SQL Queries – optional part

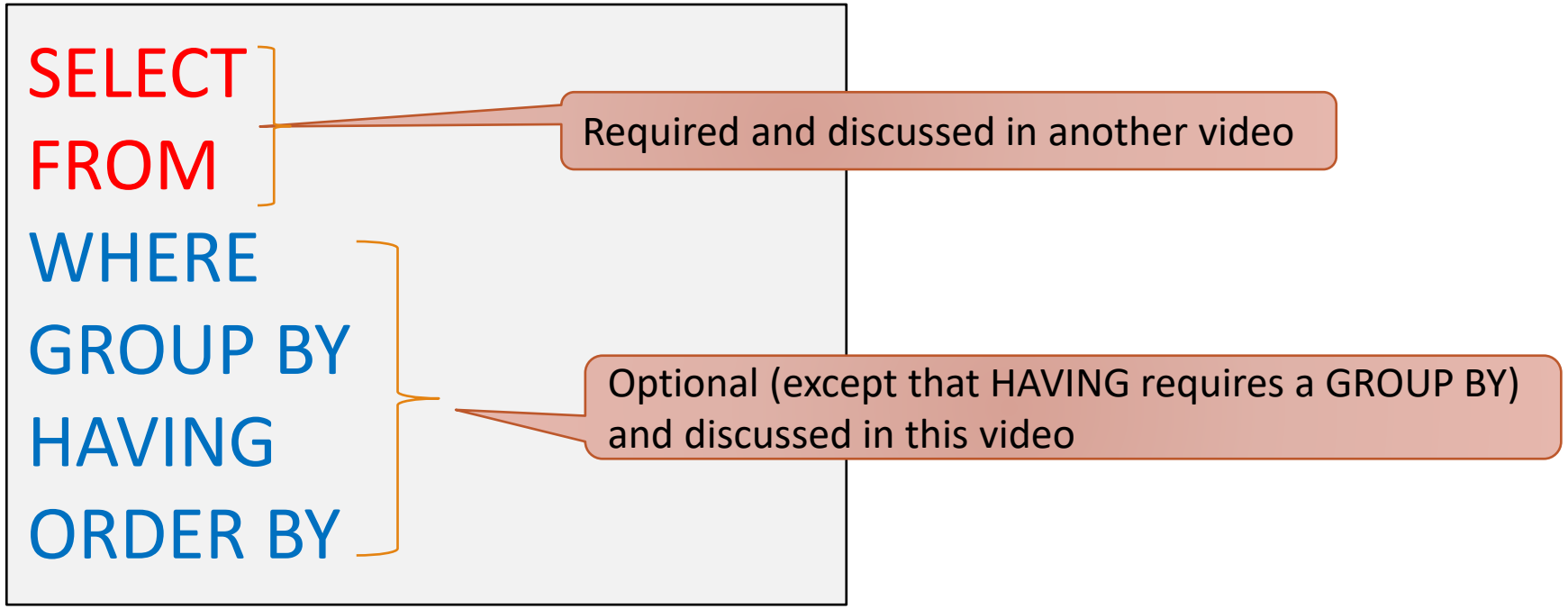
NOTE: THIS VIDEO IS REQUIRED FOR THE COURSE. THE NAME COMES FROM THAT IT IS ABOUT THE PART OF QUERIES YOU DO NOT NEED TO HAVE A VALID QUERY

Overview of this video

A run through of the optional part of SQL queries

SQL Queries

Queries in SQL have the following form:



WHERE

Already mostly covered in video on SQL DML except queries

- Because, it was also used in DELETE and UPDATE

The exception to that was that some of the things one can write in WHERE clauses (incl. WHERE clauses in DELETE and UPDATE) involves sub-queries

Since we will later be using such, it is covered next

WHERE: IN

Similar query with IN

It was previously mentioned that IN could be used in WHERE

The example used was:

```
DELETE FROM Students
WHERE name IN ('John','Sebastian');
```

or

```
SELECT *
FROM Students
WHERE name IN ('John','Sebastian');
```

However, instead of providing a list, one can also do a sub-query (here using a similar University database as in the example above):

```
DELETE FROM Students
WHERE name IN (SELECT name
               FROM Lecturers);
```

or

```
SELECT *
FROM Students
WHERE name IN (SELECT name
               FROM Lecturers);
```

WHERE: EXISTS

EXISTS is a generalization of IN

E.g. continuing with the example University database mentioned on the last slide, as we saw IN can be used to find the students with the same first name as a lecturer, but not (nicely) find the students with the same first AND last name as lecturer

In essence, with EXISTS you write a sub-query and if it returns something, the current row will be kept otherwise thrown out. You can use attributes from the current row in the sub-query

EXISTS can handle such issues, e.g.

