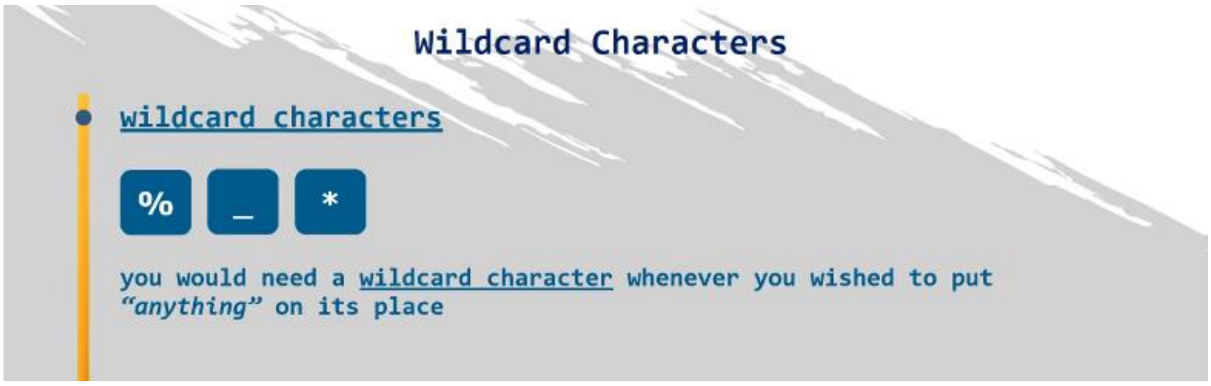
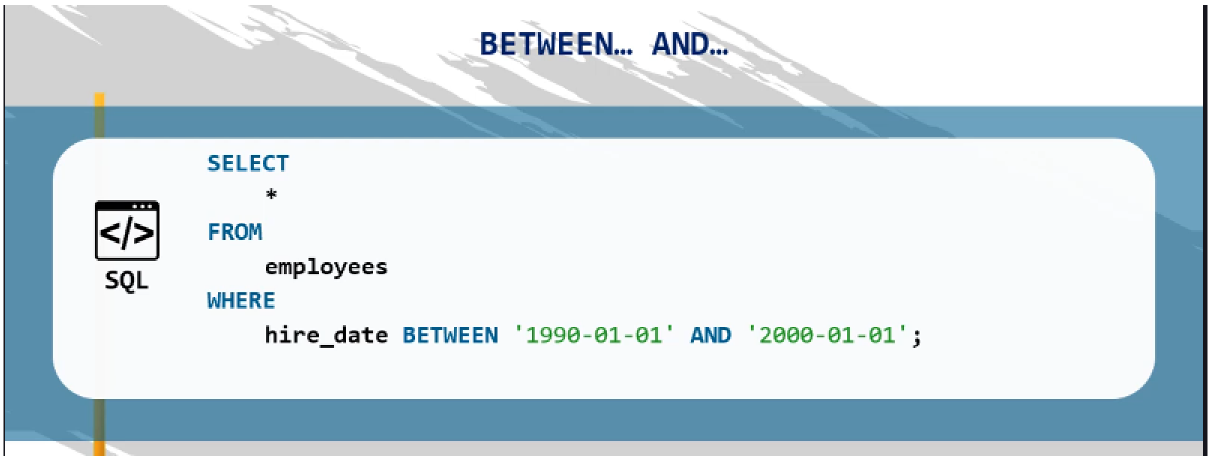


Wildcard Characters

- wildcard characters

% _ *

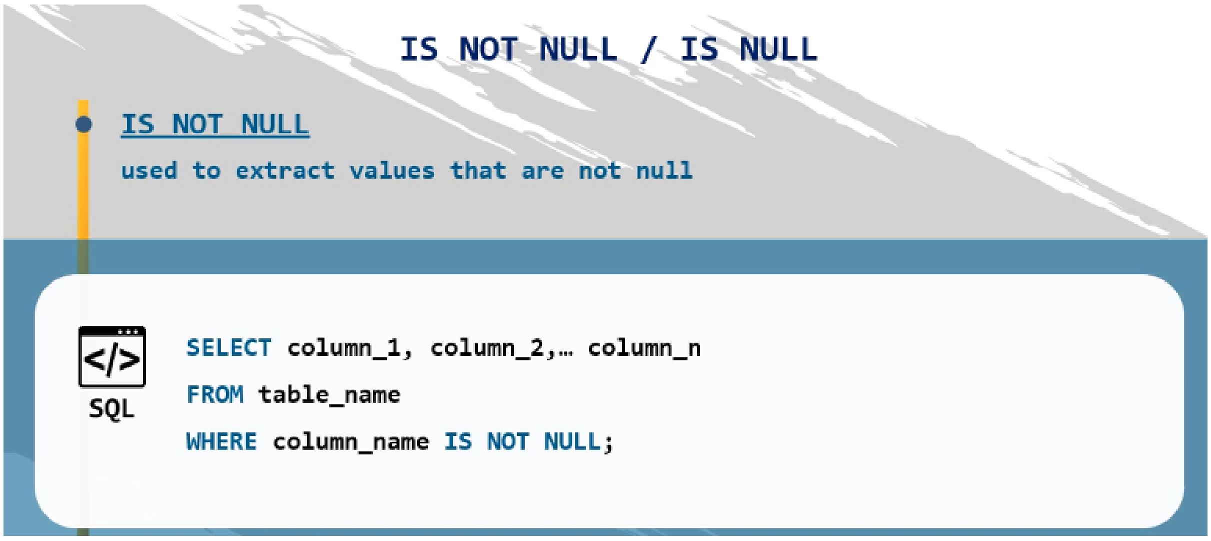
you would need a wildcard character whenever you wished to put *"anything"* on its place



BETWEEN… AND…

```sql
SELECT
    *
FROM
    employees
WHERE
    hire_date BETWEEN '1990-01-01' AND '2000-01-01';
```

The BETWEEN operator helps us designate the interval to which a given value belongs. That's why it is always used in combination with the AND operator. If you want to obtain a list of the people who were hired between the 1st of January, 1990 and the 1st of January, 2000, use the same select statement structure and indicate these higher dates in the WHERE clause using the following syntax. WHERE, higher date, BETWEEN 1990 01 01 and 2000 01 01.



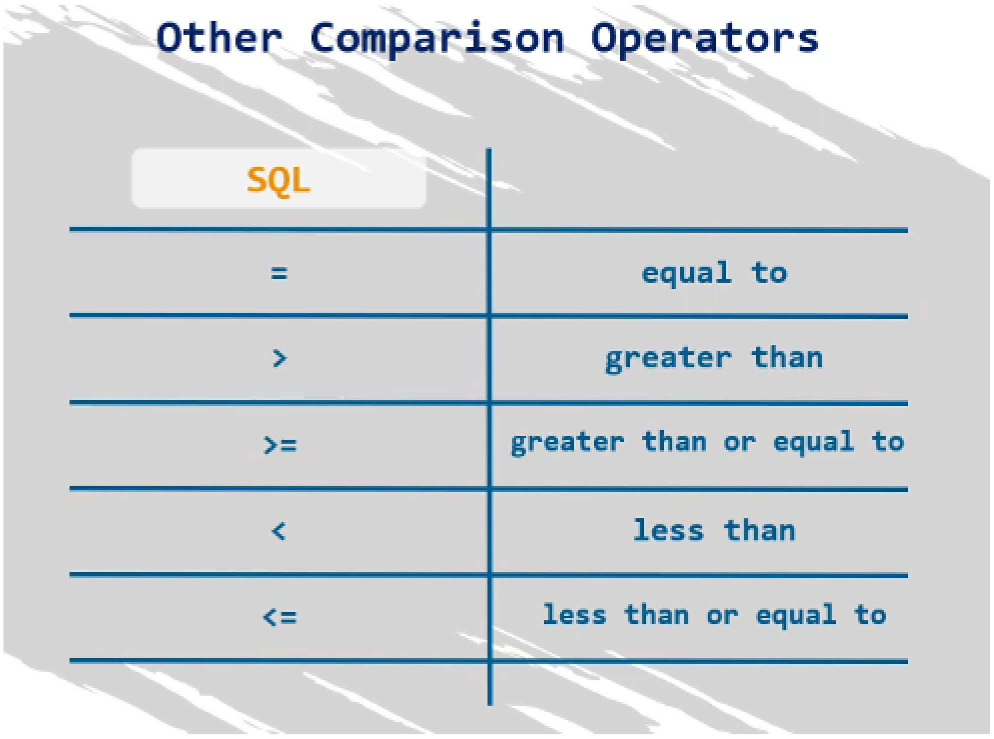
IS NOT NULL / IS NULL

- IS NOT NULL

used to extract values that are not null

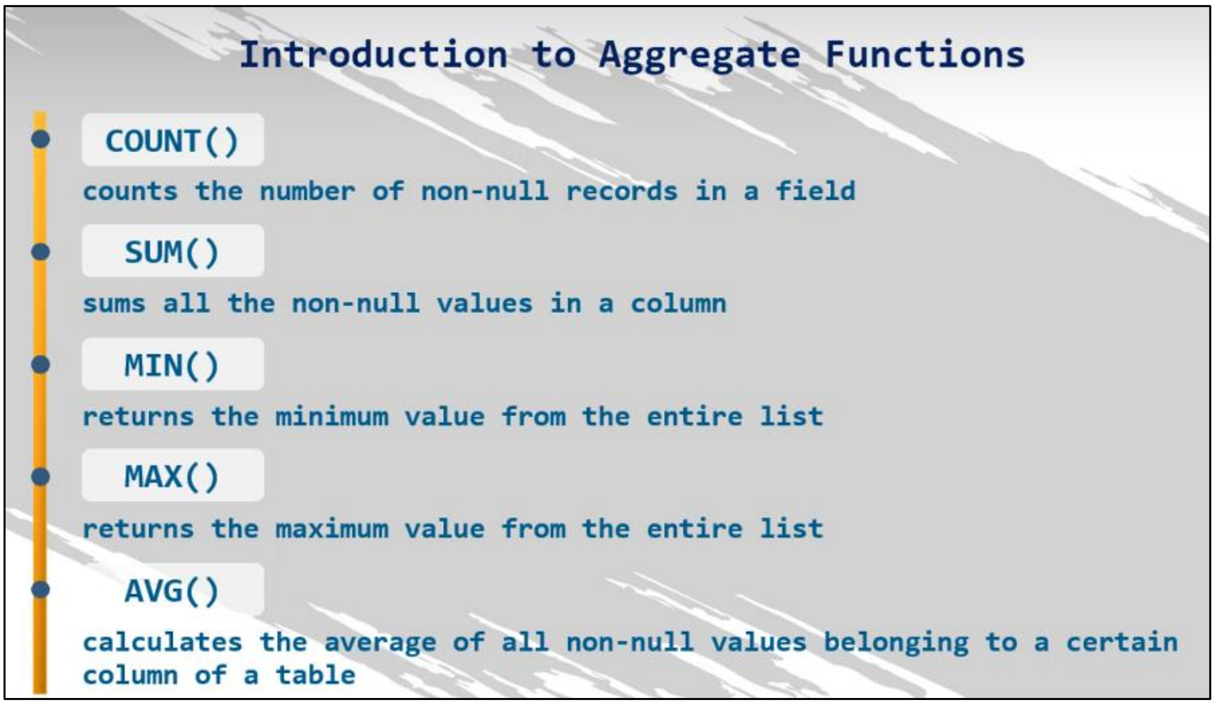
```sql
SELECT column_1, column_2,… column_n
FROM table_name
WHERE column_name IS NOT NULL;
```

Next on our agenda is the IS NOT NULL operator. As it's name suggests, it will be used to extract values that are not NULL. The syntax is intuitive. Select column names from a table where a certain column is NOT NULL.



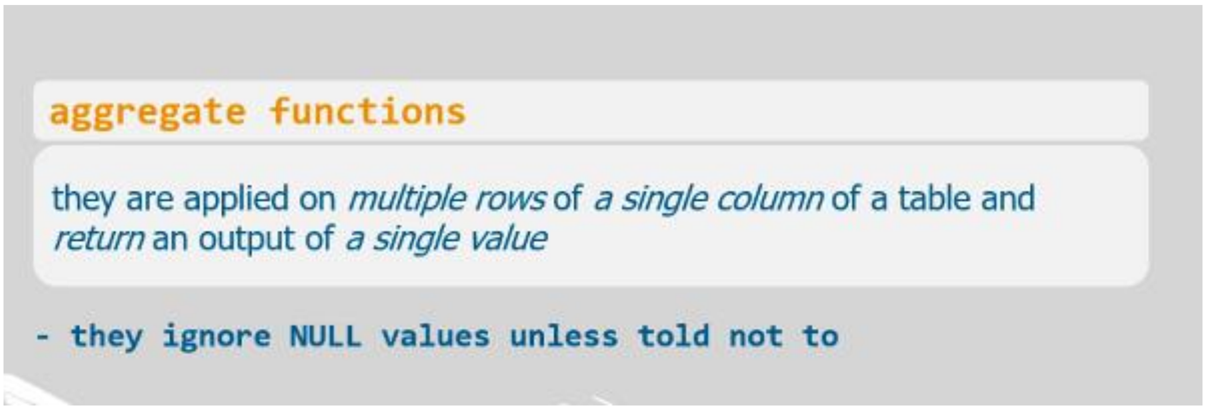
Other Comparison Operators

SQL	
=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to



Introduction to Aggregate Functions

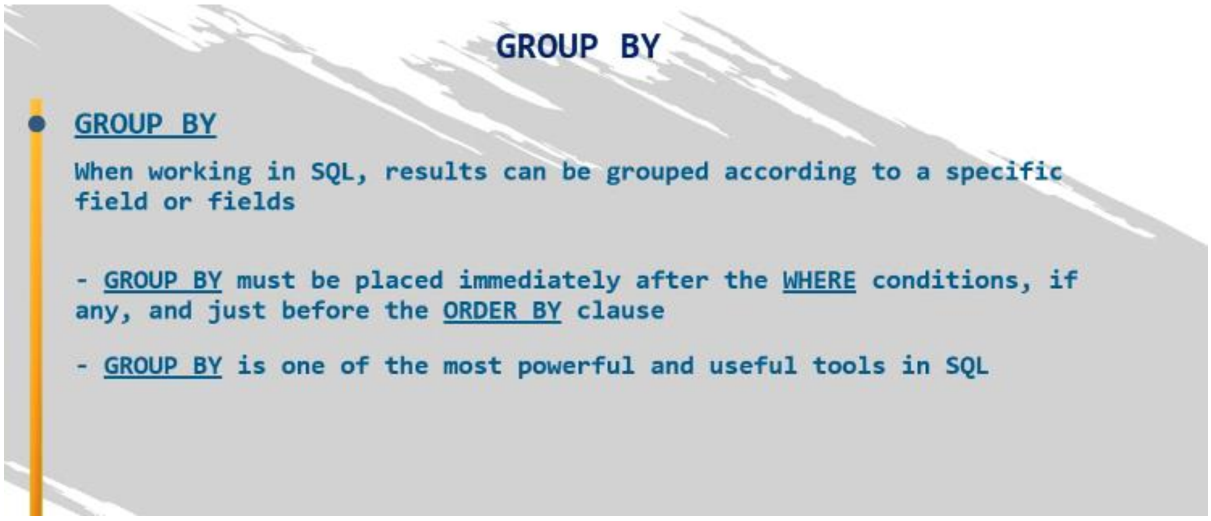
- `COUNT()`
  counts the number of non-null records in a field
- `SUM()`
  sums all the non-null values in a column
- `MIN()`
  returns the minimum value from the entire list
- `MAX()`
  returns the maximum value from the entire list
- `AVG()`
  calculates the average of all non-null values belonging to a certain column of a table


aggregate functions

they are applied on *multiple rows* of *a single column* of a table and *return* an output of *a single value*

- they ignore NULL values unless told not to

please remember an important feature of aggregate functions. They ignore null values unless told not to. This means if there were any null values in the employee number or the first name columns, count would not have counted them and would not have added them to the total.

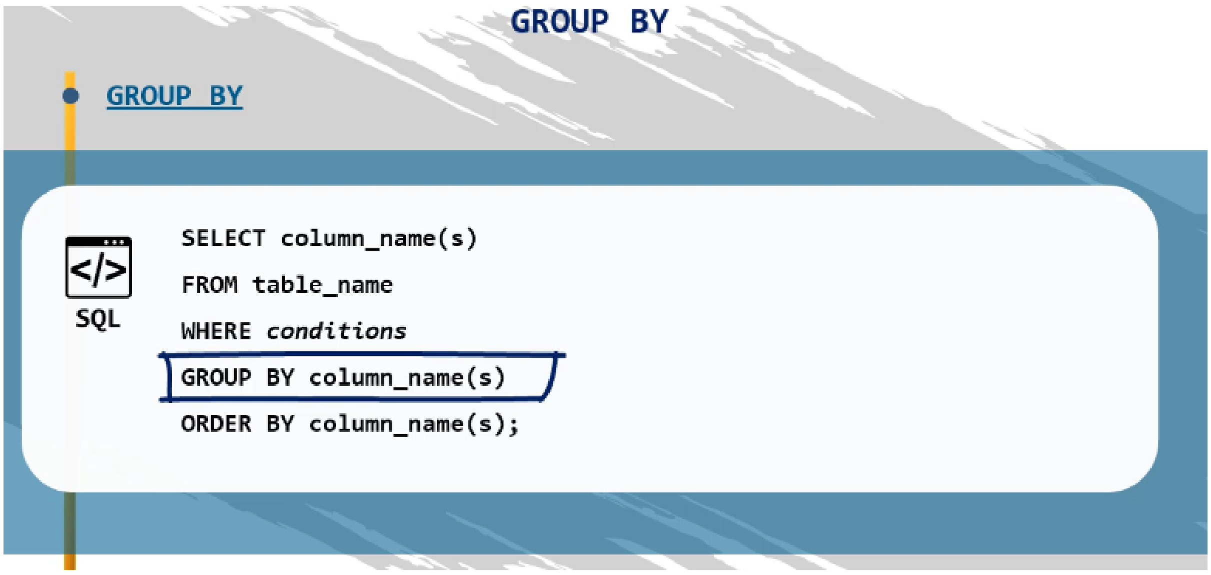

GROUP BY

GROUP BY

When working in SQL, results can be grouped according to a specific field or fields

- GROUP BY must be placed immediately after the WHERE conditions, if any, and just before the ORDER BY clause

- GROUP BY is one of the most powerful and useful tools in SQL

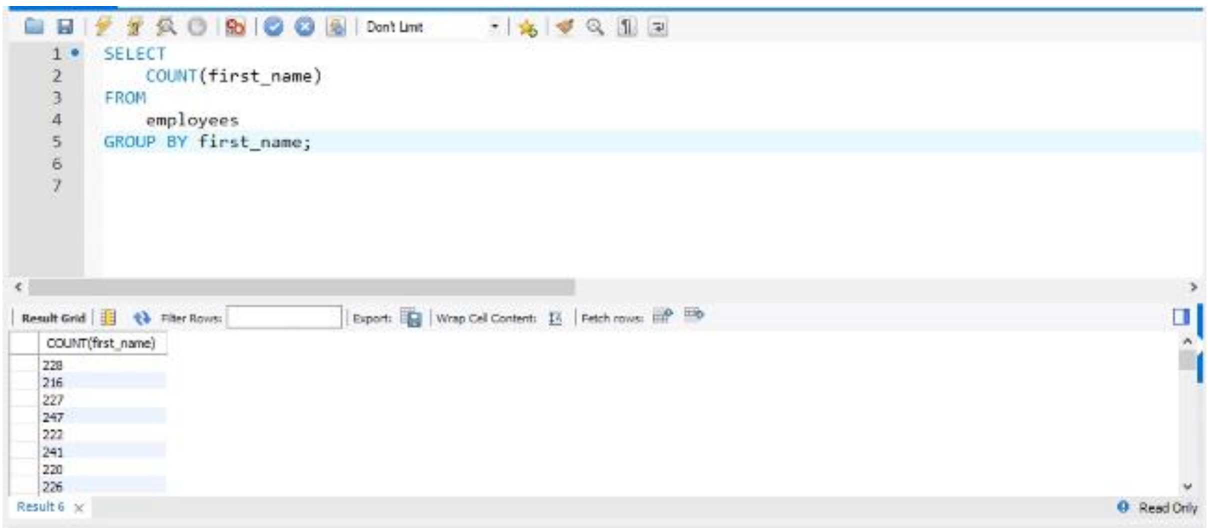

GROUP BY

GROUP BY

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
ORDER BY column_name(s);
```

The syntax to comply with is the same old select column names from a given table, where some condition or conditions have been satisfied. Group by column name or column names and then finish with order by and the same or different column names. So for the moment, please remember the group by clause is located just above the order by clause.

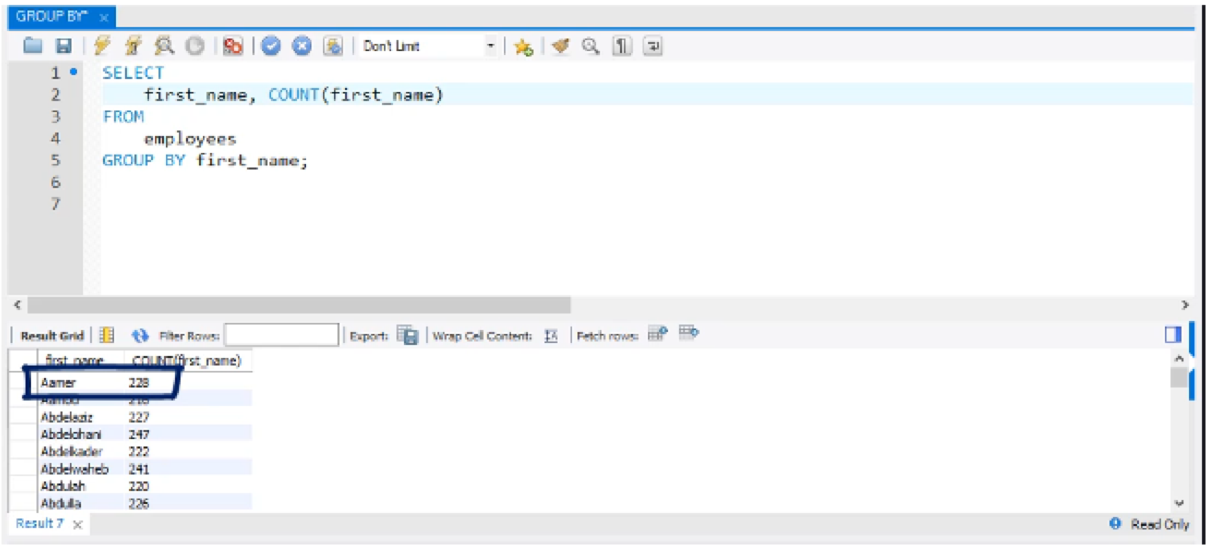
An example:



```sql
SELECT
    COUNT(first_name)
FROM
    employees
GROUP BY first_name;
```

COUNT(first_name)
228
216
227
247
222
241
220
226

In most cases, when you need an aggregate function, you must add a group by clause in your query too. Here is what I mean. Assume you need a list composed of two fields. The first must contain a distinct first name of the employee and the second the number of times this name is encountered in our database. Looking for a single total value must ring a bell straight away. If we type select count first name from employees we will get the total value of records in this table. Then if we add group by first name, we'll split the result return from the select statement into groups. Check, yes correct that's true. In this column, we see the number of times each name is encountered but we don't see the names these values refer to right. Here is a rule of thumb professionals strictly comply <u>with always include the field you have group to results by</u> in the select statement. Let's do that:

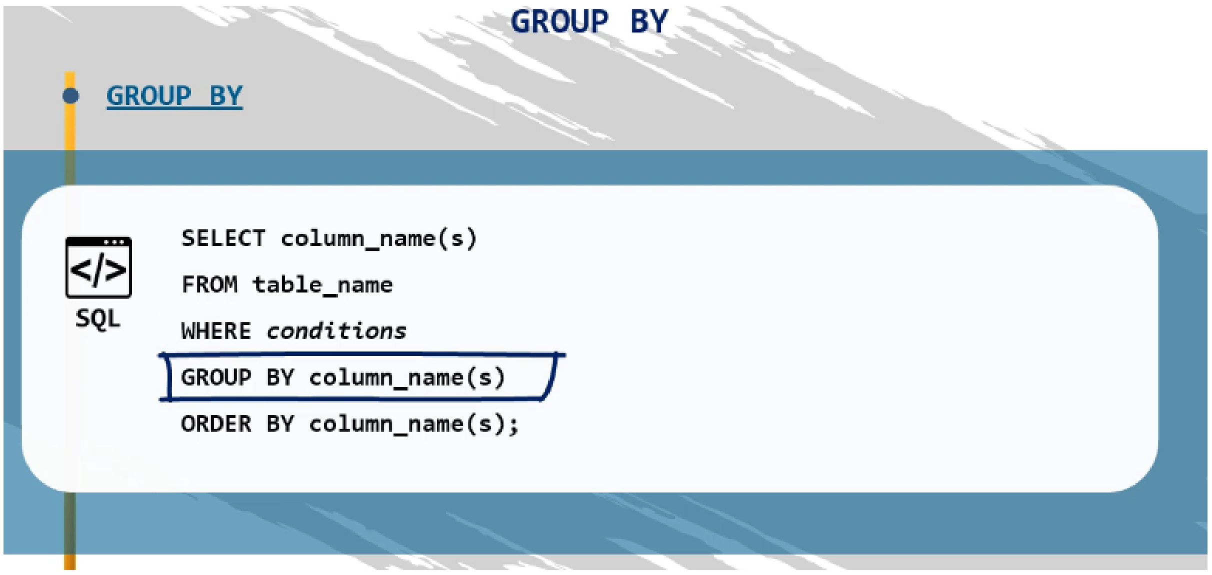


