	Released after 2000
Star Wars 1	Released before 2000
Star Wars 2	Released before 2000

SQL CASE

- CASE keyword can apply logic to manipulate values returned from a query
- It works like an IF-THEN-ELSE conditional statement of other programming languages

```
SELECT
    title,
    CASE
        WHEN year > 1999 THEN 'Released after 2000'
        ELSE 'Release before 2000'
    END AS movie_period
FROM movies
```

- Note 1: a CASE statement can include multiple conditions i.e. multiple WHEN-THEN.
- Note 2: ELSE clause is optional

Table: movies

id	title	year
1	Toy Story	2000
2	Toy Story 2	2009
3	Toy Story 3	2013
4	Star Wars 1	1980
5	Star Wars 2	1983

Query Result

title	movie_period
Toy Story	Released in 2000
Toy Story 2	Released in 2009
Toy Story 3	Released in 2013
Star Wars 1	Released in 1980
Star Wars 2	Released in 1983

SQL CASE

- We can also do value matching instead of just condition matching
- In this example, instead of condition matching (e.g. year>1999), we performs value matching on the "year" column

```
SELECT
    title,
    CASE year
        WHEN 2000 THEN 'Released in 2000'
        WHEN 2009 THEN 'Released in 2009'
        WHEN 2013 THEN 'Released in 2013'
        WHEN 1980 THEN 'Released in 1980'
        WHEN 1983 THEN 'Released in 1983'
    END AS movie_period
FROM movies
```

Basic Aggregation

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Basic Aggregations

- Sometimes, we might want to do some statistical analysis on the data (e.g. calculating sum, averages, maximum and minimum)
- This would help us in getting more insights about the data
- Say we want to know the average release year of the table. We can perform the following query using the AVG function

```
SELECT AVG(rating) FROM ratings
```

- The result would be 5.34

Basic Aggregations

Aggregation Function	Function
COUNT	counts how many rows are in a particular column
SUM	adds together all the values in a particular column
MIN and MAX	return the lowest and highest values in a particular column, respectively
AVG	calculates the average of a group of selected values

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

COUNT

- Used to count number of records in the table

```
SELECT COUNT(*) FROM ratings
```

- Returns number of rows in the table, i.e. 5

Advanced Aggregation - GROUP BY

</talentlabs>

GROUP BY

Let's consider the below data table:

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	6.2
5	E	2009	7.2
6	F	2009	9

How do we write **one** query to obtain the average movie rating of each year?

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

GROUP BY

- From what we've learnt, we can calculate the average of the rating by aggregation.
- Using WHERE clause, the rating data can be filtered by year

```
SELECT AVG(rating)
FROM sample_movies
WHERE year=1994
```

AVG(rating)
7.13

```
SELECT AVG(rating)
FROM sample_movies
WHERE year=2009
```

AVG(rating)
8.13

- However, it takes 2 queries instead of 1 to obtain the average rating. What if the table contains even more years?

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Results

year	avg_rating
1994	7.13
2009	8.13

GROUP BY

GROUP BY clause allows grouping of data by one or more fields and then perform aggregation by each grouping value

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
GROUP BY year
```

Advanced Aggregation - HAVING

</talentlabs>

year	avg_rating
1994	7.13
2009	8.13

Filtering Aggregated Values

Scenario: Let's say we need to get the year and average rating which the average rating for the year is at least 8.

We cannot directly use WHERE clause to filter the aggregation results

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
WHERE AVG(rating)>=8 -- causes error
GROUP BY year
```

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Subquery Results

year	avg_rating
1994	7.13
2009	8.13

Results

year	avg_rating
2009	8.13

Solution 1 - Using Subquery

We can use WHERE clause to filter aggregation results, but will need to leverage subquery to store the aggregation results first

```
SELECT
    year,
    avg_rating
FROM
(
    SELECT
        year,
        AVG(rating) AS avg_rating
    FROM sample_movies
    GROUP BY year
)
WHERE avg_rating >= 8 -- this works
```

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Results

year	avg_rating
2009	8.13

Solution 2 - HAVING keyword

To simplify the query, we can use HAVING clause to filter aggregation result while keeping the query simple

Solution 2 with HAVING keyword (Much longer query)

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
GROUP BY year
HAVING AVG(rating) >= 8
```

Solution 1 with Subquery (Much longer query)

```
SELECT
    year,
    avg_rating
FROM
(
    SELECT
        year,
        AVG(rating) AS avg_rating
    FROM sample_movies
    GROUP BY year
)
WHERE avg_rating>=8
```

Summary

- We've learnt subqueries and WITH keyword for subquery organization
- We've learnt CASE statement for working with conditions for data values
- We've learnt aggregations, followed by GROUP BY and HAVING for data grouping and aggregation filtering