It is common to use the constant 1 when we just want to have an output or not

```
SELECT *
FROM Students
WHERE EXISTS    (SELECT 1
                 FROM Lecturers
                 WHERE Students.name=Lecturers.name AND Students.last_name=Lecturers.last_name);
```

WHERE: Semi-join

Most types of joins are done in the FROM clause

- See the video on the required part of SQL queries

There is 1 exception though, which is semi-joins, which are done in WHERE clauses

A (left) semi-join between two tables A and B, finds the subset of rows in A that would be used to form some row of the natural join of A and B

- A right semi-join would find the subset of B instead

These are done using EXISTS (or IN, if there is only one shared attribute)

The example we saw of EXISTS was really a left semi-join of Students with Lecturers

- We only kept the rows from students that had a matching first and last name as a lecturer, which were their common attributes

WHERE: Semi-join

(This was the example for NATURAL JOIN)

Say that we want to left semi-join **Employees** to **Transactions**

```
SELECT *
FROM Employees E
WHERE EXISTS    (SELECT 1
                 FROM Transactions T
                 WHERE E.e_id=T.e_id);
```

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
1995-05-09	William	Taylor	3

Observe: Only 1 Anne row in output
even though two transactions were done by her
(the query makes it clear why it happens, but it is part of the semi-join definition)

GROUP BY

Instead of first discussing how it is used/what it does, I will start by what GROUP BY is used for

Say I want to know how many transactions Anne (i.e. e_id=1) was part of:

```
SELECT e_id, COUNT(t_id)
FROM Transactions
WHERE e_id=1;
```

e_id	COUNT(t_id)
1	2
1	2

If you want only one row, you could add DISTINCT after SELECT

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

GROUP BY cont.

Say I want to do that for each employee instead of just Anne

To do that, you just replace WHERE with GROUP BY (and remove =1)

```
SELECT e_id, COUNT(t_id)
FROM Transactions
GROUP BY e_id;
```

The meaning of COUNT is now depending on GROUP BY

e_id	COUNT(t_
1	2
3	1

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

GROUP BY: A complication

If we include GROUP BY, then we can only include in SELECT attributes we GROUP BY or aggregates

E.g. say we wanted the first name of the employees as well as their employee id and the number of their transactions

- (Note that first_name is unique for the employees in the database)

We would THINK we would do it like this:

```
SELECT first_name, e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY e_id;
```

This does NOT work though, since first_name is not in GROUP BY

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

GROUP BY: A solution to the complication

This was wrong:

```
SELECT first_name, e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY e_id;
```

This is a right way to do it:

```
SELECT first_name, e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY first_name, e_id;
```

first_name	e_id	COUNT(t_id)
Anne	1	2
William	3	1

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

GROUP BY: A solution to the complication

This was wrong:

```
SELECT first_name, e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY e_id;
```

This is another way to do it:

```
SELECT MIN(first_name), e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY e_id;
```

MIN(first_name)	e_id	COUNT(t_id)
Anne	1	2
William	3	1

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

GROUP BY: Intution

Intuitively speaking, GROUP BY works like this:

Do the previous part of the query until GROUP BY (except ignoring the part in SELECT)

Split the resulting table into sub-tables

- 1 sub-table for each value of the variables in GROUP BY
- I.e. if you had say first_name and e_id like in the example, you would have a sub-table for each pair of first_name and e_id. This means, e.g. that if two employees were named Anne, you would have two sub-tables, one for each Anne, defined by their e_id

GROUP BY: Intution cont.

```
…
FROM Employees NATURAL JOIN Transactions
GROUP BY first_name, e_id;
```

first_name	e_id	Sub-table
Anne	1	birthday, family_name, c_id, t_id 1990-11-10, Smith, 3, 1 1990-11-10, Smith, 6, 2
William	3	birthday, family_name, c_id, t_id 1995-05-09, Taylor, 19, 3

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

HAVING

HAVING is easy! It is just WHERE, but done after GROUP BY

- Meaning that it does not affect GROUP BY and can use aggregates on the sub-tables in its expressions

A simple example:

```
SELECT first_name, e_id, COUNT(t_id)
FROM Employees NATURAL JOIN Transactions
GROUP BY first_name, e_id
HAVING MIN(c_id)<5;
```

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

Transactions

t_id	c_id	e_id
1	3	1
2	6	1
3	19	3

first_name	e_id	COUNT(t_id)
Anne	1	2

ORDER BY

ORDER BY defines how the output is sorted

```
SELECT first_name
FROM Employees
ORDER BY family_name;
```

Is not actually in the output…

Employees

birthday	first_name	family_name	e_id
1990-11-10	Anne	Smith	1
2000-02-05	David	Jones	2
1995-05-09	William	Taylor	3

first_name
David
Anne
William

Can also have more than one attribute in ORDER BY, e.g. ORDER BY family_name, first_name
Then, among those with the same family_name, the ones with the first first_name would come first
(i.e. multiple attributes makes it sorted lexicographically)

Finally, it defaults to ascending order. You can do descending by writing DESC. Like, if you want to sort by family_name ascending but first_name descending, you write ORDER BY family_name, first_name DESC

family_name descending but first_name ascending would be: ORDER BY family_name DESC, first_name