```sql
SELECT
    first_name, COUNT(first_name)
FROM
    employees
GROUP BY first_name;
```

first_name	COUNT(first_name)
Aamer	228
[illegible]	[illegible]
Abdelaziz	227
Abdelghani	247
Abdelkader	222
Abdelwaheb	241
Abdulah	220
Abdulla	226

Almer can be seen 228 times Ahmad 216 and so on. Amazing, this rule is crucial because in workbench as you just saw, the query would run properly if you don't include the group by field and the select statement. But this will not be valid in some other databases. There it will be impossible to execute the query if written without the group by column in the select statement. So please stick to this simple rule. It also improves the organization and readability of your output.

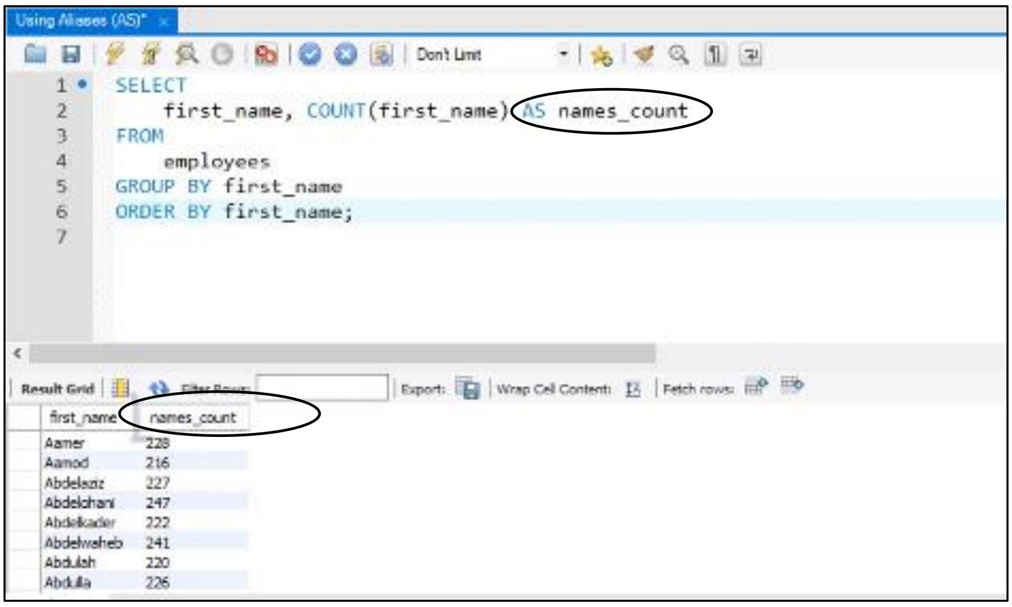
This last piece of information was an important addition to the content of this lecture. Not all blocks of code are mandatory, but you must get used to the order in which you state these blocks in the query. Remember the following logical flow. Select something from a certain table where certain conditions are met. Group the results by a column and possibly order them in a certain direction:



GROUP BY

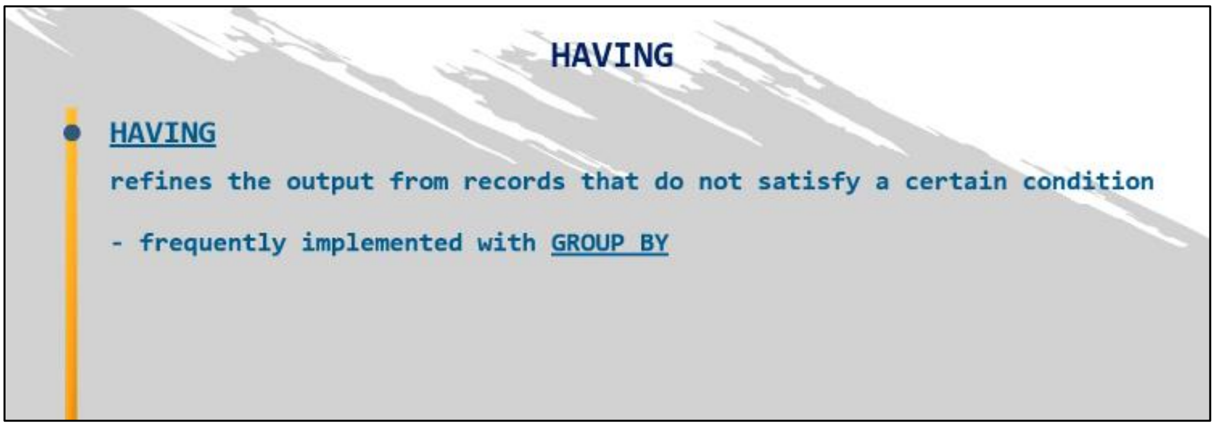
GROUP BY

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
ORDER BY column_name(s);
```



```
SELECT
    first_name, COUNT(first_name) AS names_count
FROM
    employees
GROUP BY first_name
ORDER BY first_name;
```

first_name	names_count
Aamer	228
Aamod	216
Abdelaziz	227
Abdelghani	247
Abdelkader	222
Abdelwaheb	241
Abdulah	220
Abdulla	226

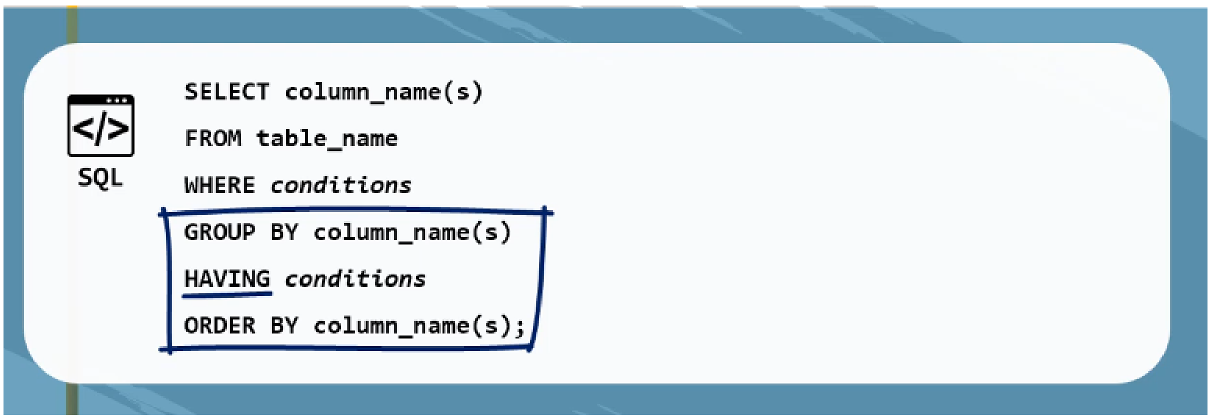


HAVING

HAVING

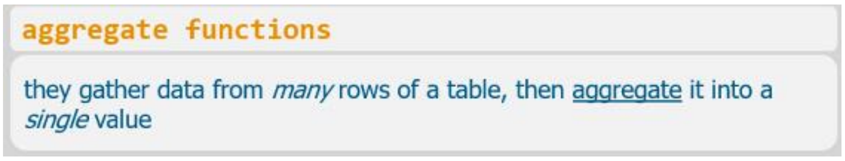
refines the output from records that do not satisfy a certain condition

- frequently implemented with GROUP BY



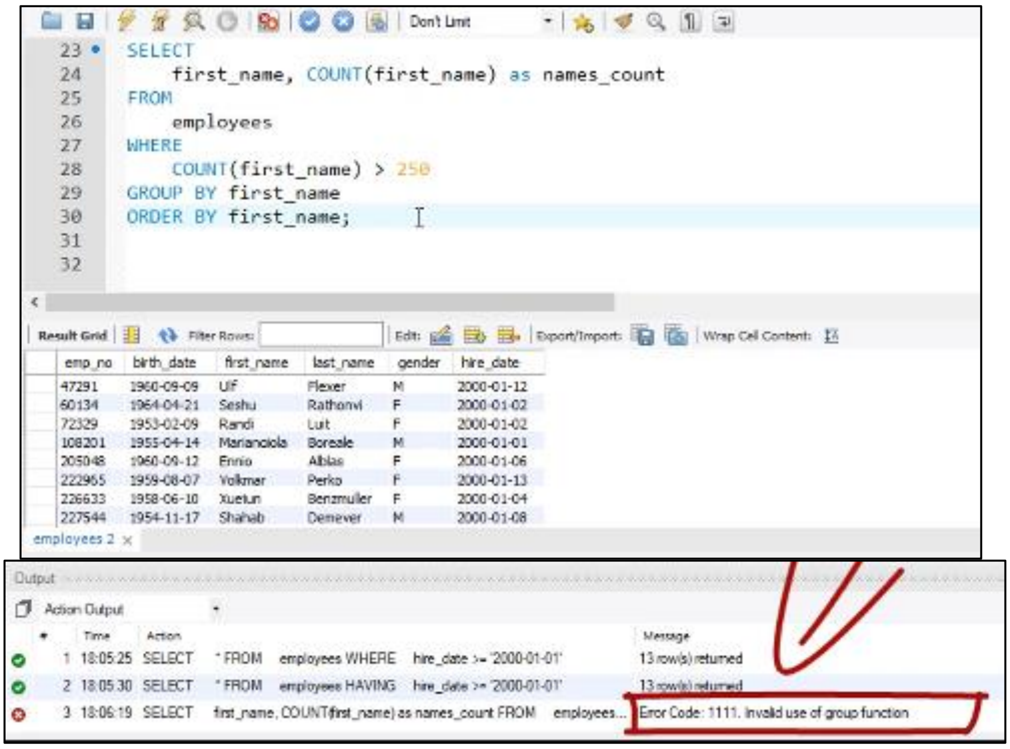
```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

‘Having’ is a clause frequently implemented with group by because it refines the output from records that do not satisfy a certain condition. why does the having clause exist? Internalising the corresponding syntax will help us explain the difference between the two keywords. So let's begin there. Having needs to be inserted between the group by and order by clauses. The difference between WHERE and HAVING is; after HAVING, you can have a condition with an aggregate function, while WHERE cannot use aggregate functions within its conditions. An aggregate function is a type of function that performs a calculation on a set of values and returns a single value e.g. COUNT(), MIN(), MAX(), SUM() and AVG().



aggregate functions

they gather from *many* rows of a table, then aggregate it into a *single* value



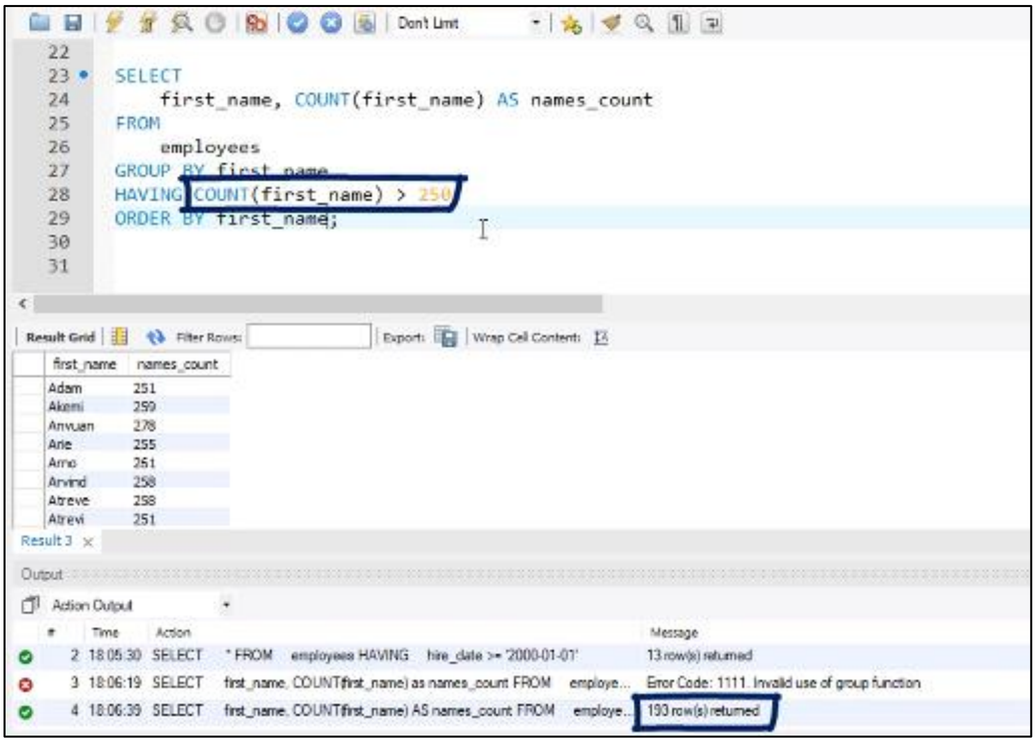
```
SELECT
    first_name, COUNT(first_name) as names_count
FROM
    employees
WHERE
    COUNT(first_name) > 250
GROUP BY first_name
ORDER BY first_name;
```

emp_no	birth_date	first_name	last_name	gender	hire_date
47291	1960-09-09	Ulf	Flexer	M	2000-01-12
60134	1964-04-21	Seshu	Rathonyi	F	2000-01-02
72329	1953-02-09	Randi	Luit	F	2000-01-02
108201	1955-04-14	Mariangiola	Boreale	M	2000-01-01
205048	1960-09-12	Ennio	Alblas	F	2000-01-06
222965	1959-08-07	Volkmar	Perko	F	2000-01-13
226633	1958-06-10	Xuejun	Benzmuller	F	2000-01-04
227544	1954-11-17	Shahab	Demeyer	M	2000-01-08

```
Output
Action Output
  #  Time      Action                                                          Message
  1  18:05:25  SELECT * FROM employees WHERE hire_date >= '2000-01-01'          13 row(s) returned
  2  18:05:30  SELECT * FROM employees HAVING hire_date >= '2000-01-01'         13 row(s) returned
  3  18:06:19  SELECT first_name, COUNT(first_name) as names_count FROM employees...   Error Code: 1111. Invalid use of group function
```

^ Assume you want to extract a list with all first names that appear more than 250 times in the employee's table. If you try to set this condition in the where clause, workbench wouldn't indicate there's a mistake in your code because this is the correct syntax. You will be shown an error message when you try to execute the query and it will be a very eloquent one. Invalid use of group function.

HOWEVER, IF WE CHANGE THE KEYWORD TO 'HAVING' and add the line of code in the right place. Just after the group by statement. <u>Now rerun the query:</u>



```sql
SELECT
    first_name, COUNT(first_name) AS names_count
FROM
    employees
GROUP BY first_name
HAVING COUNT(first_name) > 250
ORDER BY first_name;
```

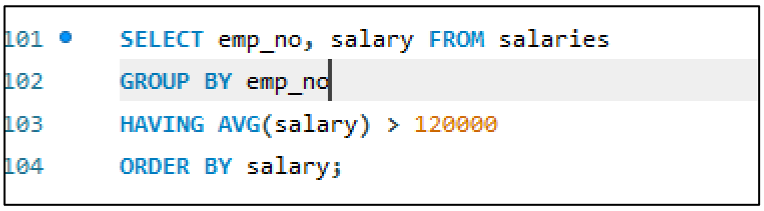
first_name	names_count
Adam	251
Akemi	259
Anvuan	278
Arie	255
Arno	261
Arvind	258
Atreve	258
Atrevi	251

```
#  Time      Action                                                          Message
2  18:05:30  SELECT * FROM employees HAVING hire_date >= '2000-01-01'        13 row(s) returned
3  18:06:19  SELECT first_name, COUNT(first_name) as names_count FROM employe...   Error Code: 1111. Invalid use of group function
4  18:06:39  SELECT first_name, COUNT(first_name) AS names_count FROM employe...   193 row(s) returned
```

Anytime an aggregate function is required for the solution of your task, we use HAVING. In the problem we just solved, extract all first names that appear more than 250 times in the employee's table. So we must first spot the phrase 250 times. <u>It leads to counting</u>. COUNT() is an aggregate function.

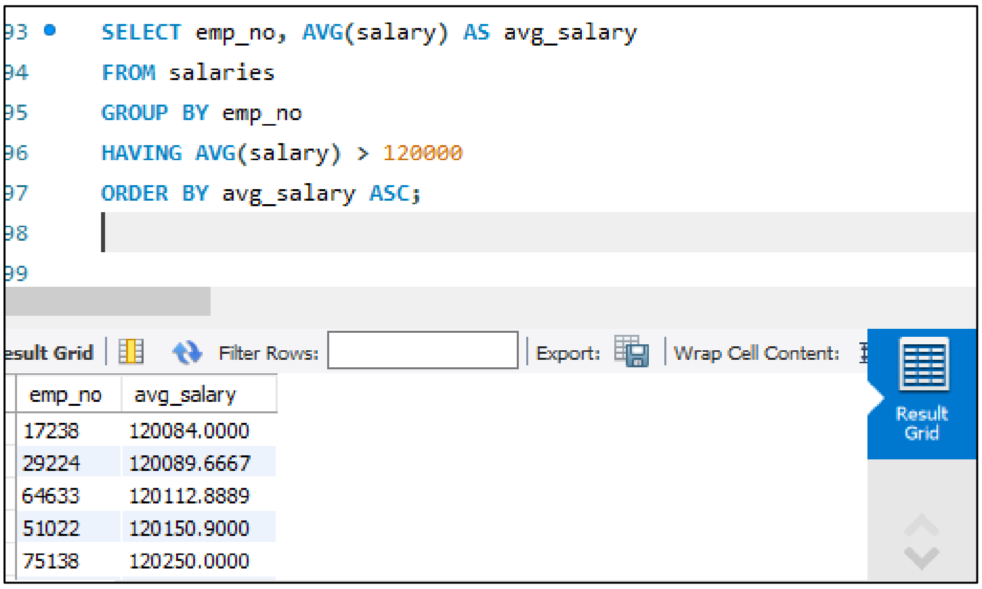
QUESTION: Select all employees whose average salary is higher than $120,000 per annum.

Hint: You should obtain 101 records.



```sql
SELECT emp_no, salary FROM salaries
GROUP BY emp_no
HAVING AVG(salary) > 120000
ORDER BY salary;
```

^This didn't work at all. But this did:

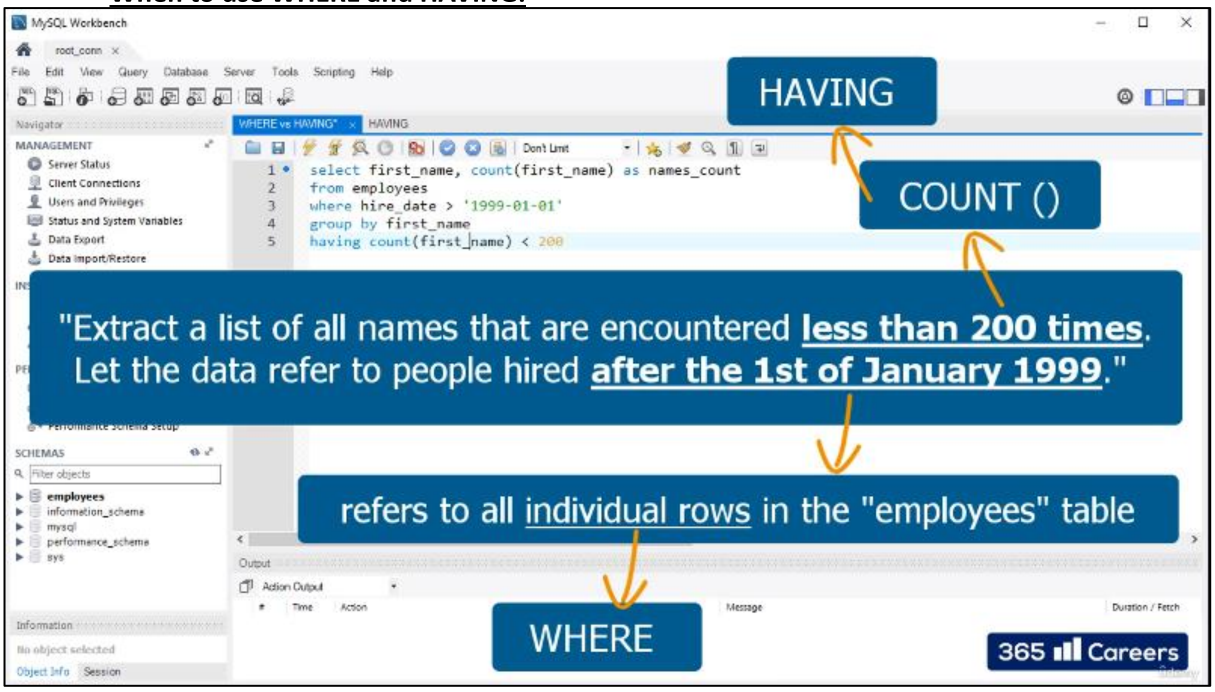


```sql
SELECT emp_no, AVG(salary) AS avg_salary
FROM salaries
GROUP BY emp_no
HAVING AVG(salary) > 120000
ORDER BY avg_salary ASC;
```

emp_no	avg_salary
17238	120084.0000
29224	120089.6667
64633	120112.8889
51022	120150.9000
75138	120250.0000

My original query didn't work because I grouped by the salary column instead of by the employee number, which meant SQL was calculating averages per salary value rather than per employee. Even when you previously tried grouping by emp_no, it still failed because my SELECT list contained salary, a non-aggregated column that wasn't included in the GROUP BY, making the query invalid SQL. To fix this, we needed to select the aggregated value - AVG(salary) - and group only by emp_no, because that's the level at which you want to calculate the average. The reason it feels like we're "averaging twice" is simply that SQL requires the aggregate to appear in both the SELECT clause (to show it) and in the HAVING clause (to filter on it); it's not actually performing the calculation twice. Once these issues were corrected, the query returned the employees whose average salary exceeds $120,000 as intended. ESSENTIALLY IN ORDER FOR HAVING() TO WORK THE COLUMN NEEDS TO BE AGGREGATED PRIOR TO PUTTING IT IN THE HAVING() FUNCTION.

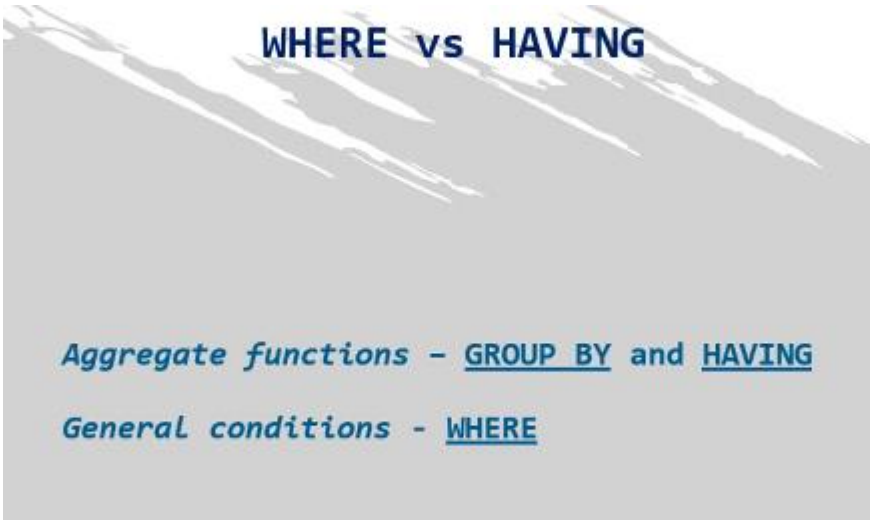
When to use WHERE and HAVING:



```
select first_name, count(first_name) as names_count
from employees
where hire_date > '1999-01-01'
group by first_name
having count(first_name) < 200
```

"Extract a list of all names that are encountered **less than 200 times**. Let the data refer to people hired **after the 1st of January 1999**."

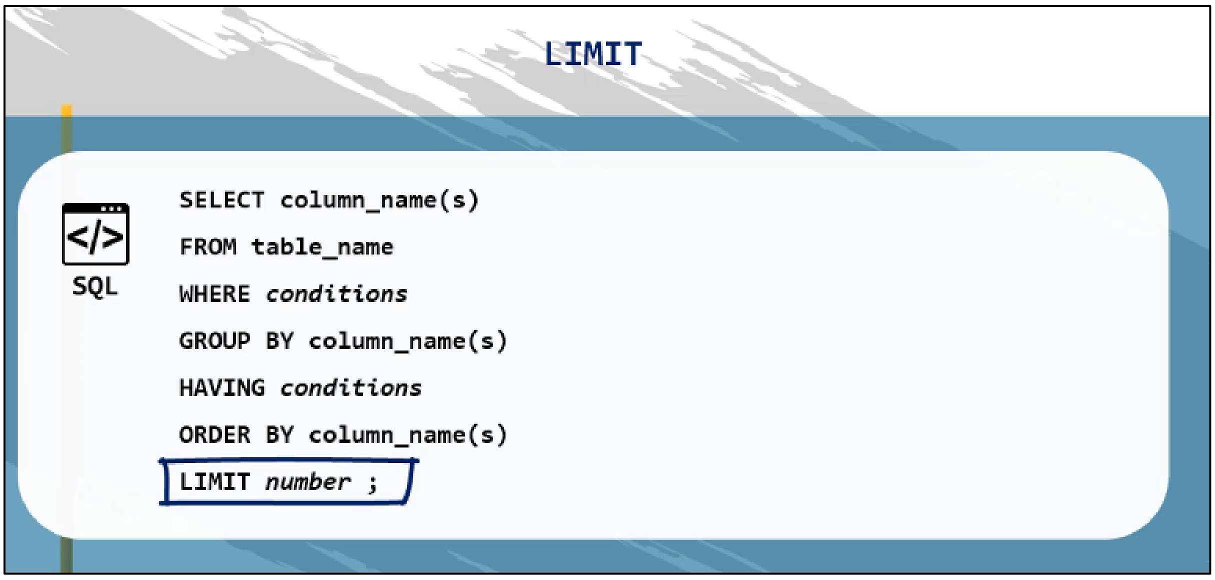
refers to all individual rows in the "employees" table



WHERE vs HAVING

Aggregate functions - GROUP BY and HAVING

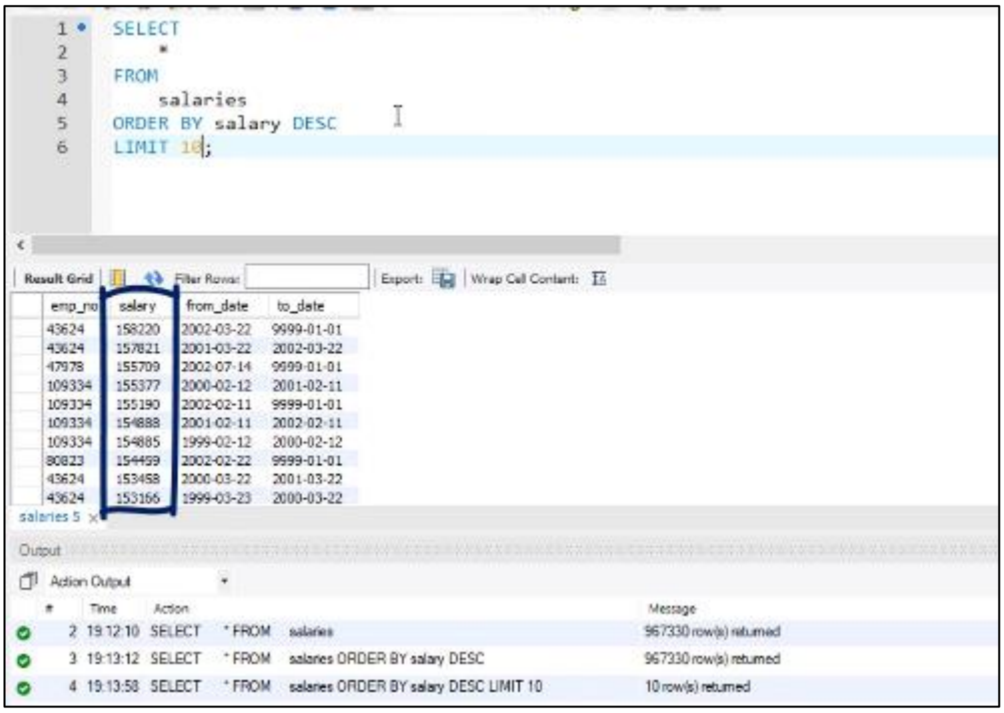
General conditions - WHERE



LIMIT

SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s)
LIMIT number ;
```



```
SELECT
    *
FROM
    salaries
ORDER BY salary DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_no	salary	from_date	to_date
43624	158220	2002-03-22	9999-01-01
43624	157821	2001-03-22	2002-03-22
47978	155709	2002-07-14	9999-01-01
109334	155377	2000-02-12	2001-02-11
109334	155190	2002-02-11	9999-01-01
109334	154888	2001-02-11	2002-02-11
109334	154885	1999-02-12	2000-02-12
80823	154459	2002-02-22	9999-01-01
43624	153458	2000-03-22	2001-03-22
43624	153166	1999-03-23	2000-03-22

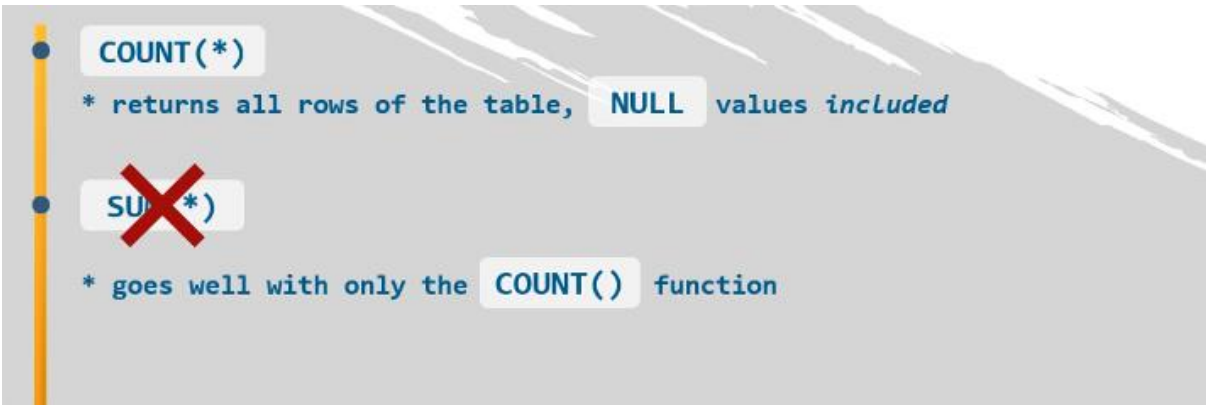
salaries 5

Output

Action Output

#	Time	Action	Message
2	19:12:10	SELECT * FROM salaries	967330 row(s) returned
3	19:13:12	SELECT * FROM salaries ORDER BY salary DESC	967330 row(s) returned
4	19:13:58	SELECT * FROM salaries ORDER BY salary DESC LIMIT 10	10 row(s) returned

He also explained the other ways one can limit the output of our queries – it's not hard whatsoever, I guess you can always go back to Section 21 video 175 should it ever become something of an enigma, but I highly doubt it.

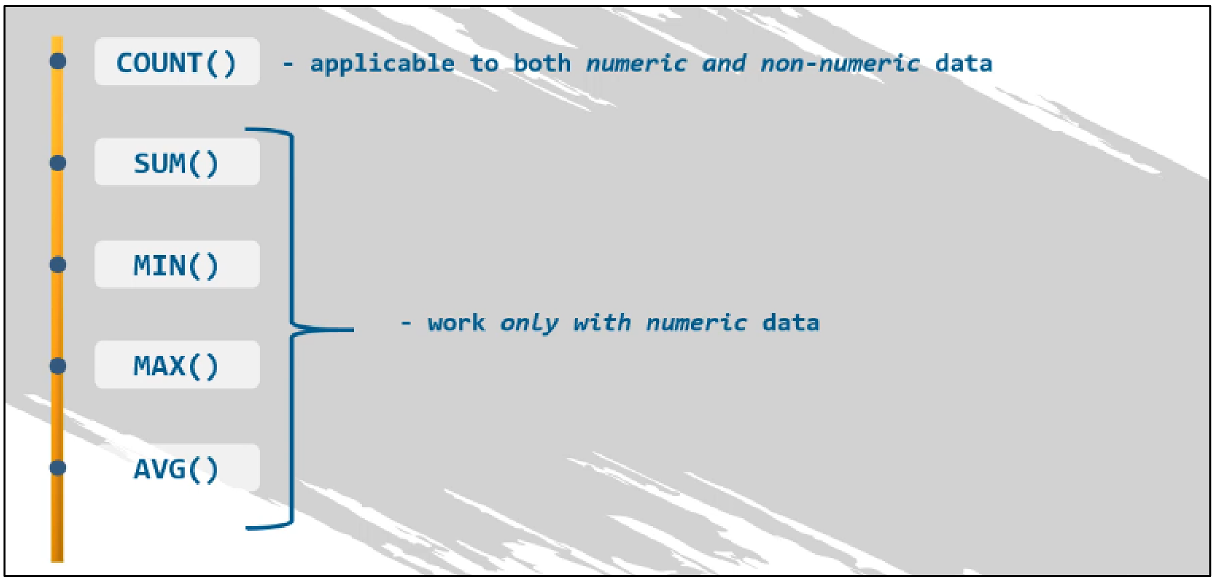


COUNT(*)

* returns all rows of the table, NULL values *included*

~~SUM(*)~~

* goes well with only the COUNT() function



COUNT() - applicable to both *numeric and non-numeric* data

SUM()

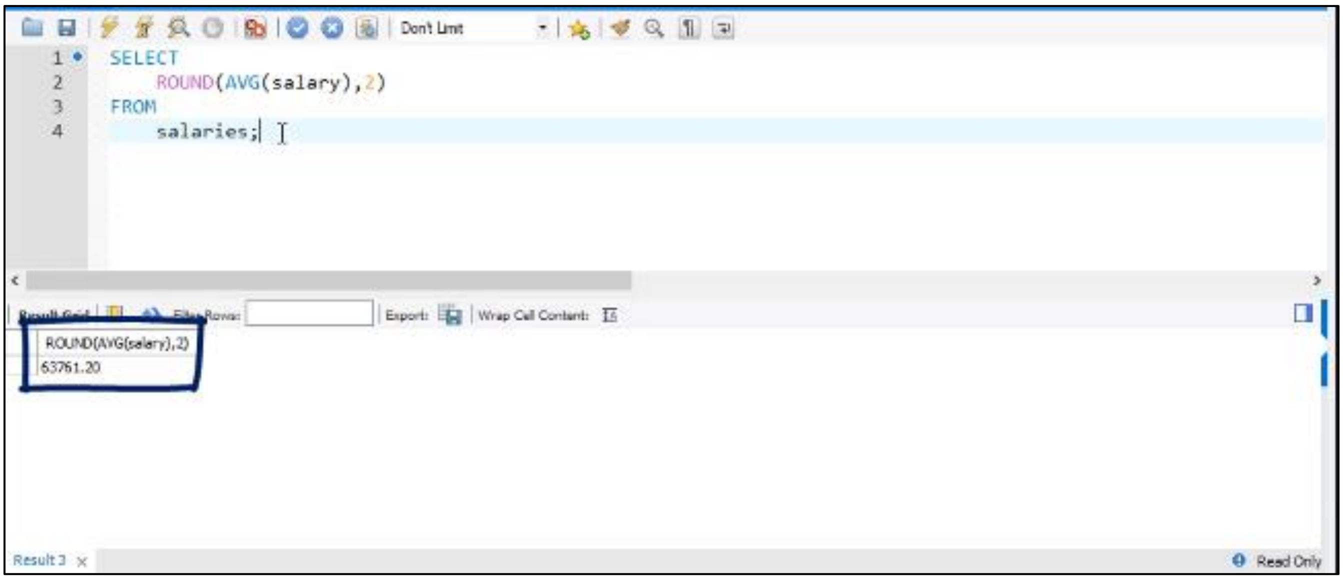
MIN()

MAX()

AVG()

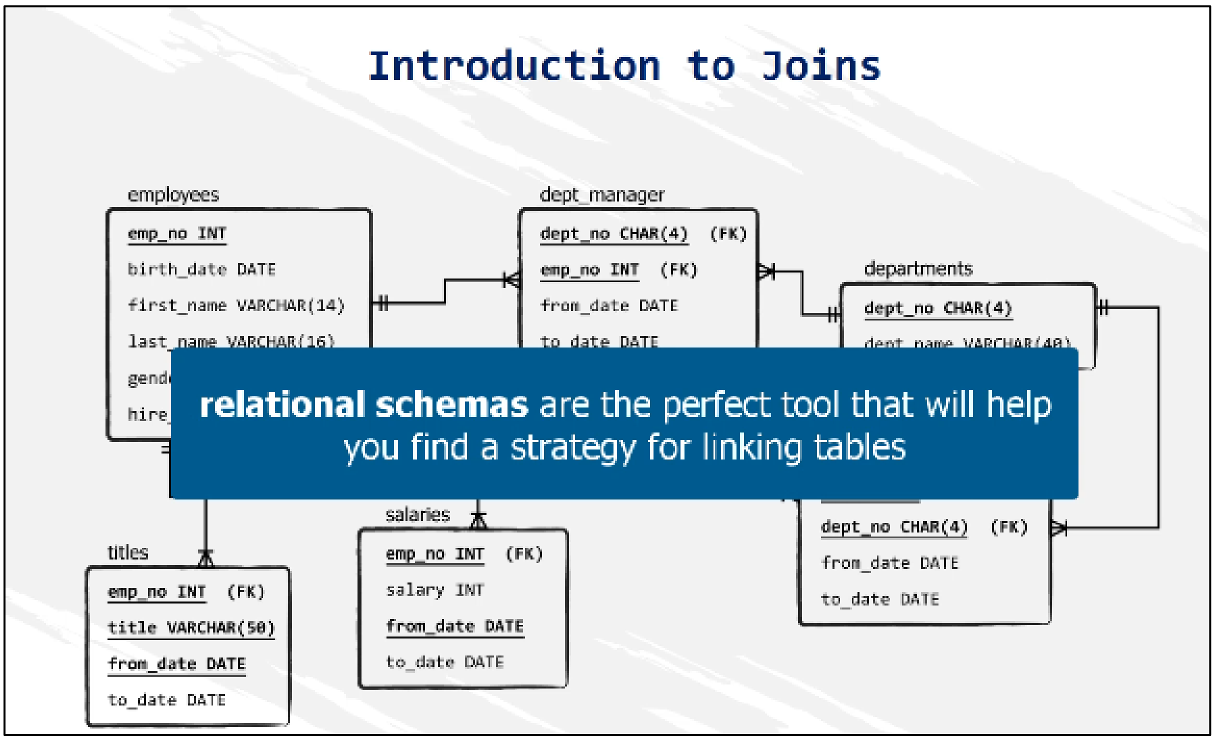
- work *only* with numeric data

Use of ROUND() function:



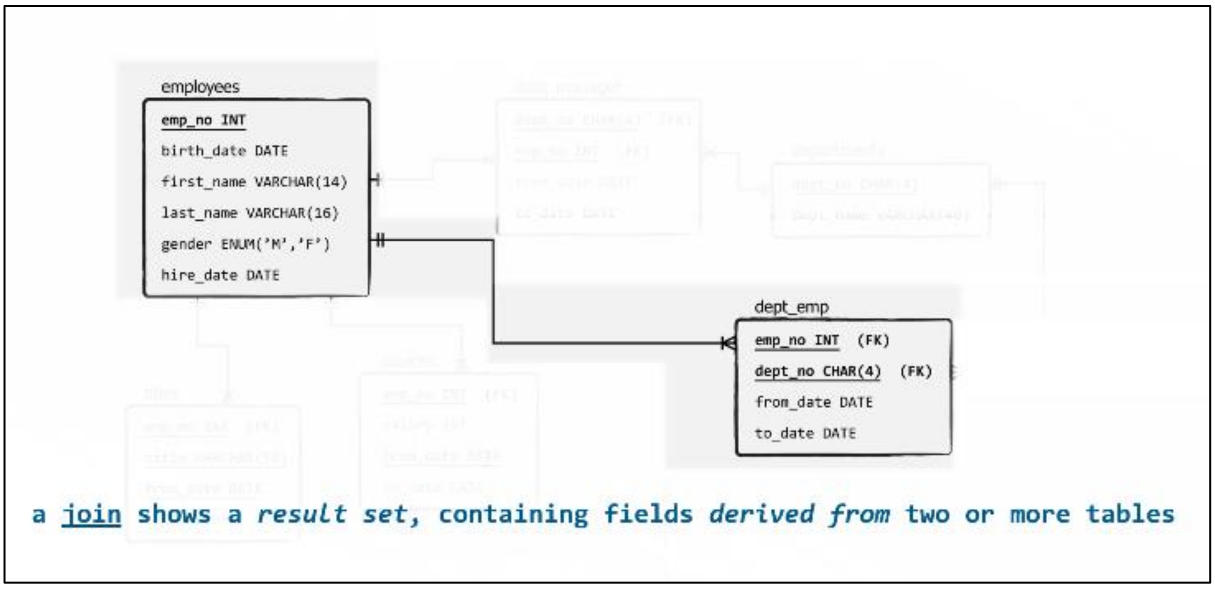
```
SELECT
    ROUND(AVG(salary),2)
FROM
    salaries;
```

```
ROUND(AVG(salary),2)
63761.20
```

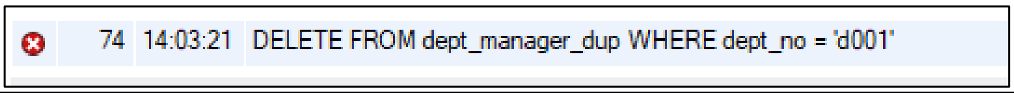


Introduction to Joins

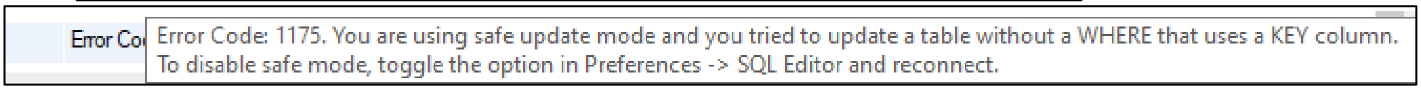
relational schemas are the perfect tool that will help you find a strategy for linking tables



Error code:1175 workaround:



```
74  14:03:21  DELETE FROM dept_manager_dup WHERE dept_no = 'd001'
```



```
Error Co  Error Code: 1175. You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column.
          To disable safe mode, toggle the option in Preferences -> SQL Editor and reconnect.
```

<u>Why the error happens:</u>

MySQL Workbench's *safe update mode* blocks any DELETE or UPDATE query unless:

- **The WHERE clause uses a key column (usually a PRIMARY KEY or an indexed column), or**
- **You include a LIMIT.**

In your table, dept_no is NOT a key column (because your CREATE TABLE did not define a PRIMARY KEY).

Therefore:

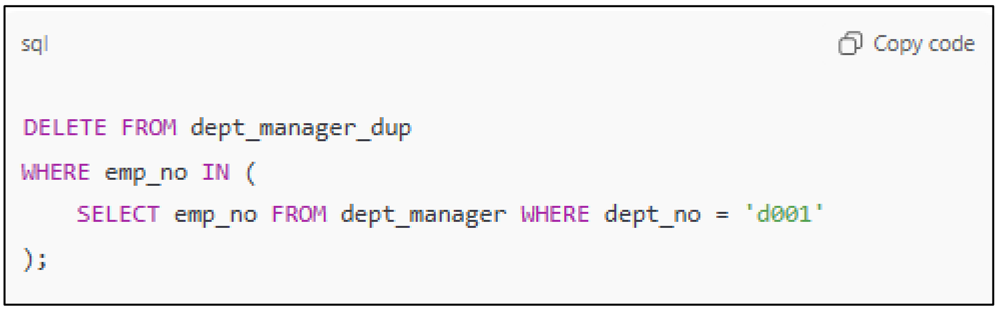
```sql
DELETE FROM dept_manager_dup
WHERE dept_no = 'd001';
```

gets blocked.

This will delete all rows matching dept_no = 'd001' while staying within safe-update rules.

<u>Alternative workaround (without disabling safe mode):</u>

Use the primary key — emp_no — in the WHERE clause:



```sql
DELETE FROM dept_manager_dup
WHERE emp_no IN (
    SELECT emp_no FROM dept_manager WHERE dept_no = 'd001'
);
```

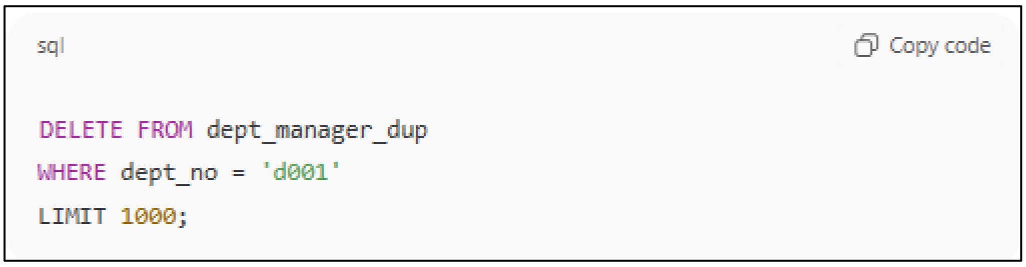
However, MySQL won't allow deleting from a table while selecting from the same table unless wrapped differently.

So the safer practical workaround is:

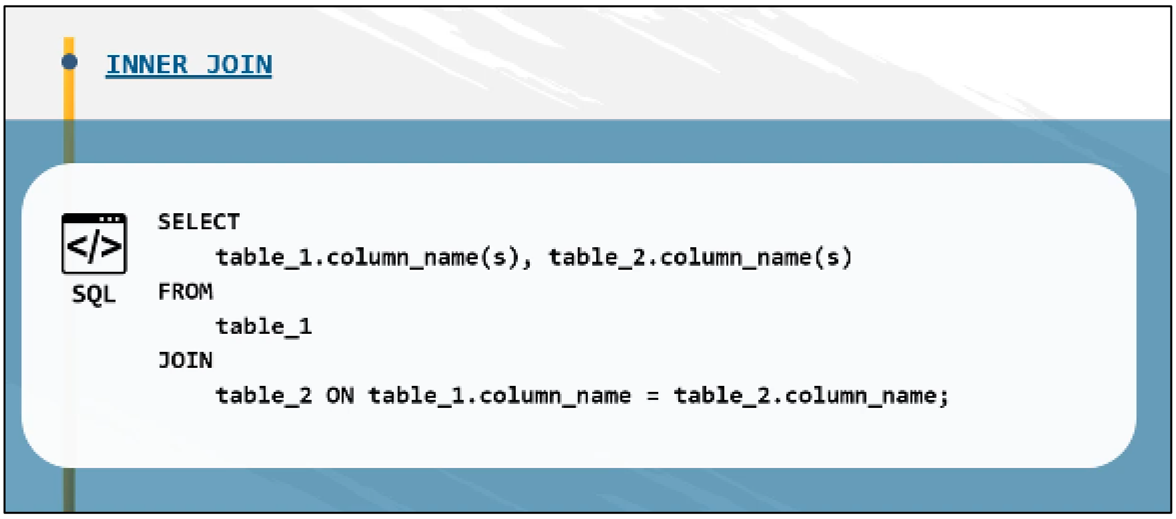
✓ Use LIMIT.

DON'T DISABLE SAFE MODE.

Final recommendation:

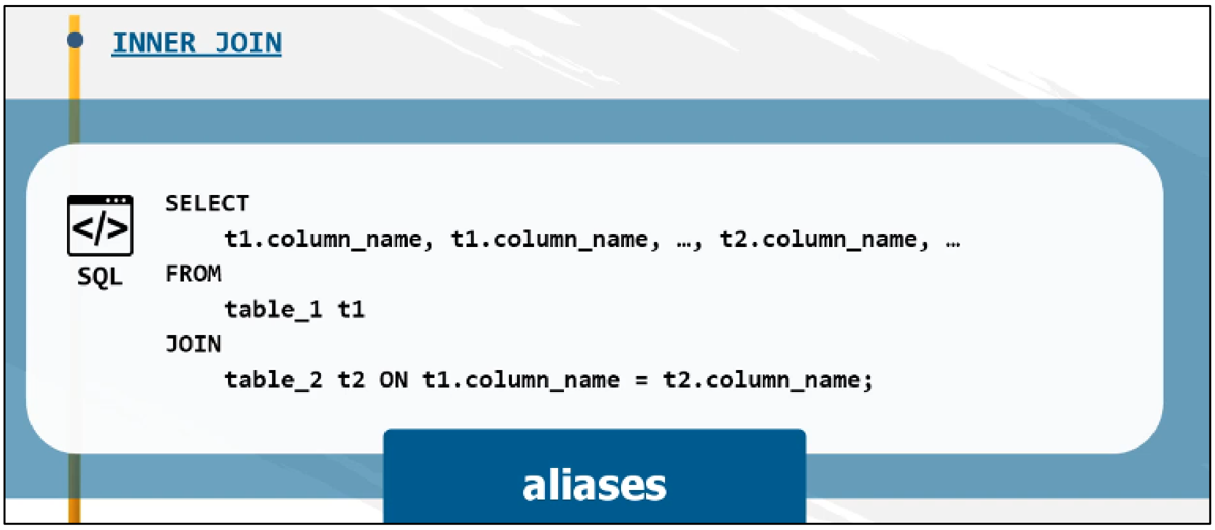


```sql
DELETE FROM dept_manager_dup
WHERE dept_no = 'd001'
LIMIT 1000;
```



```sql
SELECT
    table_1.column_name(s), table_2.column_name(s)
FROM
    table_1
JOIN
    table_2 ON table_1.column_name = table_2.column_name;
```

In a select statement, write all columns you wish to see in the result. It is essential to designate the tables to which the columns belong, as the data is not contained in a single table this time. That's why, besides typing the keyword "from", and the name of the first table, you should proceed by writing "join" and the name of the second table. The syntax allows us to specify the fields we would like to see in the result in the tables we are matching.



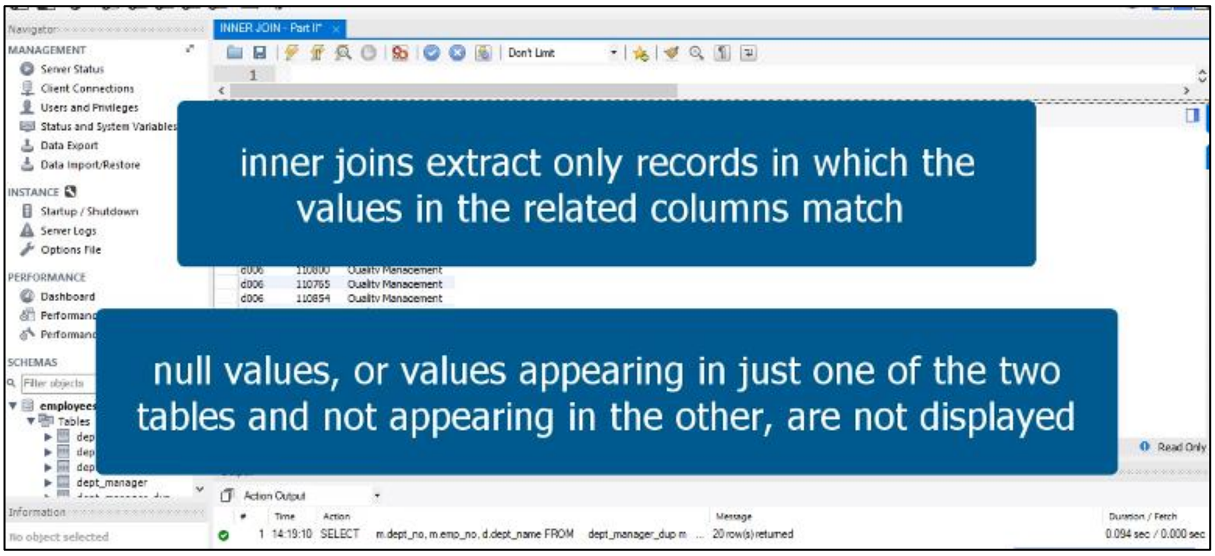
INNER JOIN

SQL

```
SELECT
    t1.column_name, t1.column_name, …, t2.column_name, …
FROM
    table_1 t1
JOIN
    table_2 t2 ON t1.column_name = t2.column_name;
```

aliases

Before we proceed with our example, let's share a fundamental coding practice that professionals use in the joint syntax. **Aliases**. More precisely, we're talking about aliases of the table's names. This means table one can be renamed to say T1 and table two to T2. When used for assigning table names, the aliases are usually added right after the original table name, without using the keyword "as". Then, instead of typing the entire table's names in the select block, we can use T1 and T2, respectively.



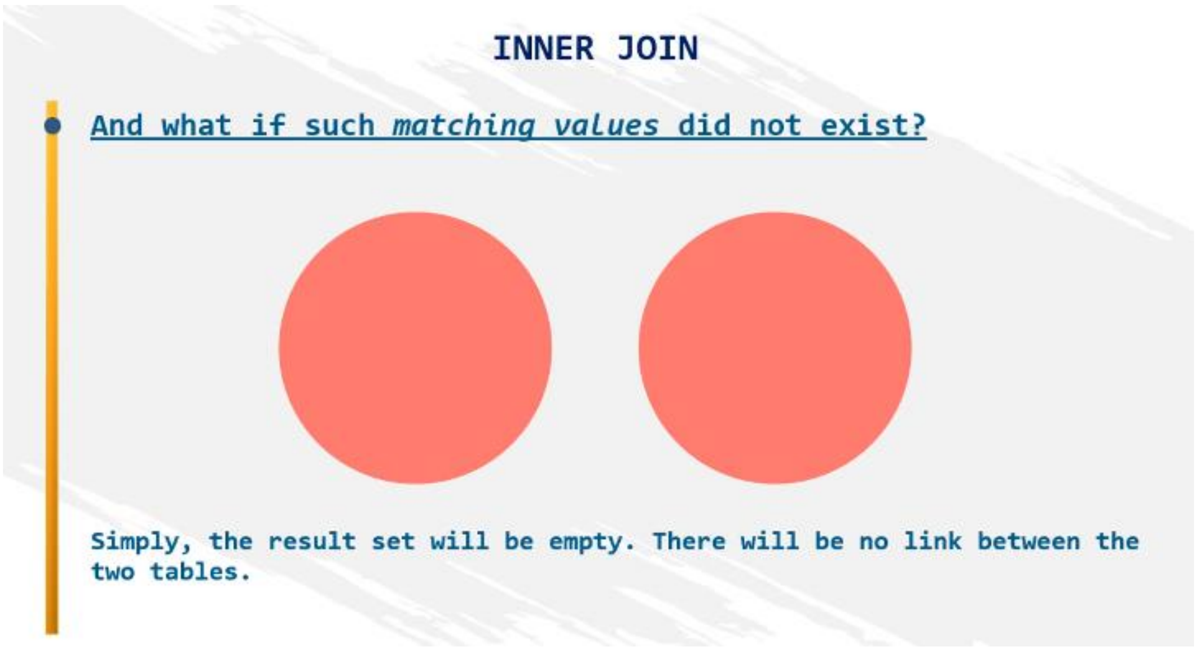
inner joins extract only records in which the values in the related columns match

null values, or values appearing in just one of the two tables and not appearing in the other, are not displayed

```
d006    110765  Quality Management
d006    110854  Quality Management
```

```
#  Time      Action                                                       Message             Duration / Fetch
1  14:19:10  SELECT  m.dept_no, m.emp_no, d.dept_name FROM  dept_manager_dup m ...  20 row(s) returned  0.094 sec / 0.000 sec
```

So when using the JOIN/ INNER JOIN function, we won't get any null values or incomplete records.



INNER JOIN

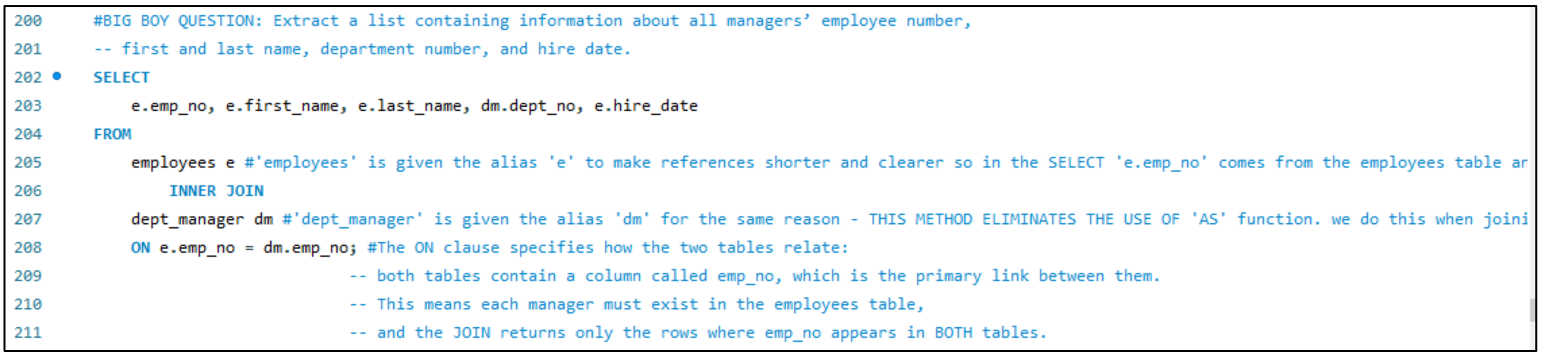
And what if such *matching values* did not exist?

Simply, the result set will be empty. There will be no link between the two tables.

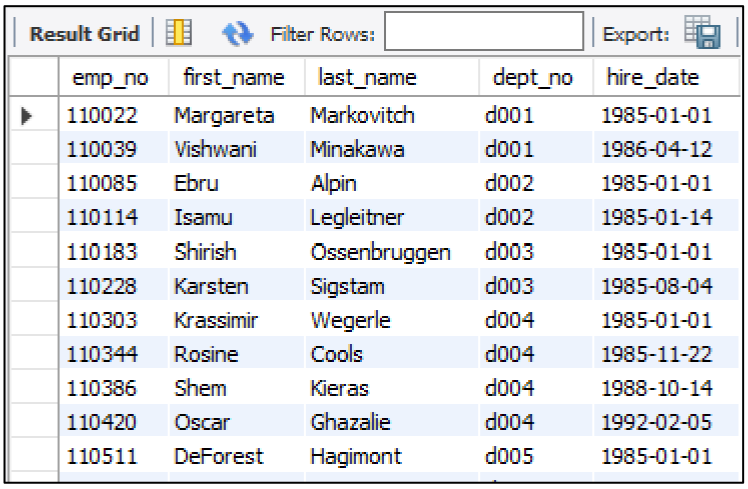
The terms "JOIN" and "INNER JOIN" refer to the same type of SQL operation used to combine rows from two or more tables based on a related column between them. Here are the key points about their usage:

1. **Functionality**: Both "JOIN" and "INNER JOIN" return rows where there is a match in both tables. If there are no matching records, those rows will not be included in the result set.
2. **Interchangeability**: As per the course material, they are functionally equivalent, meaning you can use either term without affecting the outcome of your SQL query.
3. **Readability**: Using "INNER JOIN" may enhance clarity, especially in queries that involve multiple types of joins (like LEFT JOIN or RIGHT JOIN). This helps in identifying which type of join is being applied at a glance.
4. **Preference**: While you can use either "JOIN" or "INNER JOIN," some developers prefer to use "INNER JOIN" for the sake of clarity, particularly in complex queries.

My work:

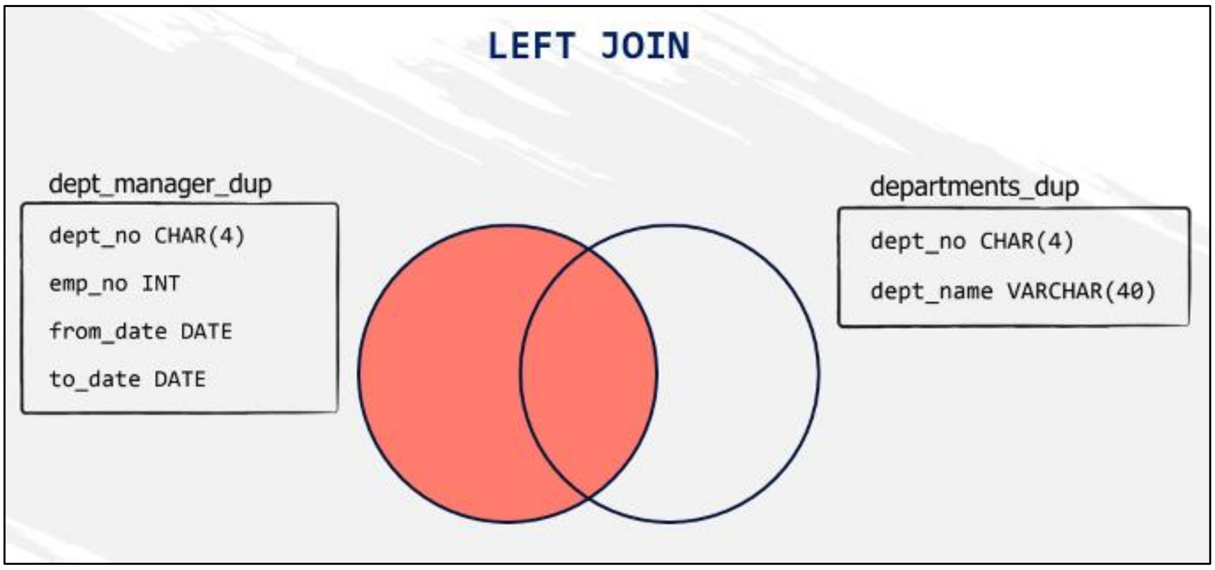


```sql
#BIG BOY QUESTION: Extract a list containing information about all managers’ employee number,
-- first and last name, department number, and hire date.
SELECT
    e.emp_no, e.first_name, e.last_name, dm.dept_no, e.hire_date
FROM
    employees e #'employees' is given the alias 'e' to make references shorter and clearer so in the SELECT 'e.emp_no' comes from the employees table ar
        INNER JOIN
    dept_manager dm #'dept_manager' is given the alias 'dm' for the same reason - THIS METHOD ELIMINATES THE USE OF 'AS' function. we do this when joini
    ON e.emp_no = dm.emp_no; #The ON clause specifies how the two tables relate:
                          -- both tables contain a column called emp_no, which is the primary link between them.
                          -- This means each manager must exist in the employees table,
                          -- and the JOIN returns only the rows where emp_no appears in BOTH tables.
```

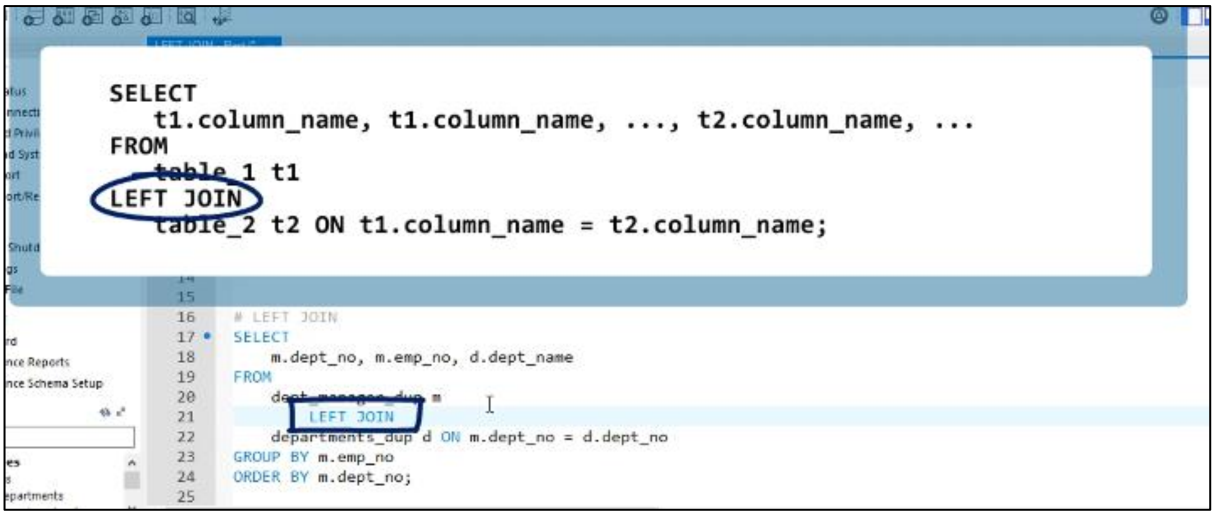


Result Grid | Filter Rows: | Export:

emp_no	first_name	last_name	dept_no	hire_date
110022	Margareta	Markovitch	d001	1985-01-01
110039	Vishwani	Minakawa	d001	1986-04-12
110085	Ebru	Alpin	d002	1985-01-01
110114	Isamu	Legleitner	d002	1985-01-14
110183	Shirish	Ossenbruggen	d003	1985-01-01
110228	Karsten	Sigstam	d003	1985-08-04
110303	Krassimir	Wegerle	d004	1985-01-01
110344	Rosine	Cools	d004	1985-11-22
110386	Shem	Kieras	d004	1988-10-14
110420	Oscar	Ghazalie	d004	1992-02-05
110511	DeForest	Hagimont	d005	1985-01-01

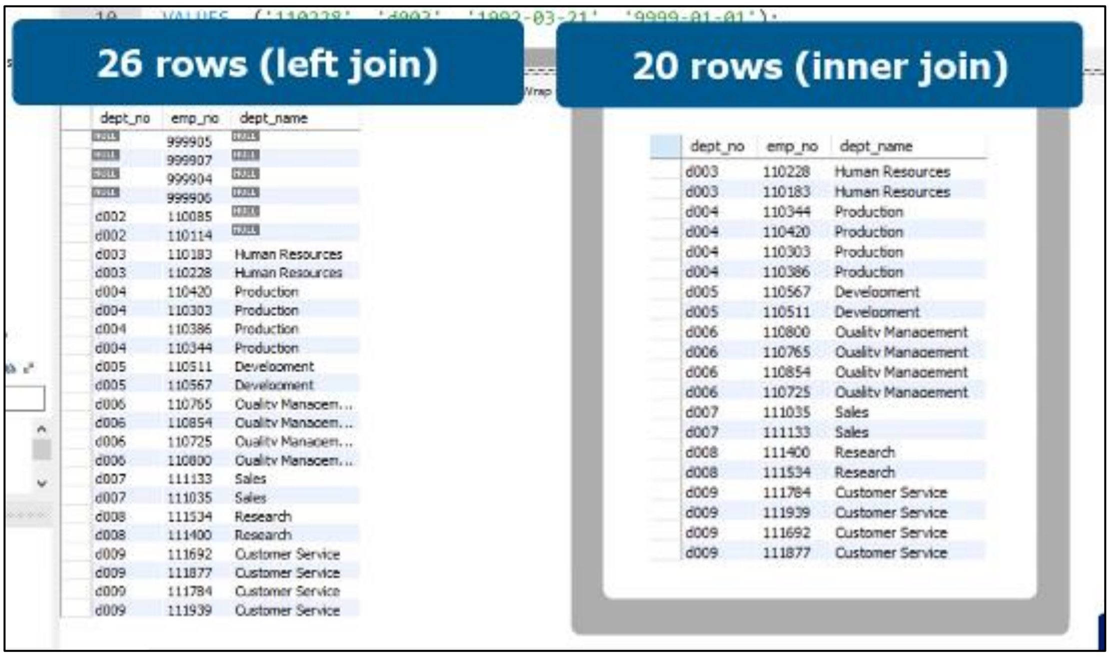


The Venn diagram you see here allows you to visualise how a Left Join works. Its output allows us to see all records from the table on the left side of the Join, including all matching rows of the two tables. That's why, compared to the Inner Join, the results set, coloured in red, includes the rest of the area of the left table. In SQL terms, this translates in retrieving all matching values of the two tables, plus all values from the left table that match no values from the right table.



```
SELECT
   t1.column_name, t1.column_name, ..., t2.column_name, ...
FROM
   table_1 t1
LEFT JOIN
   table_2 t2 ON t1.column_name = t2.column_name;
```

```
# LEFT JOIN
SELECT
    m.dept_no, m.emp_no, d.dept_name
FROM
    dept_manager_dup m
        LEFT JOIN
    departments_dup d ON m.dept_no = d.dept_no
GROUP BY m.emp_no
ORDER BY m.dept_no;
```



26 rows (left join)

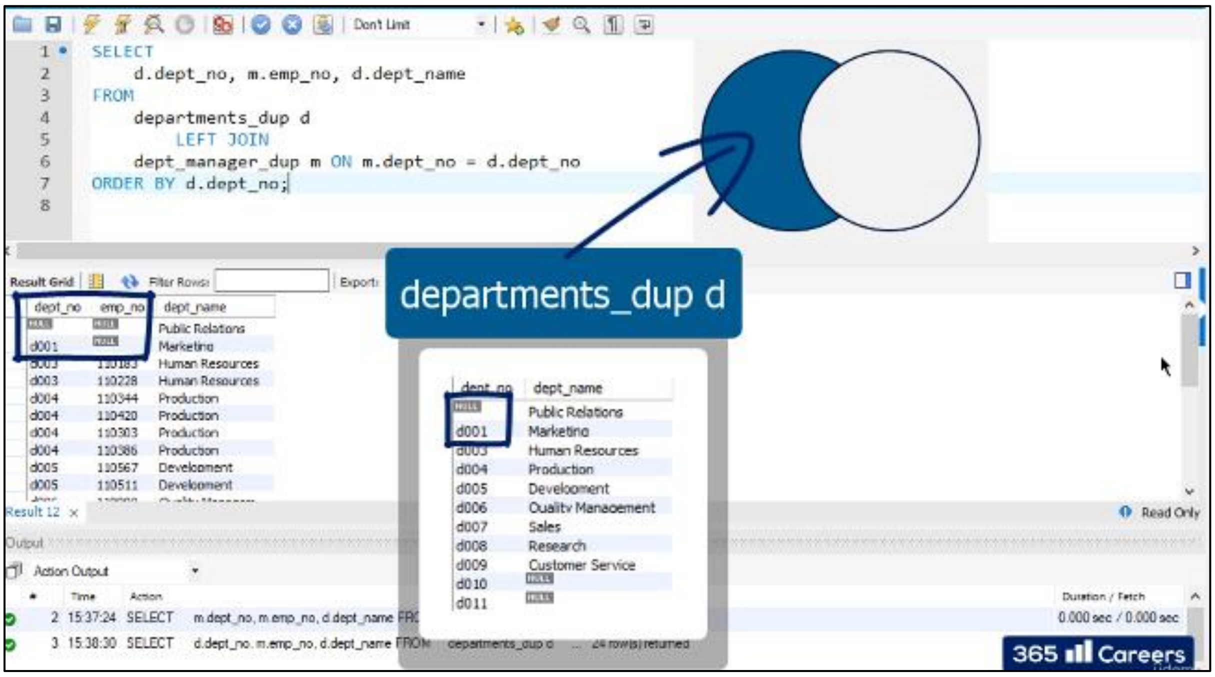
dept_no	emp_no	dept_name
NULL	999905	NULL
NULL	999907	NULL
NULL	999904	NULL
NULL	999906	NULL
d002	110085	NULL
d002	110114	NULL
d003	110183	Human Resources
d003	110228	Human Resources
d004	110420	Production
d004	110303	Production
d004	110386	Production
d004	110344	Production
d005	110511	Development
d005	110567	Development
d006	110765	Quality Managem...
d006	110854	Quality Managem...
d006	110725	Quality Managem...
d006	110800	Quality Managem...
d007	111133	Sales
d007	111035	Sales
d008	111534	Research
d008	111400	Research
d009	111692	Customer Service
d009	111877	Customer Service
d009	111784	Customer Service
d009	111939	Customer Service

20 rows (inner join)

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development
d006	110800	Quality Management
d006	110765	Quality Management
d006	110854	Quality Management
d006	110725	Quality Management
d007	111035	Sales
d007	111133	Sales
d008	111400	Research
d008	111534	Research
d009	111784	Customer Service
d009	111939	Customer Service
d009	111692	Customer Service
d009	111877	Customer Service

It returned 26 rows. Six rows more than the 20 rows we obtained in the example about Inner Joins. Basically, this is proof that differently from what we saw for Inner Joins, when working with Left Joins, the order in which you join tables matters. Having the manager's table, M, or the department's table, D, on the left can change results completely.

e.g.



```sql
SELECT
    d.dept_no, m.emp_no, d.dept_name
FROM
    departments_dup d
        LEFT JOIN
    dept_manager_dup m ON m.dept_no = d.dept_no
ORDER BY d.dept_no;
```

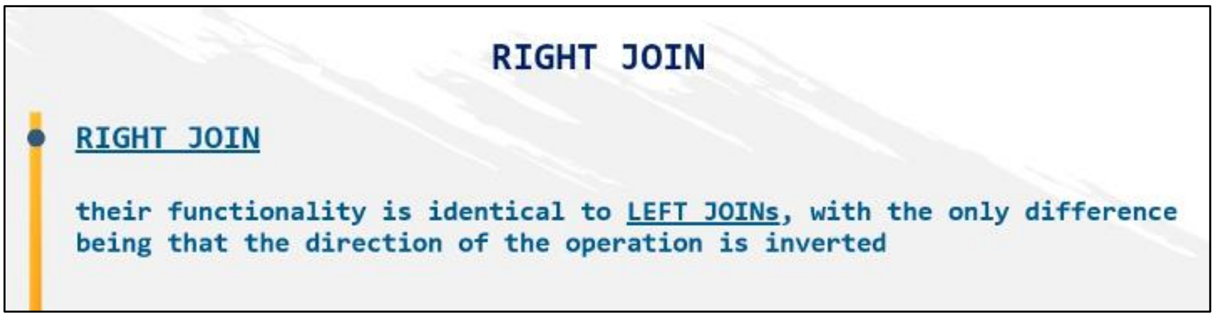
dept_no	emp_no	dept_name
NULL	NULL	Public Relations
d001	NULL	Marketing
d003	110183	Human Resources
d003	110228	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development

departments_dup d

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

```
2  15:37:24  SELECT  m.dept_no, m.emp_no, d.dept_name FRO
3  15:38:30  SELECT  d.dept_no, m.emp_no, d.dept_name FROM  departments_dup d  ...  24 row(s) returned
```

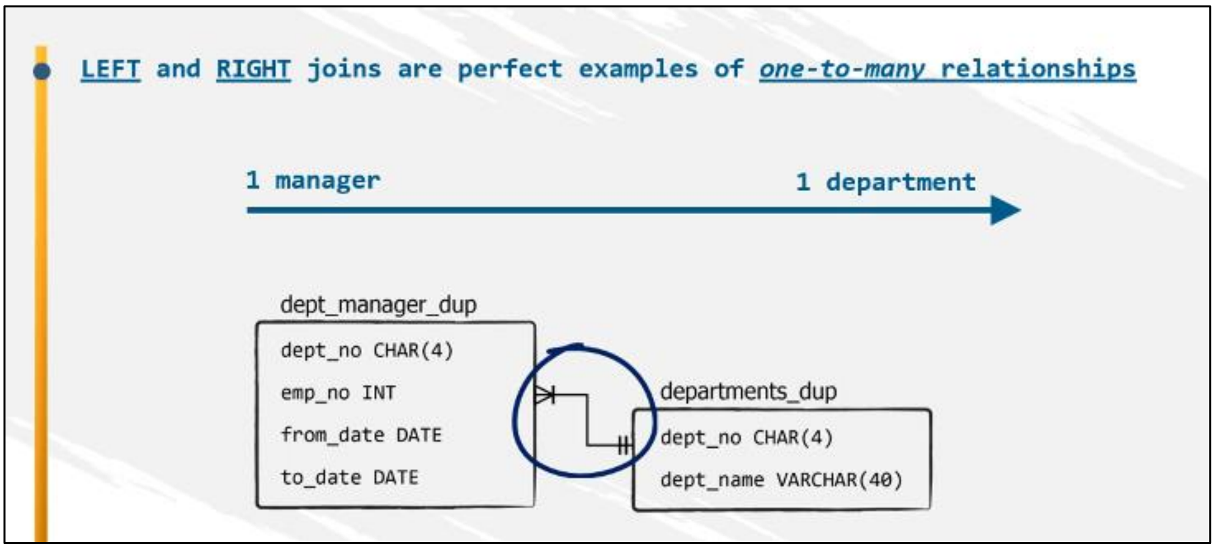
LEFT JOIN = LEFT OUTER JOIN (Used interchangeably)



RIGHT JOIN

RIGHT JOIN

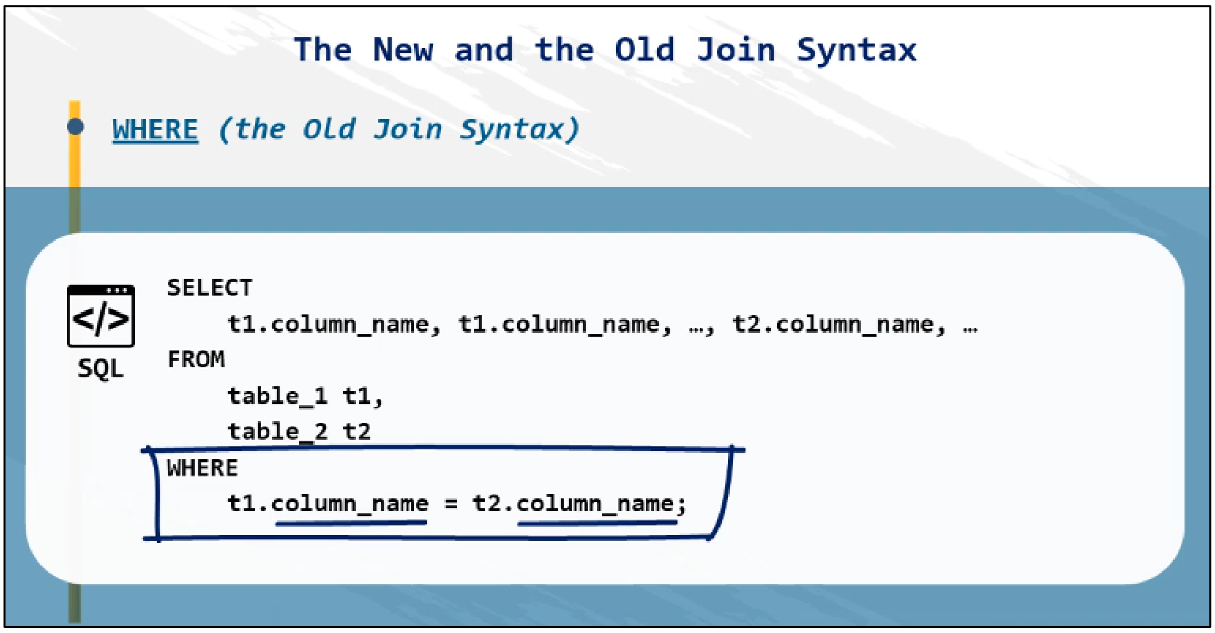
their functionality is identical to LEFT JOINs, with the only difference being that the direction of the operation is inverted



LEFT and RIGHT joins are perfect examples of *one-to-many relationships*

In addition, when talking about relationships, left and right joins are perfect examples of one-to-many relationships in my SQL. For instance, in our last example, when we used a left join, each department from the department's duplicate table, as represented by the department number, could have been the department of one or more managers from the department manager duplicate. A manager who is also an employee can belong to a single department only. This is an example of how the one-to-many relationship can be exhibited in a left or right join case.

The same results we obtained by using the JOIN function, can also be obtained via the WHERE function: **<u>THE NEW & OLD JOIN SYNTAX</u>**

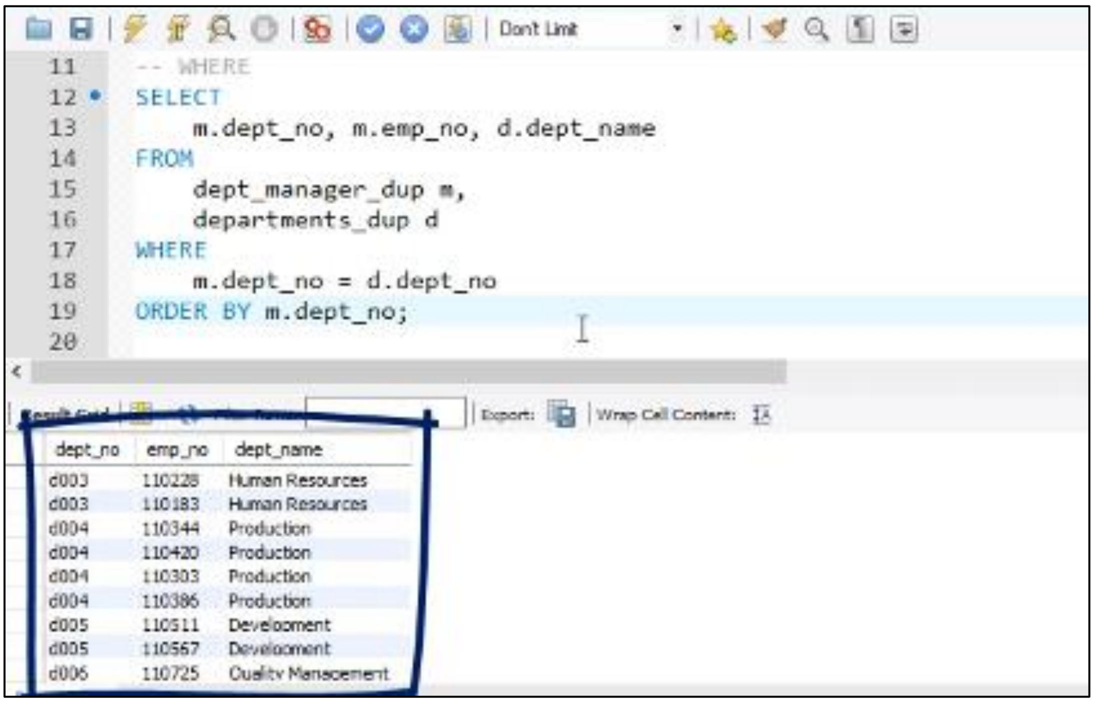


The New and the Old Join Syntax

- WHERE *(the Old Join Syntax)*

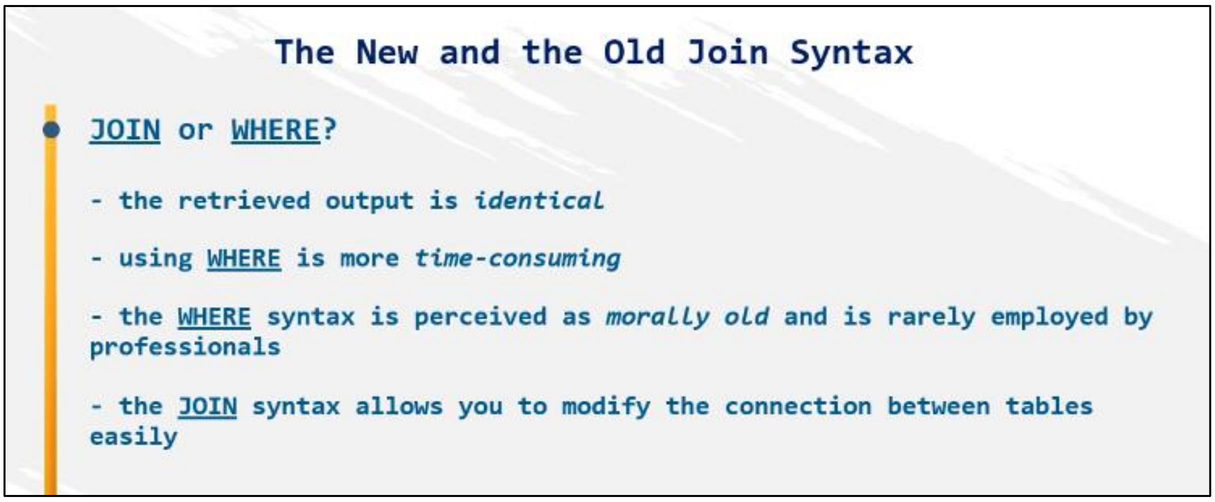
SQL

```
SELECT
    t1.column_name, t1.column_name, …, t2.column_name, …
FROM
    table_1 t1,
    table_2 t2
WHERE
    t1.column_name = t2.column_name;
```



```
-- WHERE
SELECT
    m.dept_no, m.emp_no, d.dept_name
FROM
    dept_manager_dup m,
    departments_dup d
WHERE
    m.dept_no = d.dept_no
ORDER BY m.dept_no;
```

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110511	Development
d005	110567	Development
d006	110725	Quality Management



The New and the Old Join Syntax

- JOIN or WHERE?

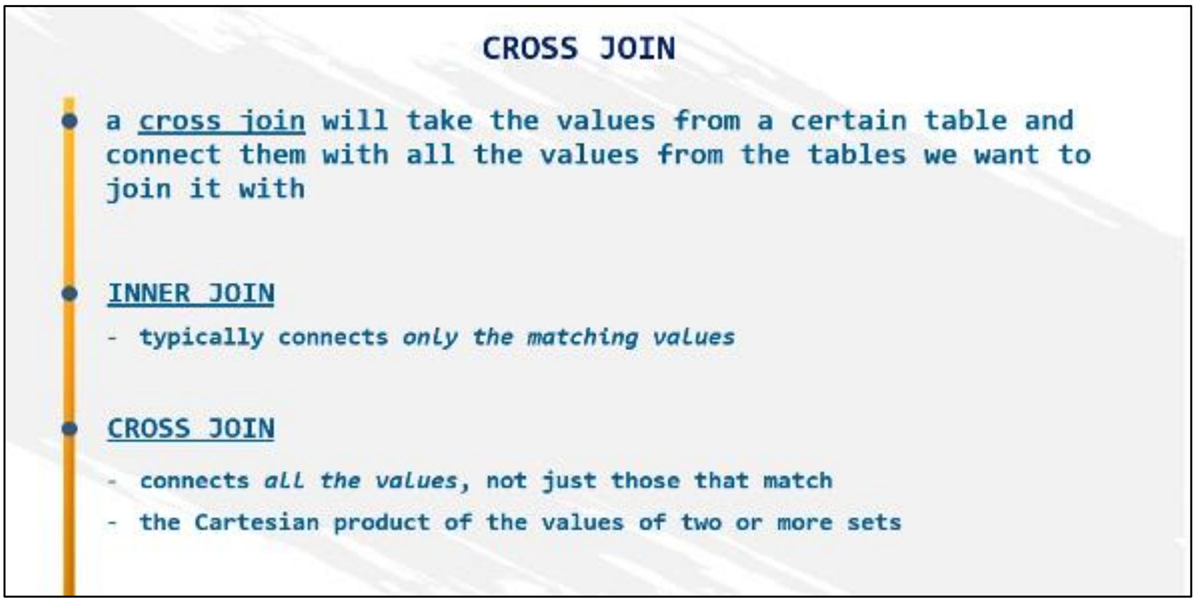
  - the retrieved output is *identical*

  - using WHERE is more *time-consuming*

  - the WHERE syntax is perceived as *morally old* and is rarely employed by professionals

  - the JOIN syntax allows you to modify the connection between tables easily

CROSS JOIN FUNCTION



CROSS JOIN

- a cross join will take the values from a certain table and connect them with all the values from the tables we want to join it with

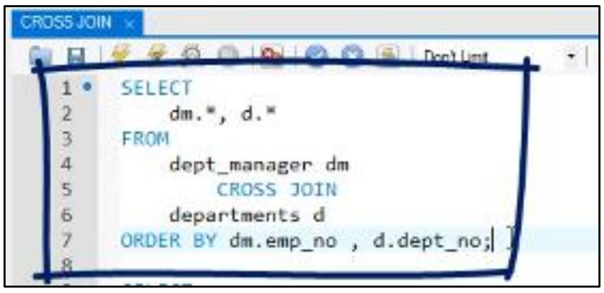
- INNER JOIN
  - typically connects *only the matching values*

- CROSS JOIN
  - connects *all the values*, not just those that match
  - the Cartesian product of the values of two or more sets

A cross join will take the values from a certain table and connect them with all the values from the tables we want to join it with. This is in contrast to the inner join that typically connects only to matching values. A cross join will connect all the values, not just those that match. That's why, from a mathematical point of view, a cross join is the Cartesian product of the values of two or more sets.

A cross join is particularly useful when the tables in a database are not well-connected. We must admit that the Employees Database is not really suitable for applying this kind of join meaningfully since its tables are indeed well connected. However, we can still use the employees database just to do an exercise with a cross join, can't we?

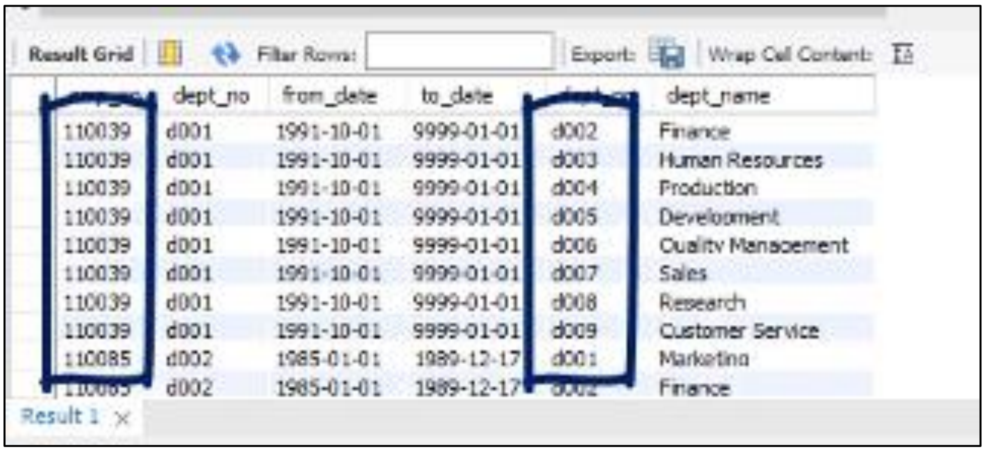
Here's an example:



```
SELECT
    dm.*, d.*
FROM
    dept_manager dm
        CROSS JOIN
    departments d
ORDER BY dm.emp_no , d.dept_no;
```

To visualize our output better, we will order the values by employee number as specified in the department manager table. and then by department number as specified in the department's table.

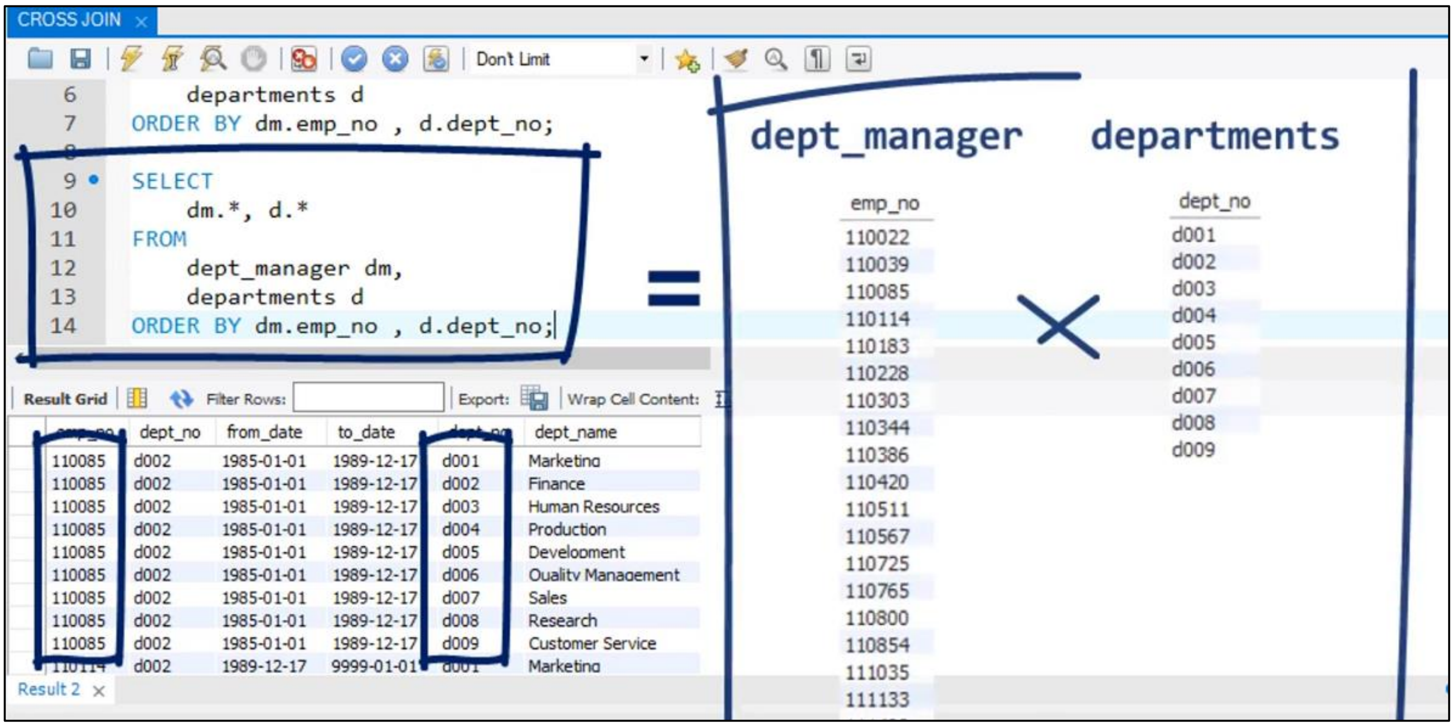
RESULTS:



emp_no	dept_no	from_date	to_date	dept_no	dept_name
110039	d001	1991-10-01	9999-01-01	d002	Finance
110039	d001	1991-10-01	9999-01-01	d003	Human Resources
110039	d001	1991-10-01	9999-01-01	d004	Production
110039	d001	1991-10-01	9999-01-01	d005	Development
110039	d001	1991-10-01	9999-01-01	d006	Quality Management
110039	d001	1991-10-01	9999-01-01	d007	Sales
110039	d001	1991-10-01	9999-01-01	d008	Research
110039	d001	1991-10-01	9999-01-01	d009	Customer Service
110085	d002	1985-01-01	1989-12-17	d001	Marketing
110085	d002	1985-01-01	1989-12-17	d002	Finance

We can observe that all department managers have been connected with all departments. In other words, nine different department numbers correspond to the employee number of each manager. NOTICE: how emp_no and dept_no are in consecutive order? Remember it was first initially ordered by emp_no, then it would then be ordered by dept_no for the second table. And in the joining of both tables where the first table ends, a new order is established.

ANOTHER INTERESTING WAY OF DOING A CROSS JOIN -WITHOUT WHERE OR THE JOIN STATEMENT:



```
    departments d
ORDER BY dm.emp_no , d.dept_no;

SELECT
    dm.*, d.*
FROM
    dept_manager dm,
    departments d
ORDER BY dm.emp_no , d.dept_no;
```

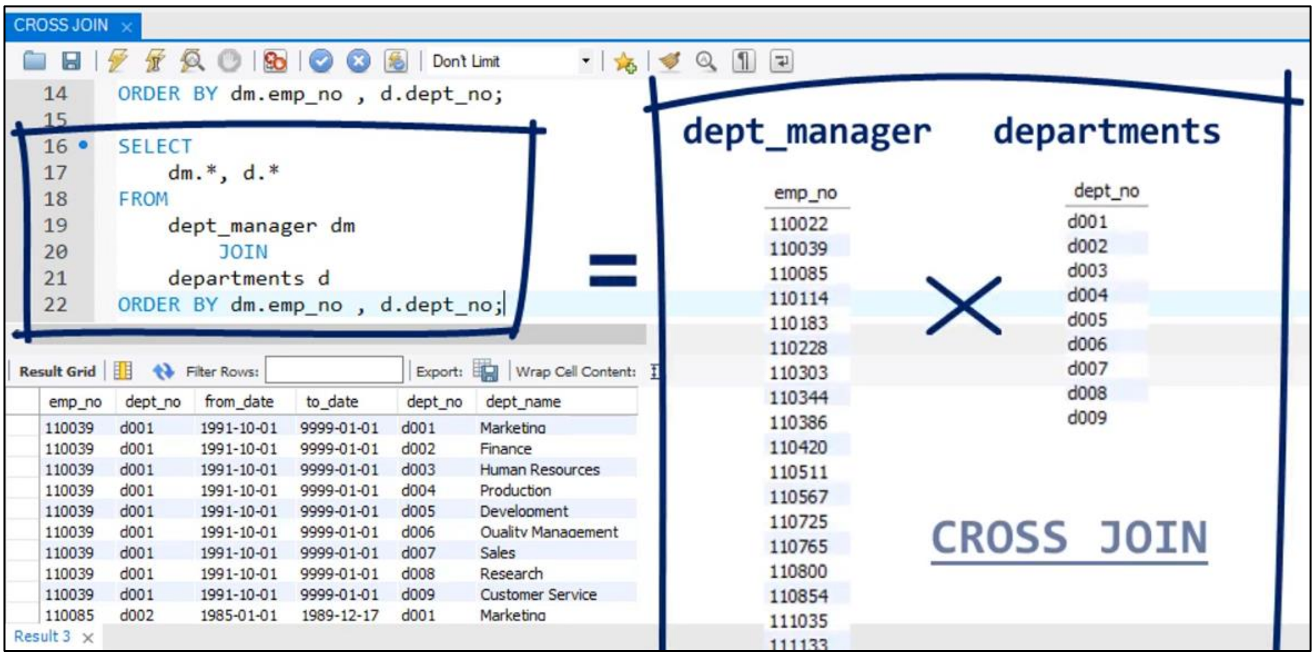
emp_no	dept_no	from_date	to_date	dept_no	dept_name
110085	d002	1985-01-01	1989-12-17	d001	Marketing
110085	d002	1985-01-01	1989-12-17	d002	Finance
110085	d002	1985-01-01	1989-12-17	d003	Human Resources
110085	d002	1985-01-01	1989-12-17	d004	Production
110085	d002	1985-01-01	1989-12-17	d005	Development
110085	d002	1985-01-01	1989-12-17	d006	Quality Management
110085	d002	1985-01-01	1989-12-17	d007	Sales
110085	d002	1985-01-01	1989-12-17	d008	Research
110085	d002	1985-01-01	1989-12-17	d009	Customer Service
110114	d002	1989-12-17	9999-01-01	d001	Marketing

= dept_manager × departments

emp_no	dept_no
110022	d001
110039	d002
110085	d003
110114	d004
110183	d005
110228	d006
110303	d007
110344	d008
110386	d009
110420	
110511	
110567	
110725	
110765	
110800	
110854	
111035	
111133	

Well, the result is the same. The answer is that this is exactly the output of a join between these two tables with no wear statement with which we can set a condition to the tables. Hence, the result is a cross join between department manager and departments.

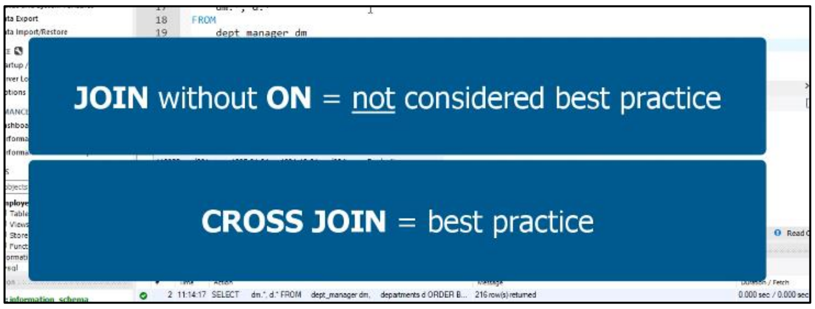
GOING Further in our analysis:



```
ORDER BY dm.emp_no , d.dept_no;

SELECT
    dm.*, d.*
FROM
    dept_manager dm
        JOIN
    departments d
ORDER BY dm.emp_no , d.dept_no;
```

emp_no	dept_no	from_date	to_date	dept_no	dept_name
110039	d001	1991-10-01	9999-01-01	d001	Marketing
110039	d001	1991-10-01	9999-01-01	d002	Finance
110039	d001	1991-10-01	9999-01-01	d003	Human Resources
110039	d001	1991-10-01	9999-01-01	d004	Production
110039	d001	1991-10-01	9999-01-01	d005	Development
110039	d001	1991-10-01	9999-01-01	d006	Quality Management
110039	d001	1991-10-01	9999-01-01	d007	Sales
110039	d001	1991-10-01	9999-01-01	d008	Research
110039	d001	1991-10-01	9999-01-01	d009	Customer Service
110085	d002	1985-01-01	1989-12-17	d001	Marketing

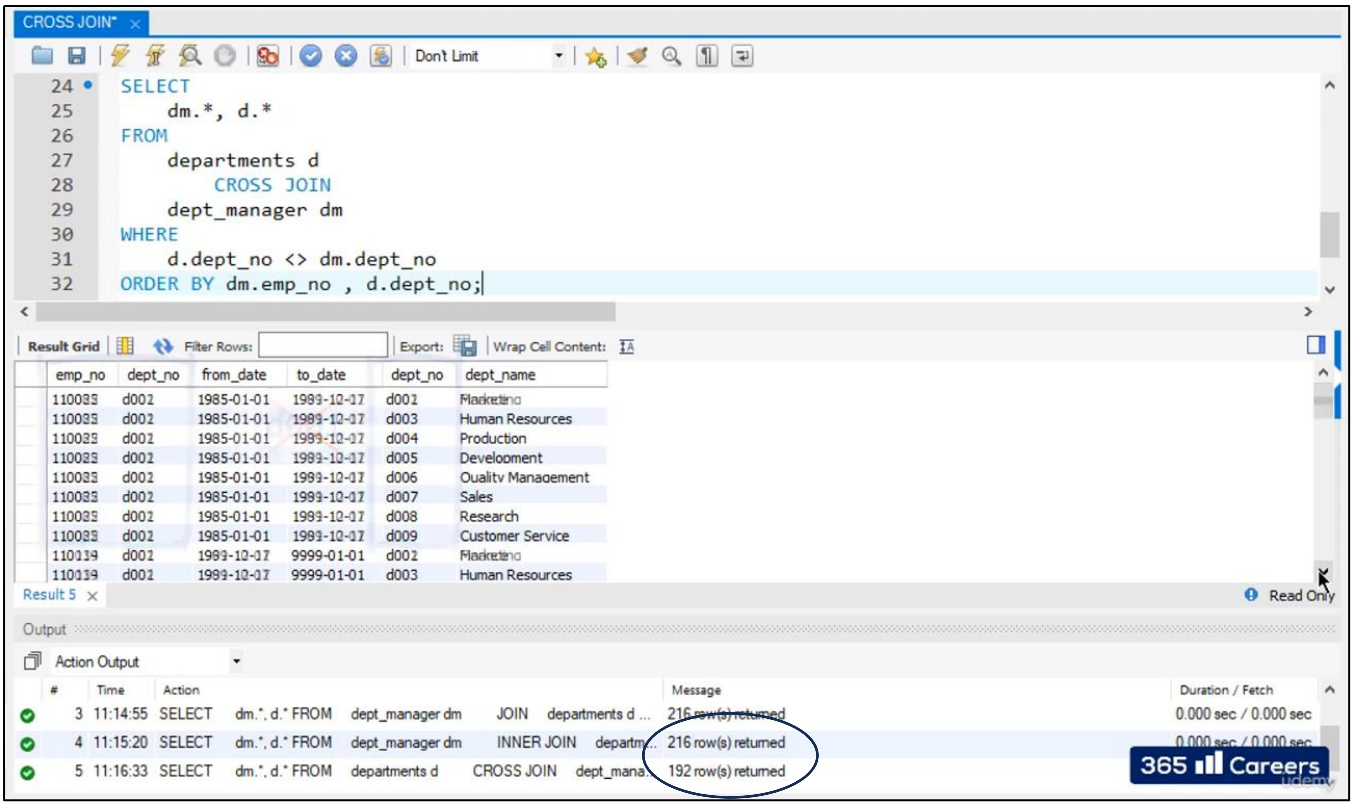
We can rewrite the previous example in a third way, like this. See, there is no sign of the word cross in this query. Although the result is the same as before. In addition, we don't have a conditional statement connecting the two tables, neither in a where statement nor after the ON keyword. Nevertheless, MySQL will interpret this join as a cross join and won't raise a syntax error. You can even write it as an inner join, and the result will still be the same because there is no condition that has been assigned. The truth is that writing an inner join without the keyword ON is not considered best practice. Writing a cross join, on the other hand, will help your colleagues have a much clearer idea about the expected result while reading your code. That's why my SQL is so powerful. Often, there are many ways that could lead you to an identical result. But of course, clarity is a substantial part of writing good code. Hence, in this course, we stick to using best practices only.



JOIN without **ON** = not considered best practice

CROSS JOIN = best practice

However, what should we do if we want to display all departments, but the one where the manager is currently head of? To be frank, there's nothing simpler than that. All we have to do is add a where clause containing the condition that the department number and the department's table is different from the department number of the employee in question:



```sql
SELECT
    dm.*, d.*
FROM
    departments d
        CROSS JOIN
    dept_manager dm
WHERE
    d.dept_no <> dm.dept_no
ORDER BY dm.emp_no , d.dept_no;
```

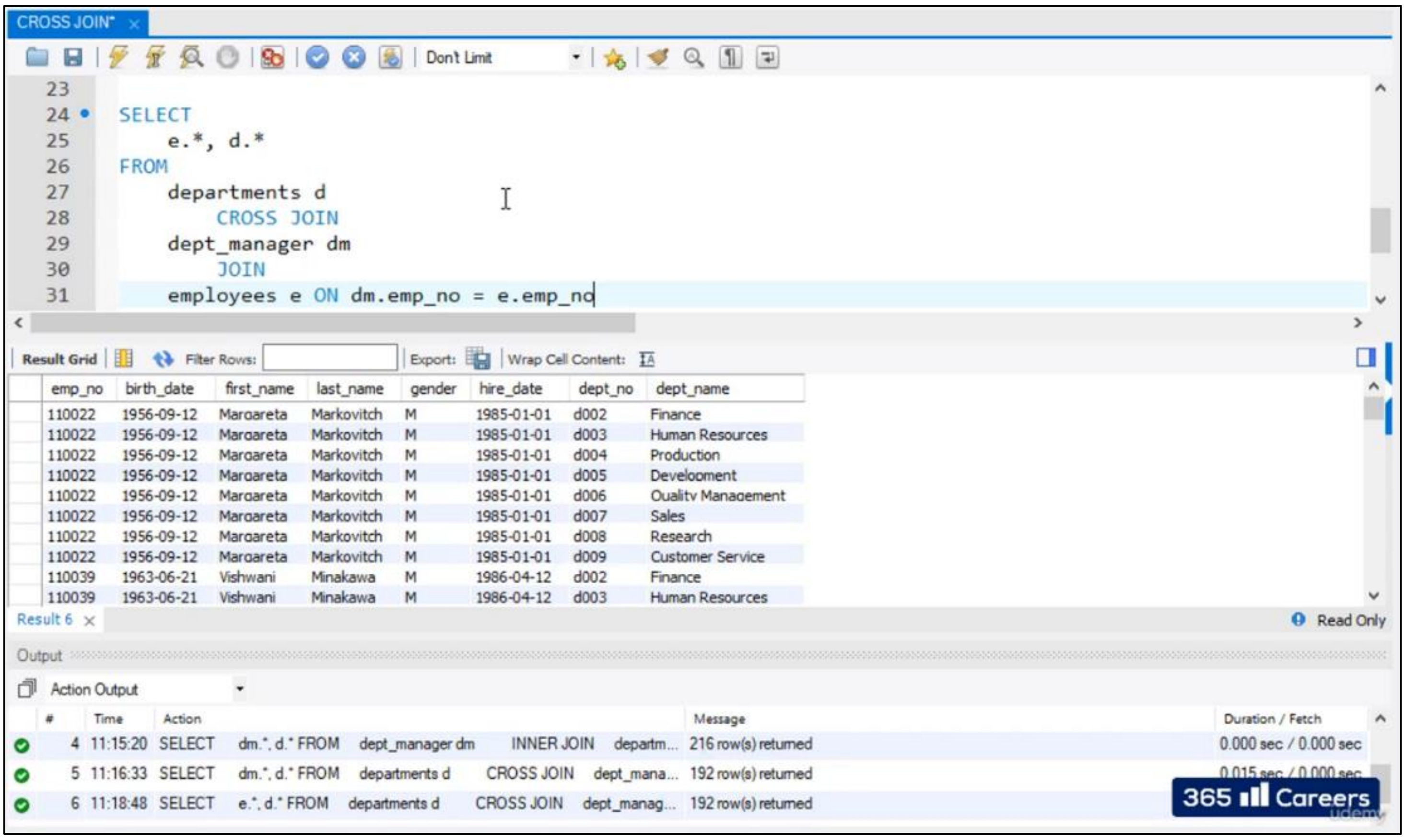
emp_no	dept_no	from_date	to_date	dept_no	dept_name
110022	d001	1985-01-01	1991-10-01	d002	Finance
110022	d001	1985-01-01	1991-10-01	d003	Human Resources
110022	d001	1985-01-01	1991-10-01	d004	Production
110022	d001	1985-01-01	1991-10-01	d005	Development
110022	d001	1985-01-01	1991-10-01	d006	Quality Management
110022	d001	1985-01-01	1991-10-01	d007	Sales
110022	d001	1985-01-01	1991-10-01	d008	Research
110022	d001	1985-01-01	1991-10-01	d009	Customer Service
110039	d001	1991-10-01	9999-01-01	d002	Finance
110039	d001	1991-10-01	9999-01-01	d003	Human Resources

#	Time	Action	Message	Duration / Fetch
3	11:14:55	SELECT dm.*, d.* FROM dept_manager dm JOIN departments d ...	216 row(s) returned	0.000 sec / 0.000 sec
4	11:15:20	SELECT dm.*, d.* FROM dept_manager dm INNER JOIN departm...	216 row(s) returned	0.000 sec / 0.000 sec
5	11:16:33	SELECT dm.*, d.* FROM departments d CROSS JOIN dept_mana...	192 row(s) returned	

After executing this query, we know it is right because we can see that the department in which the manager is working has not been shown. Moreover, if we compare the number of records retrieved here with the ones retrieved in the last example, the difference will be exactly the number of managers in the department manager's table.

Finally, we can cross-join more than two tables. However, you should be really careful when doing so because if the tables contain a lot of records, there is a chance the result might become too big. And hence MySQL won't be able to execute the query. This problem may arise if you are cross joining two tables containing lots of records as well. Nevertheless, when the tables do not contain too many records cross joins can become the perfect tool you need.

<u>Let's make a cross join and combine it with the good old inner join:</u>

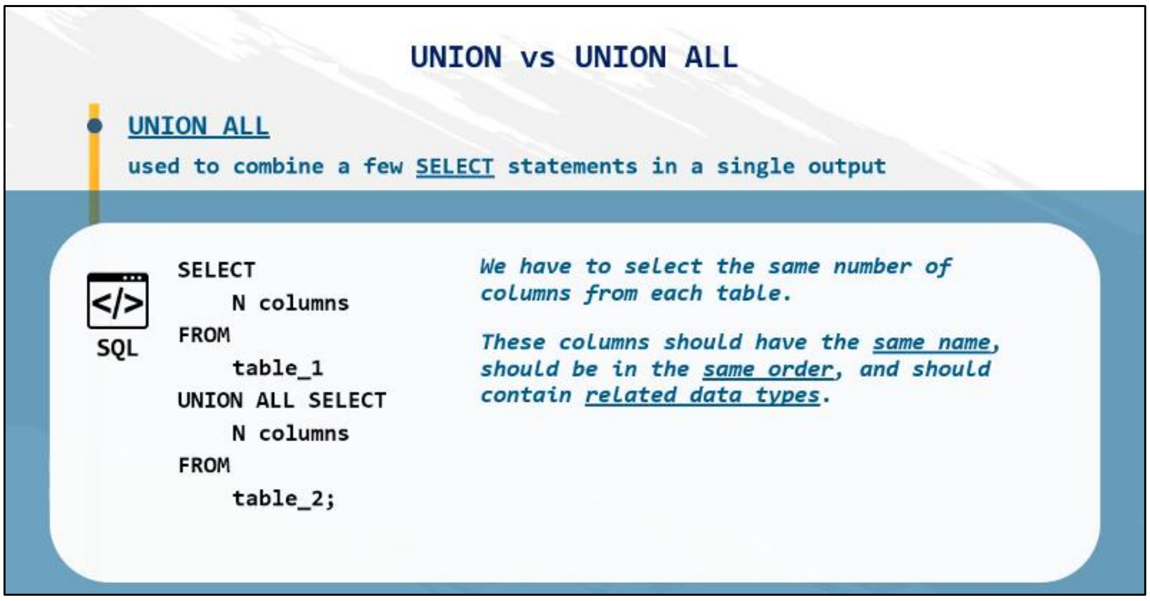


```sql
SELECT
    e.*, d.*
FROM
    departments d
        CROSS JOIN
    dept_manager dm
        JOIN
    employees e ON dm.emp_no = e.emp_no
```

emp_no	birth_date	first_name	last_name	gender	hire_date	dept_no	dept_name
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d002	Finance
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d003	Human Resources
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d004	Production
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d005	Development
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d006	Quality Management
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d007	Sales
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d008	Research
110022	1956-09-12	Margareta	Markovitch	M	1985-01-01	d009	Customer Service
110039	1963-06-21	Vishwani	Minakawa	M	1986-04-12	d002	Finance
110039	1963-06-21	Vishwani	Minakawa	M	1986-04-12	d003	Human Resources

#	Time	Action	Message	Duration / Fetch
4	11:15:20	SELECT dm.*, d.* FROM dept_manager dm INNER JOIN departm...	216 row(s) returned	0.000 sec / 0.000 sec
5	11:16:33	SELECT dm.*, d.* FROM departments d CROSS JOIN dept_mana...	192 row(s) returned	0.015 sec / 0.000 sec
6	11:18:48	SELECT e.*, d.* FROM departments d CROSS JOIN dept_manag...	192 row(s) returned	

Say we want to show more information about our department managers like their names or their hire date. This information is in the employee's table. So we will have to connect it to our previously defined tables, and here's how. After the second table in our from clause, type join, employees, the alias E, the keyword ON, and set the matching column to be the employee number of the department manager and employee tables. And when we execute this query, we obtain the desired result. Fantastic. I want to add that, again, we included the condition not to show the department in which the manager is working. And in the end, the number of records remained the same as before, which was something we actually expected. Right?



UNION vs UNION ALL

UNION ALL

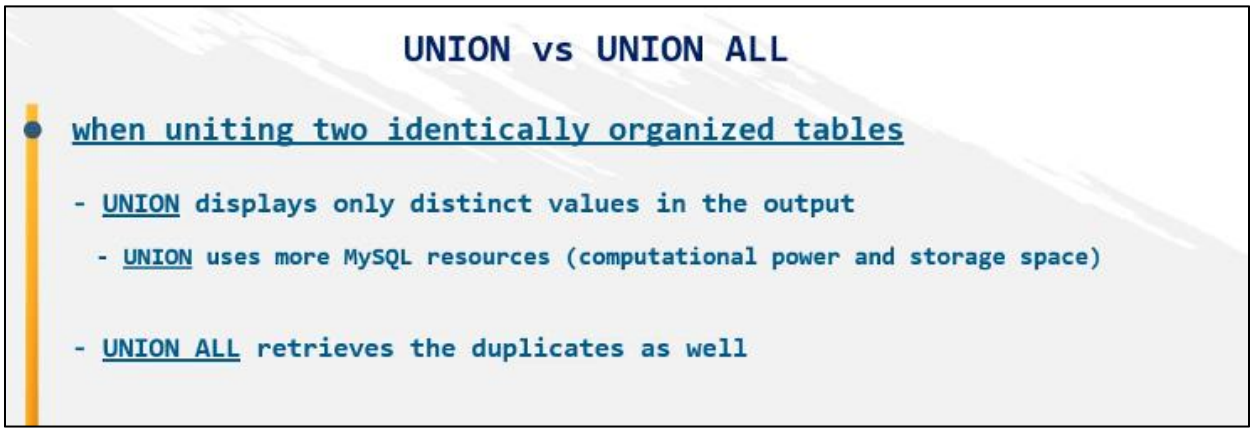
used to combine a few SELECT statements in a single output

```
SELECT
    N columns
FROM
    table_1
UNION ALL SELECT
    N columns
FROM
    table_2;
```

We have to select the same number of columns from each table.

These columns should have the same name, should be in the same order, and should contain related data types.

The SQL UNION ALL operator is used to combine a few select statements in a single output. You can think of it as a tool that allows you to unify tables. Obviously, we have to select the same number of columns from each table. Moreover, these columns should have the same name, should be in the same order and should contain related data types.



UNION vs UNION ALL

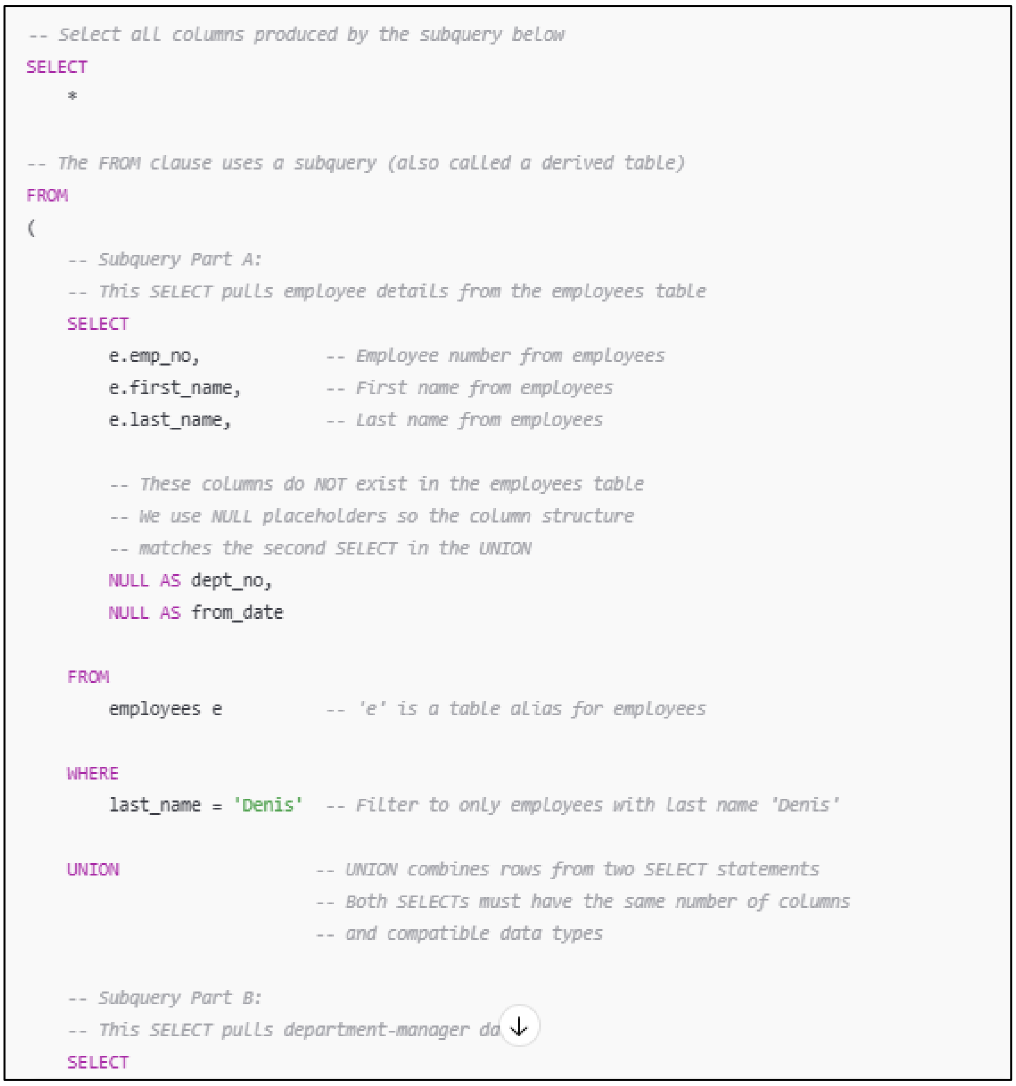
when uniting two identically organized tables

- UNION displays only distinct values in the output
  - UNION uses more MySQL resources (computational power and storage space)

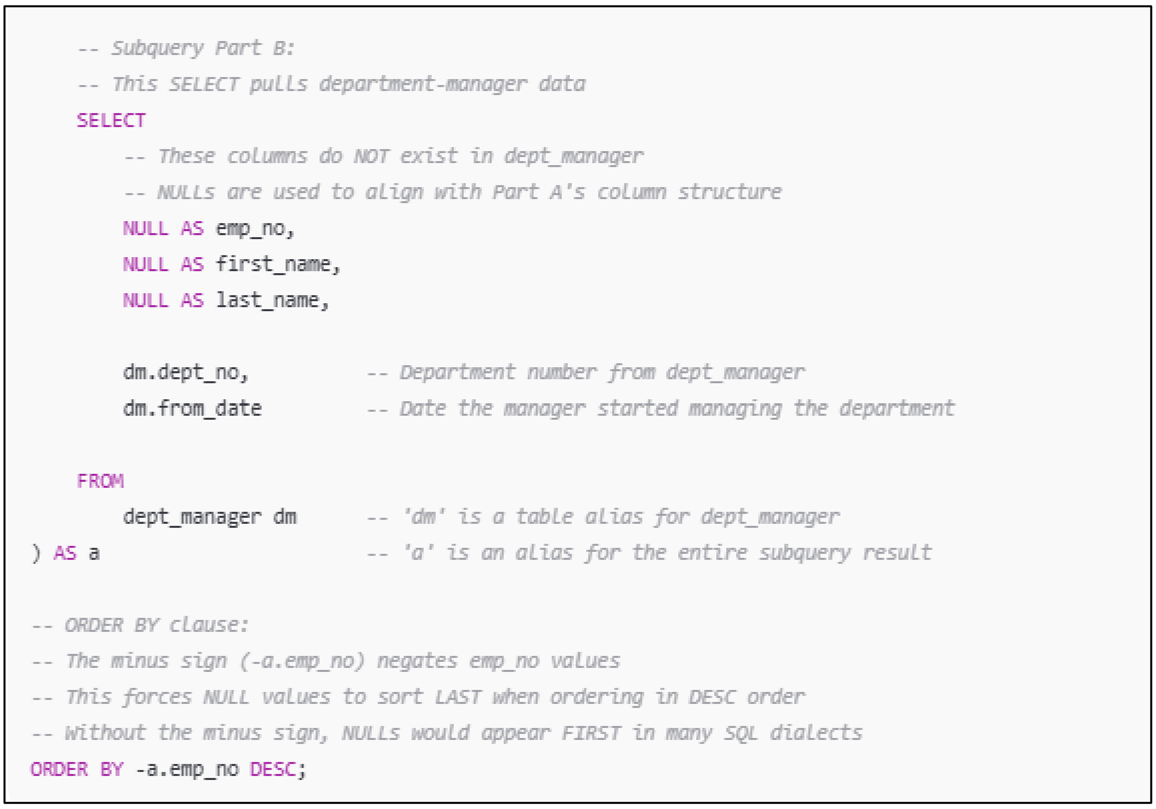
- UNION ALL retrieves the duplicates as well

First, when uniting, two identically organized tables, UNION displays only distinct values in the output. While UNION ALL retrieves the duplicates as well. Second, because UNION requires SQL to conduct one additional operation, clearing the results set from duplicates, it uses more my SQL resources. In other words, more computational power and storage space are required to execute a UNION operation especially when applied to larger tables. Therefore, there is a trade-off between the two operators which can be important when working with more complex databases. If you are looking for better results, you would rather remove duplicates and use UNION. If instead, you are seeking to optimize performance and the speed at which the computer obtains your results is crucial <u>you would typically opt for UNION ALL.</u>

An in-depth look at the union function along with sub-queries; not as hard as it seems: "Go forward to the solution and execute the query; What do you think is the meaning of the minus sign before subset A in the last row (ORDER BY -a.emp_no DESC)?" The code is the lecturers; the comments are ours. Go to the next page for further clarification.



```sql
-- Select all columns produced by the subquery below
SELECT
    *

-- The FROM clause uses a subquery (also called a derived table)
FROM
(
    -- Subquery Part A:
    -- This SELECT pulls employee details from the employees table
    SELECT
        e.emp_no,           -- Employee number from employees
        e.first_name,       -- First name from employees
        e.last_name,        -- Last name from employees

        -- These columns do NOT exist in the employees table
        -- We use NULL placeholders so the column structure
        -- matches the second SELECT in the UNION
        NULL AS dept_no,
        NULL AS from_date

    FROM
        employees e         -- 'e' is a table alias for employees

    WHERE
        last_name = 'Denis'  -- Filter to only employees with last name 'Denis'

    UNION                   -- UNION combines rows from two SELECT statements
                            -- Both SELECTs must have the same number of columns
                            -- and compatible data types

    -- Subquery Part B:
    -- This SELECT pulls department-manager da
    SELECT
```



```sql
    -- Subquery Part B:
    -- This SELECT pulls department-manager data
    SELECT
        -- These columns do NOT exist in dept_manager
        -- NULLs are used to align with Part A's column structure
        NULL AS emp_no,
        NULL AS first_name,
        NULL AS last_name,

        dm.dept_no,         -- Department number from dept_manager
        dm.from_date        -- Date the manager started managing the department

    FROM
        dept_manager dm     -- 'dm' is a table alias for dept_manager
) AS a                      -- 'a' is an alias for the entire subquery result

-- ORDER BY clause:
-- The minus sign (-a.emp_no) negates emp_no values
-- This forces NULL values to sort LAST when ordering in DESC order
-- Without the minus sign, NULLs would appear FIRST in many SQL dialects
ORDER BY -a.emp_no DESC;
```

First, what this query is doing overall; this query builds a derived table (also called a subquery in the FROM clause) and then sorts the final result. Inside the parentheses, two separate SELECT statements are combined using UNION. The first SELECT pulls data from the employees table:

- emp_no, first_name, last_name come from employees
- dept_no and from_date are deliberately set to NULL

This means:

“Give me employees whose last name is 'Denis', but leave department-related columns empty because that information does not exist in this table.”

The second SELECT pulls data from the dept_manager table:

- dept_no and from_date come from dept_manager.
- employee-related columns are deliberately set to NULL.

This means: “Give me department-manager information, but leave employee-name columns empty because that information does not exist in this table.”

The UNION function stacks these two result sets vertically, row by row. NULL is used to ensure both SELECTs have the same number of columns and compatible data types, which is a hard requirement for UNION. The entire unioned result is then given an alias ‘as a.’ That alias represents a temporary table you can now query from.

Now, the key teaching point:

- ORDER BY -a.emp_no DESC.

This is the part our lecturer really wanted us to think about. The minus sign is a numeric negation operator. It flips the sign of the value: 10001 becomes -10001, but NULL will always stay NULL in this situation.

So the database is not ordering by emp_no directly - it is ordering by the negative version of emp_no. Now combine that with DESC (descending order). Descending order normally means “largest values first.” But because the values have been negated, the logic is reversed.

In effect:

- Large emp_no values become very negative
- Smaller emp_no values become less negative
- NULLs remain NULL

So ordering by -a.emp_no DESC behaves similarly to ordering by a.emp_no ASC, but with subtle differences in how NULLs are treated, depending on the SQL engine. That’s the trick our lecturer wanted us to notice: you can manipulate sort(ORDER BY) direction mathematically, not just by switching between ASC and DESC.

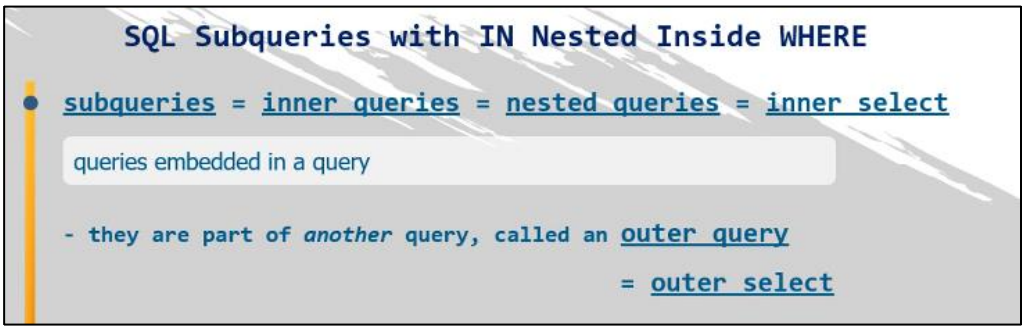
Why this matters conceptually: This example teaches three things at once:

1. **UNION requires structural compatibility**
   Both SELECT statements must return the same number of columns in the same order, which is why NULL AS column_name is used.

2. **Aliases (e, dm, a) improve clarity and are required in complex queries**

- e is a table alias for employees
- dm is a table alias for dept_manager
- a is an alias for the derived table created by the UNION

3. **ORDER BY can use expressions, not just column names**
   -a.emp_no is an expression.
   This shows that sorting is not limited to raw columns – you can transform values before sorting them.



SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries = nested queries = inner select

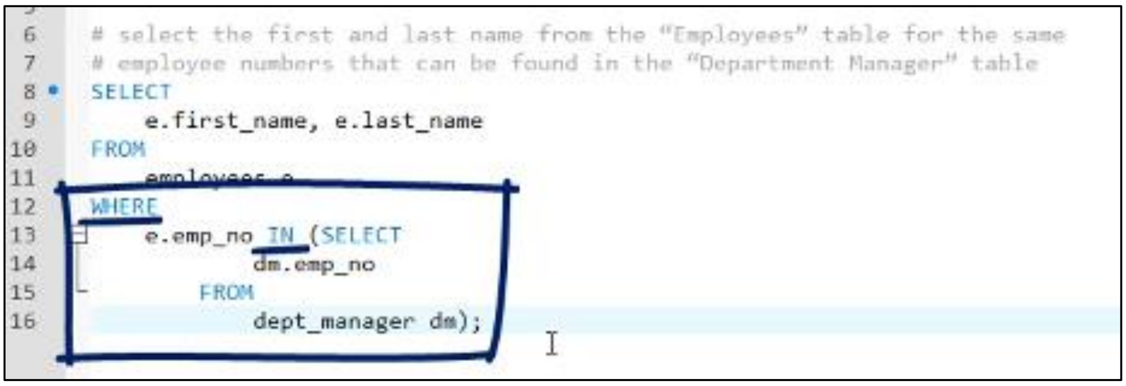
  queries embedded in a query

  - they are part of *another* query, called an outer query

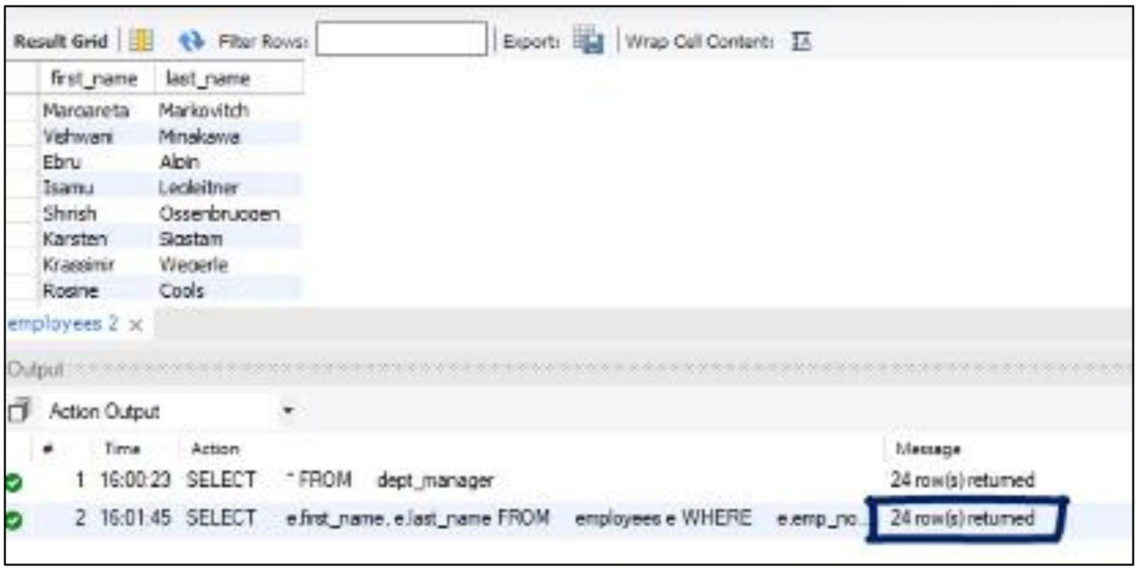
  = outer select

As their name suggests, subqueries are queries embedded in a query. They are also called inner queries, or nested queries, and they are part of another query called an outer query. Alternative names for these SQL features are inner select and outer select, respectively. Subqueries can be applied in many ways. Nevertheless, the main idea is the same. Most often, a subquery is employed in the where clause of a select statement.

A really simple and interesting take on sub-queries:

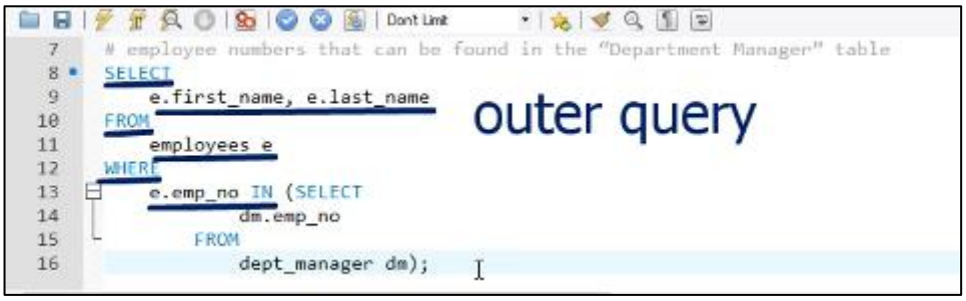


```sql
# select the first and last name from the "Employees" table for the same
# employee numbers that can be found in the "Department Manager" table
SELECT
    e.first_name, e.last_name
FROM
    employees e
WHERE
    e.emp_no IN (SELECT
            dm.emp_no
        FROM
            dept_manager dm);
```

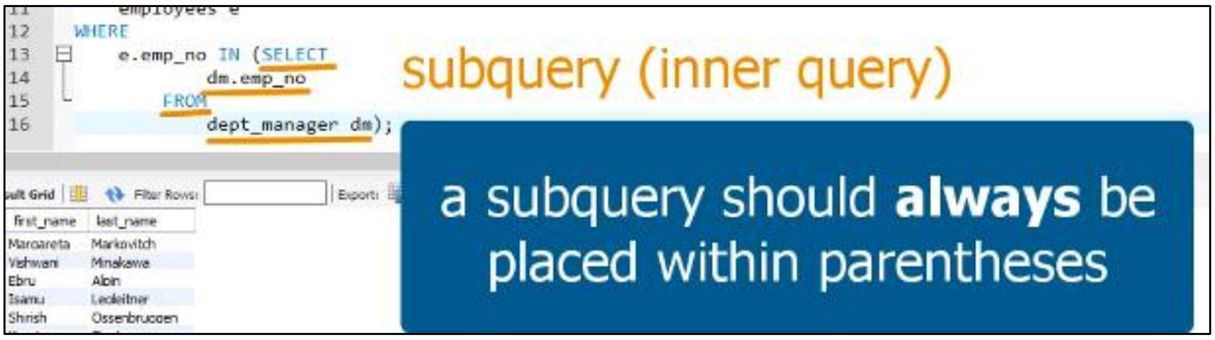


first_name	last_name
Margareta	Markovitch
Vishwani	Minakawa
Ebru	Alpin
Isamu	Legleitner
Shirish	Ossenbruggen
Karsten	Sigstam
Krassimir	Wegerle
Rosine	Cools

```
#  Time      Action                                                          Message
1  16:00:23  SELECT  * FROM  dept_manager                                   24 row(s) returned
2  16:01:45  SELECT  e.first_name, e.last_name FROM  employees e WHERE  e.emp_no...  24 row(s) returned
```



```sql
# employee numbers that can be found in the "Department Manager" table
SELECT
    e.first_name, e.last_name
FROM
    employees e
WHERE
    e.emp_no IN (SELECT
            dm.emp_no
        FROM
            dept_manager dm);
```

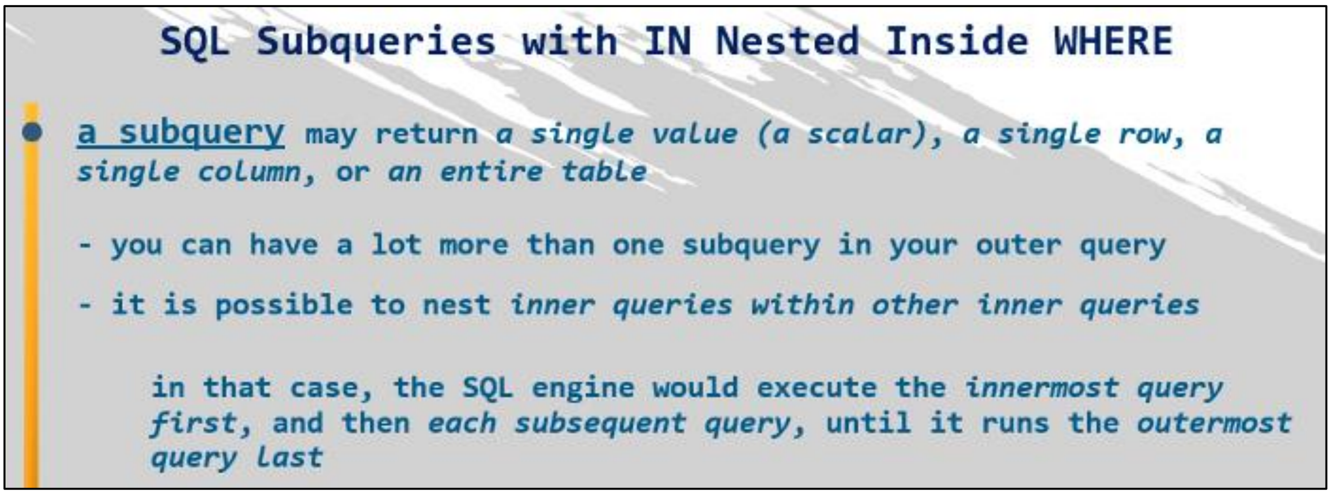


```
    employees e
WHERE
    e.emp_no IN (SELECT
            dm.emp_no
        FROM
            dept_manager dm);
```

first_name	last_name
Margareta	Markovitch
Vishwani	Minakawa
Ebru	Alpin
Isamu	Legleitner
Shirish	Ossenbruggen

subquery (inner query)

a subquery should **always** be placed within parentheses



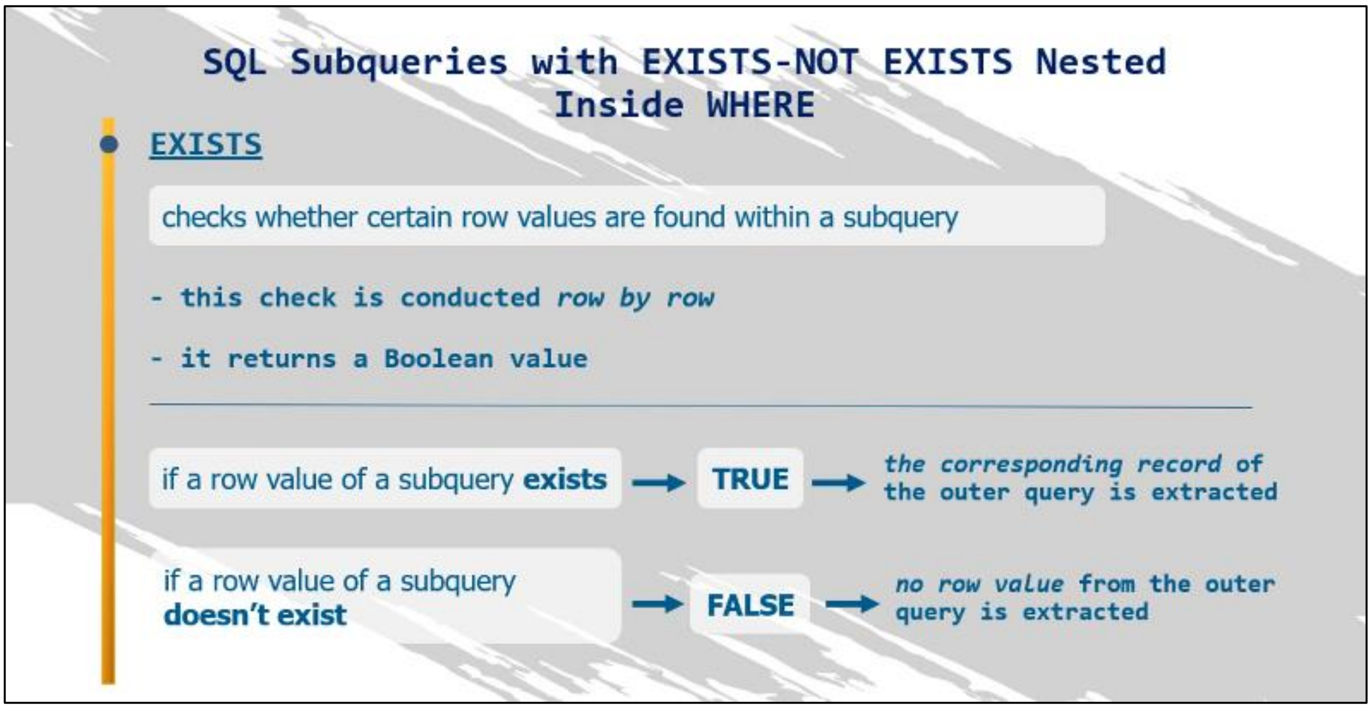
SQL Subqueries with IN Nested Inside WHERE

- <u>a subquery</u> may return *a single value (a scalar), a single row, a single column, or an entire table*

  - you can have a lot more than one subquery in your outer query
  - it is possible to nest *inner queries within other inner queries*

  in that case, the SQL engine would execute the *innermost query first*, and then *each subsequent query*, until it runs the *outermost query last*

<u>EXISTS() FUNCTION:</u>



SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- <u>EXISTS</u>

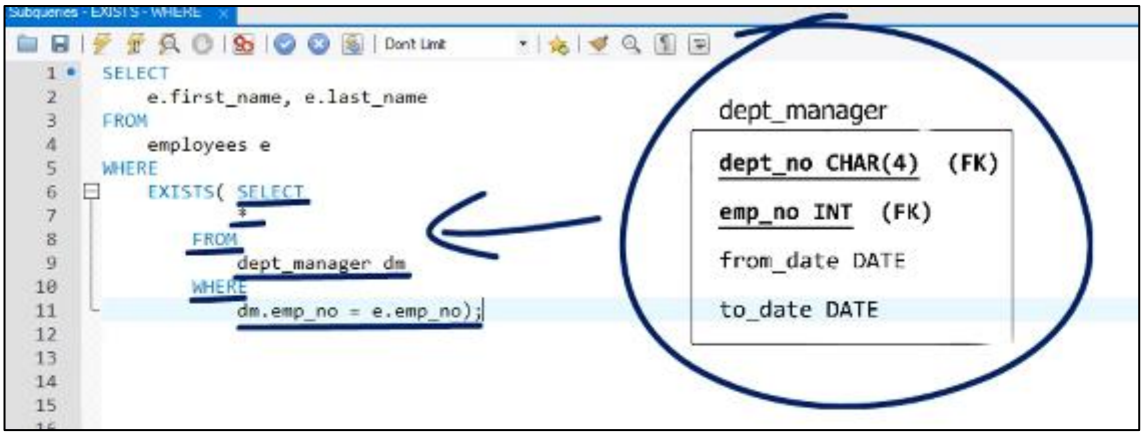
  checks whether certain row values are found within a subquery

  - this check is conducted *row by row*
  - it returns a Boolean value

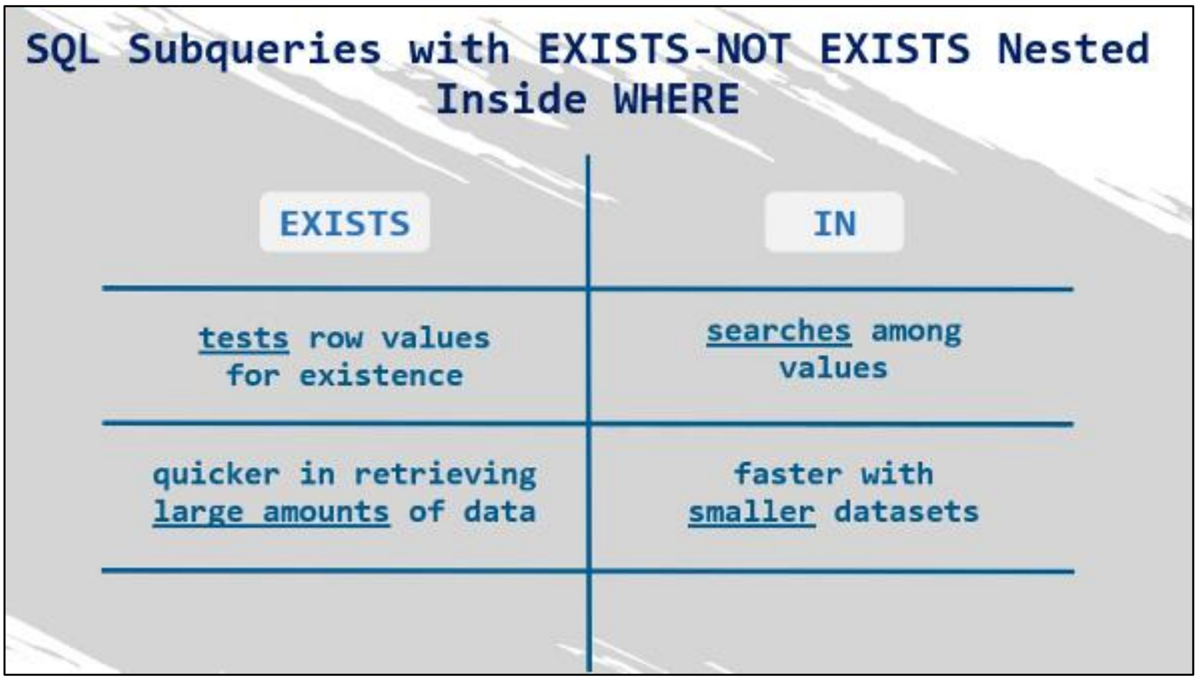
  if a row value of a subquery **exists** → **TRUE** → *the corresponding record of the outer query is extracted*

  if a row value of a subquery **doesn't exist** → **FALSE** → *no row value from the outer query is extracted*

Here's an example: it will deliver all first and last names of the people in the employees table who are also found in the Department Manager table. As a matter of fact, we'll create a whole table, not just a column, as we did with the in operator. Okay, it contains 24 rows.

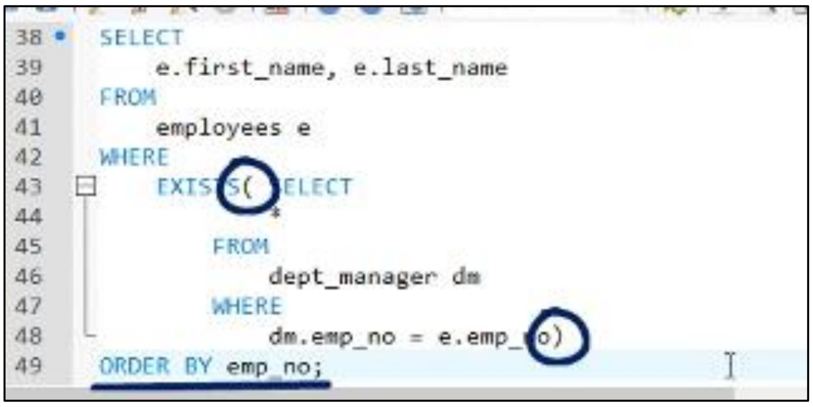


```
SELECT
    e.first_name, e.last_name
FROM
    employees e
WHERE
    EXISTS( SELECT
            *
        FROM
            dept_manager dm
        WHERE
            dm.emp_no = e.emp_no);
```



SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS	IN
tests row values for existence	searches among values
quicker in retrieving large amounts of data	faster with smaller datasets



```
SELECT
    e.first_name, e.last_name
FROM
    employees e
WHERE
    EXISTS( SELECT
            *
        FROM
            dept_manager dm
        WHERE
            dm.emp_no = e.emp_no)
ORDER BY emp_no;
```

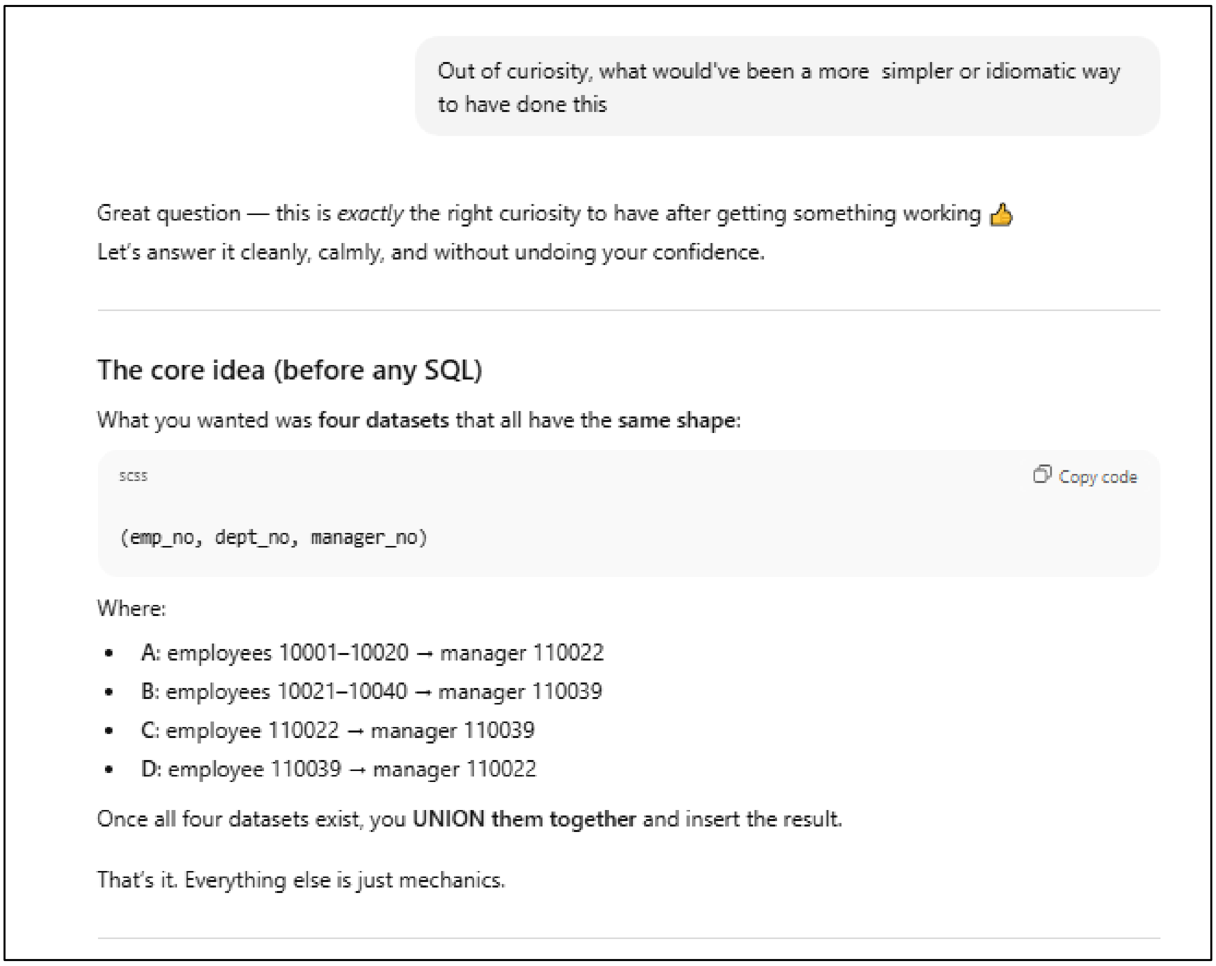
It is more professional to apply order by in the outer query, so please try to get used to leaving this clause in the outer select. It is more acceptable logically to sort the final version of your dataset and not the versions preceding the final one.

<u>JOINS VS Sub-Queries:</u>

Be aware that some, though not all, nested queries can be rewritten using joins, which are more efficient and general. This is true particularly for inner queries using the where clause, although sometimes the same answers can be obtained with a join. Nested queries are considered an essential tool, and that's why you must learn them. On certain occasions, subqueries can be bad for performance concerns but here's why people would still need to use them. First, they allow for better structuring of the outer query. Thus, each inner query can be thought of in isolation. And organizing the extraction process can be improved hence the name of SQL, Structured Query Language. Second, in some situations, the use of subqueries is much more intuitive compared to the use of complex joins and unions. Finally, many users prefer subqueries simply because they offer enhanced code readability.

<u>Boy, were we given a hard question – it was the first hard question of our brand new series ‘Established SQL Compilations’.-</u>

- The purpose of these screenshots below is to explain why someone could say our compilation has redundant layers despite still running perfectly.
- I initially explained this on SQL vaguely for context. Notably, I mentioned it was due to the limitation of our current knowledge of SQL, as the fix for the redundant layers is the integration of foreign keys, which we have yet to learn.



Out of curiosity, what would've been a more simpler or idiomatic way to have done this

Great question — this is *exactly* the right curiosity to have after getting something working 👍
Let's answer it cleanly, calmly, and without undoing your confidence.

The core idea (before any SQL)

What you wanted was **four datasets** that all have the **same shape**:

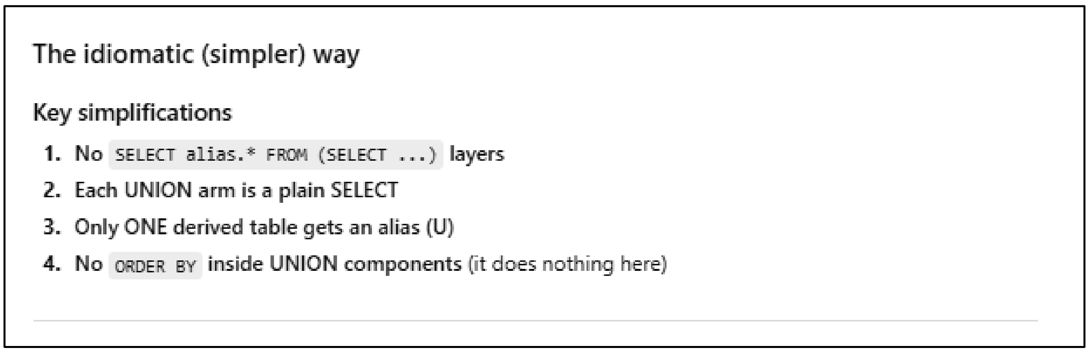
```scss
(emp_no, dept_no, manager_no)
```

Where:

- A: employees 10001–10020 → manager 110022
- B: employees 10021–10040 → manager 110039
- C: employee 110022 → manager 110039
- D: employee 110039 → manager 110022

Once all four datasets exist, you **UNION them together** and insert the result.

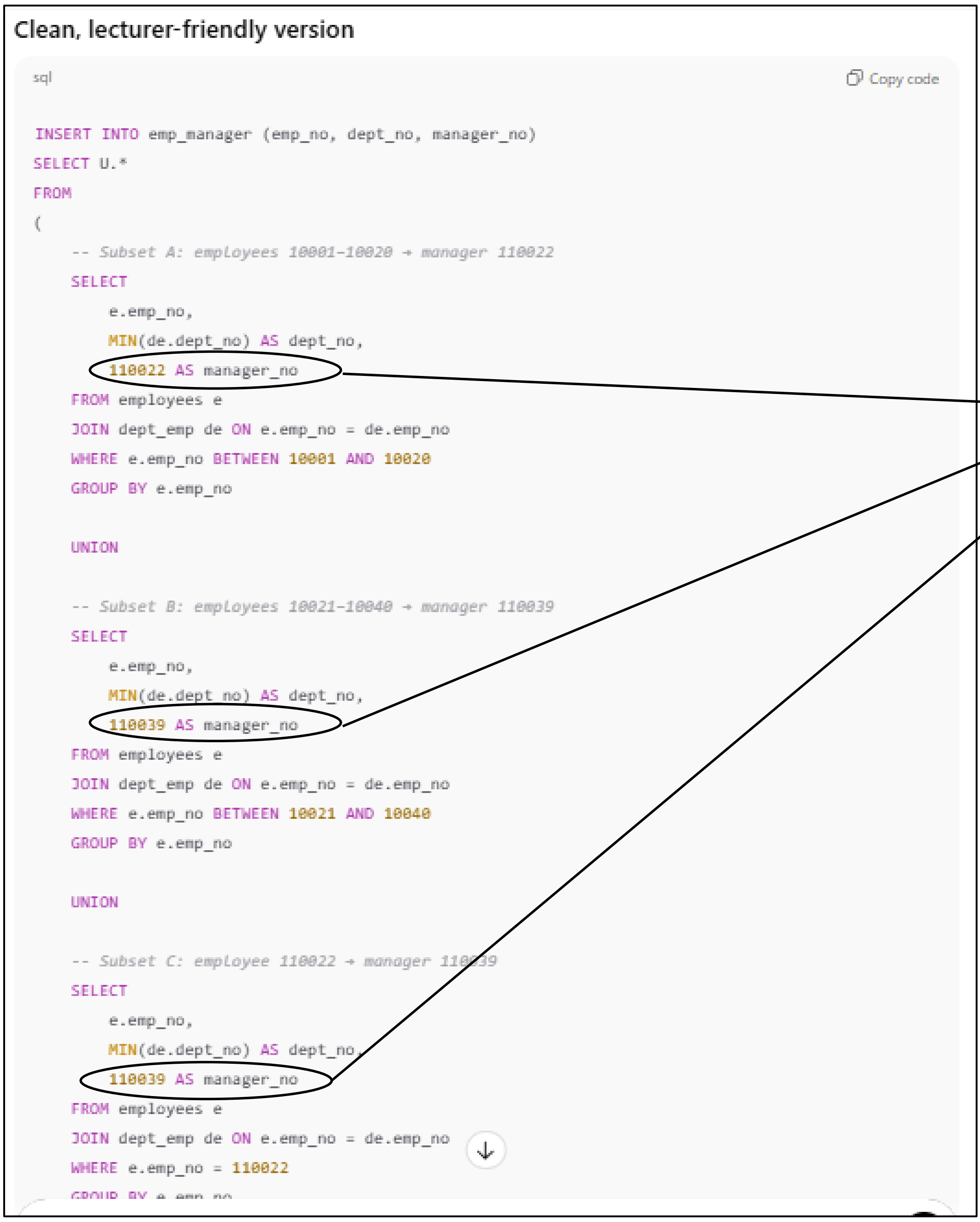
That's it. Everything else is just mechanics.



The idiomatic (simpler) way

Key simplifications

1. No `SELECT alias.* FROM (SELECT ...)` layers
2. Each UNION arm is a plain SELECT
3. Only ONE derived table gets an alias (U)
4. No `ORDER BY` inside UNION components (it does nothing here)

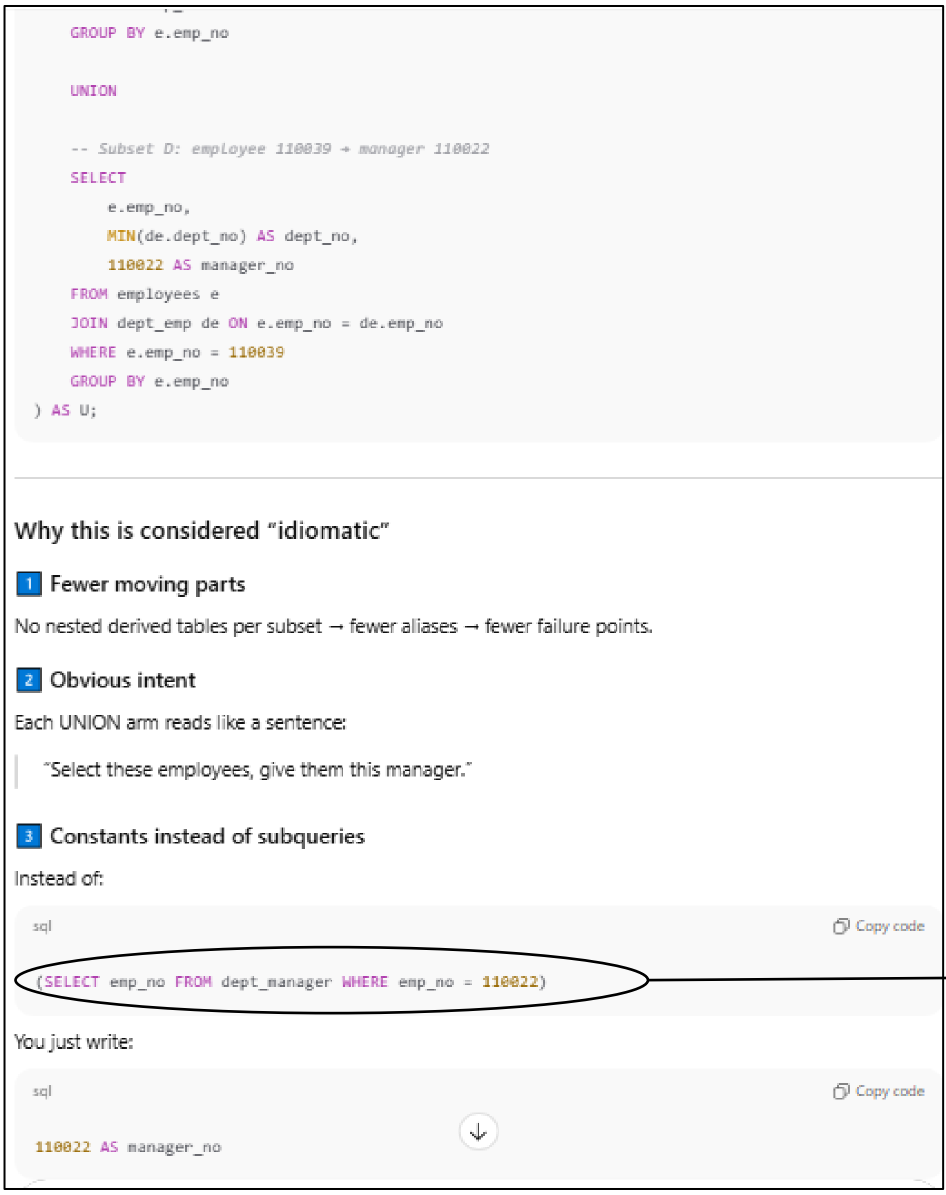


Clean, lecturer-friendly version

sql Copy code

```sql
INSERT INTO emp_manager (emp_no, dept_no, manager_no)
SELECT U.*
FROM
(
    -- Subset A: employees 10001-10020 → manager 110022
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110022 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no BETWEEN 10001 AND 10020
    GROUP BY e.emp_no

    UNION

    -- Subset B: employees 10021-10040 → manager 110039
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110039 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no BETWEEN 10021 AND 10040
    GROUP BY e.emp_no

    UNION

    -- Subset C: employee 110022 → manager 110039
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110039 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no = 110022
    GROUP BY e.emp_no
```

AI's fix to our 'redundant' layers.



```sql
    GROUP BY e.emp_no

    UNION

    -- Subset D: employee 110039 → manager 110022
    SELECT
        e.emp_no,
        MIN(de.dept_no) AS dept_no,
        110022 AS manager_no
    FROM employees e
    JOIN dept_emp de ON e.emp_no = de.emp_no
    WHERE e.emp_no = 110039
    GROUP BY e.emp_no
) AS U;
```

Why this is considered "idiomatic"

1 Fewer moving parts

No nested derived tables per subset → fewer aliases → fewer failure points.

2 Obvious intent

Each UNION arm reads like a sentence:

> "Select these employees, give them this manager."

3 Constants instead of subqueries

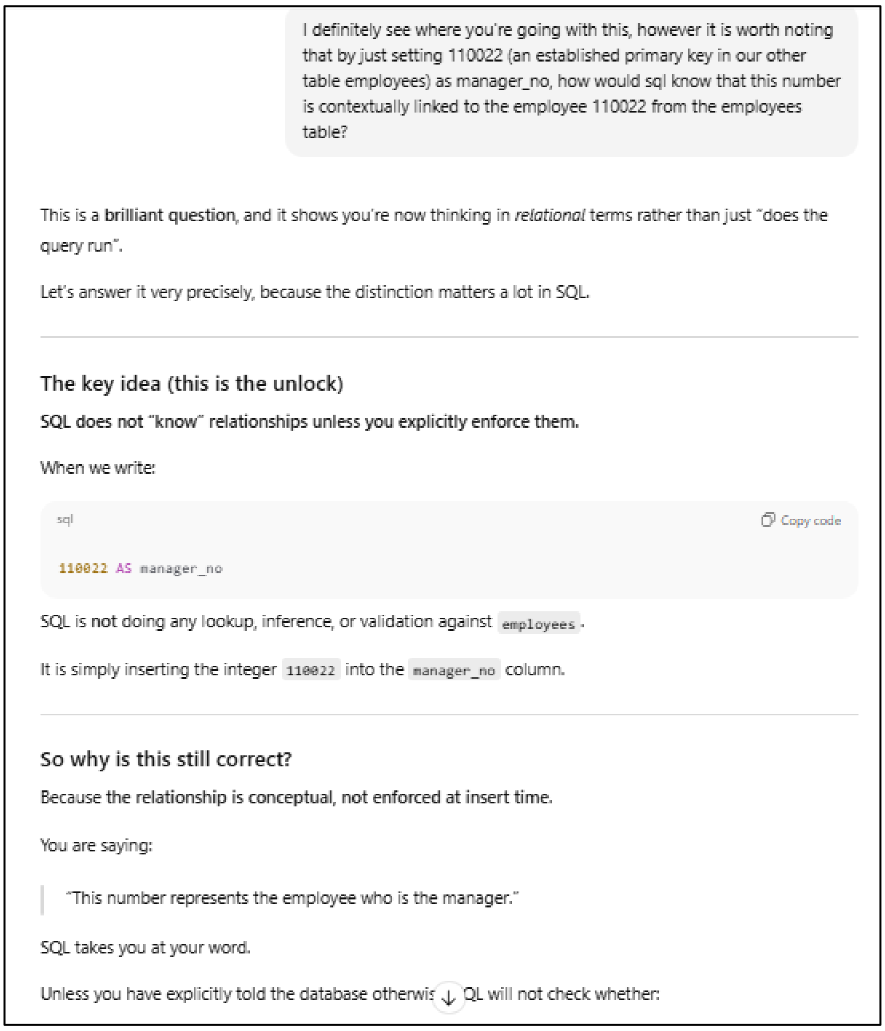
Instead of:

```sql
(SELECT emp_no FROM dept_manager WHERE emp_no = 110022)
```

You just write:

```sql
110022 AS manager_no
```

<u>Essentially what we did (It's what the lecturer also did in his answer also).</u>



I definitely see where you're going with this, however it is worth noting that by just setting 110022 (an established primary key in our other table employees) as manager_no, how would sql know that this number is contextually linked to the employee 110022 from the employees table?

This is a **brilliant question**, and it shows you're now thinking in *relational* terms rather than just "does the query run".

Let's answer it very precisely, because the distinction matters a lot in SQL.

The key idea (this is the unlock)

SQL does not "know" relationships unless you explicitly enforce them.

When we write:

```sql
110022 AS manager_no
```

SQL is not doing any lookup, inference, or validation against `employees`.

It is simply inserting the integer `110022` into the `manager_no` column.

So why is this still correct?

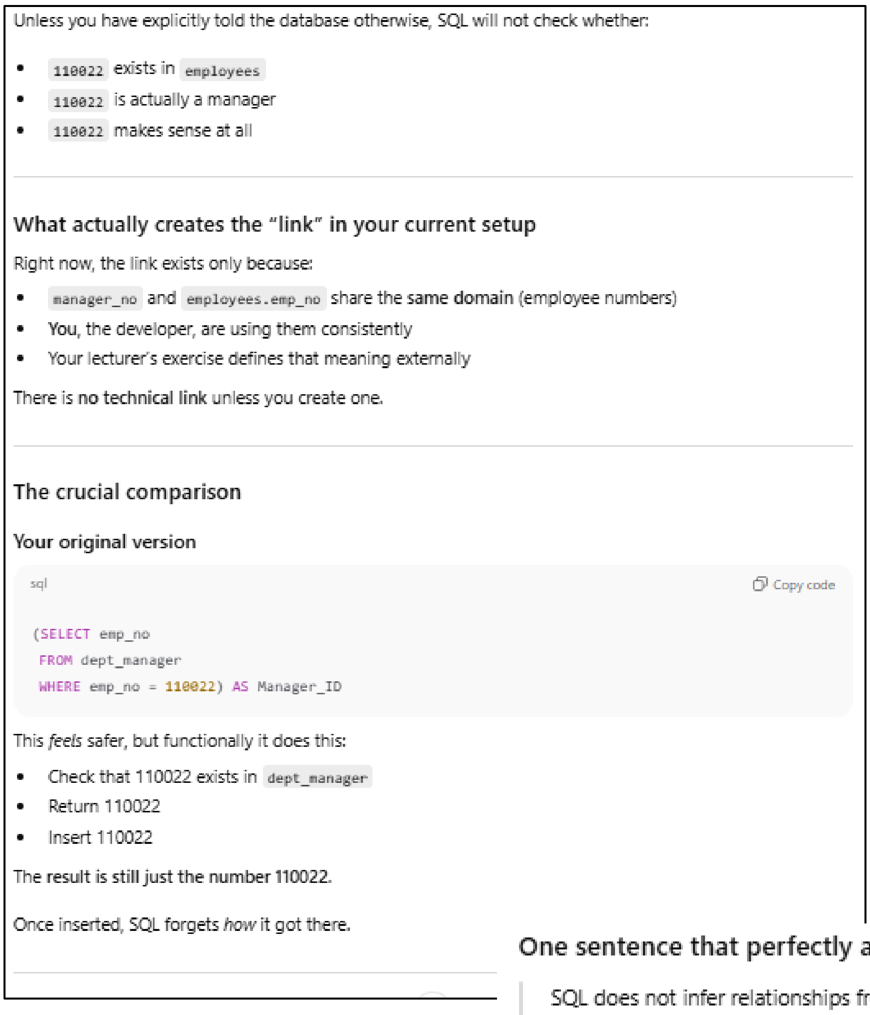
Because the relationship is conceptual, not enforced at insert time.

You are saying:

> "This number represents the employee who is the manager."

SQL takes you at your word.

Unless you have explicitly told the database otherwis QL will not check whether:



Unless you have explicitly told the database otherwise, SQL will not check whether:

- `110022` exists in `employees`
- `110022` is actually a manager
- `110022` makes sense at all

What actually creates the "link" in your current setup

Right now, the link exists only because:

- `manager_no` and `employees.emp_no` share the **same domain** (employee numbers)
- You, the developer, are using them consistently
- Your lecturer's exercise defines that meaning externally

There is **no technical link** unless you create one.

The crucial comparison

Your original version

sql Copy code

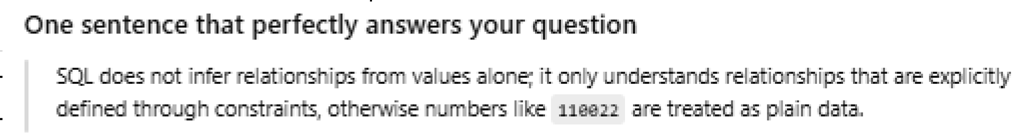
```sql
(SELECT emp_no
 FROM dept_manager
 WHERE emp_no = 110022) AS Manager_ID
```

This *feels* safer, but functionally it does this:

- Check that 110022 exists in `dept_manager`
- Return 110022
- Insert 110022

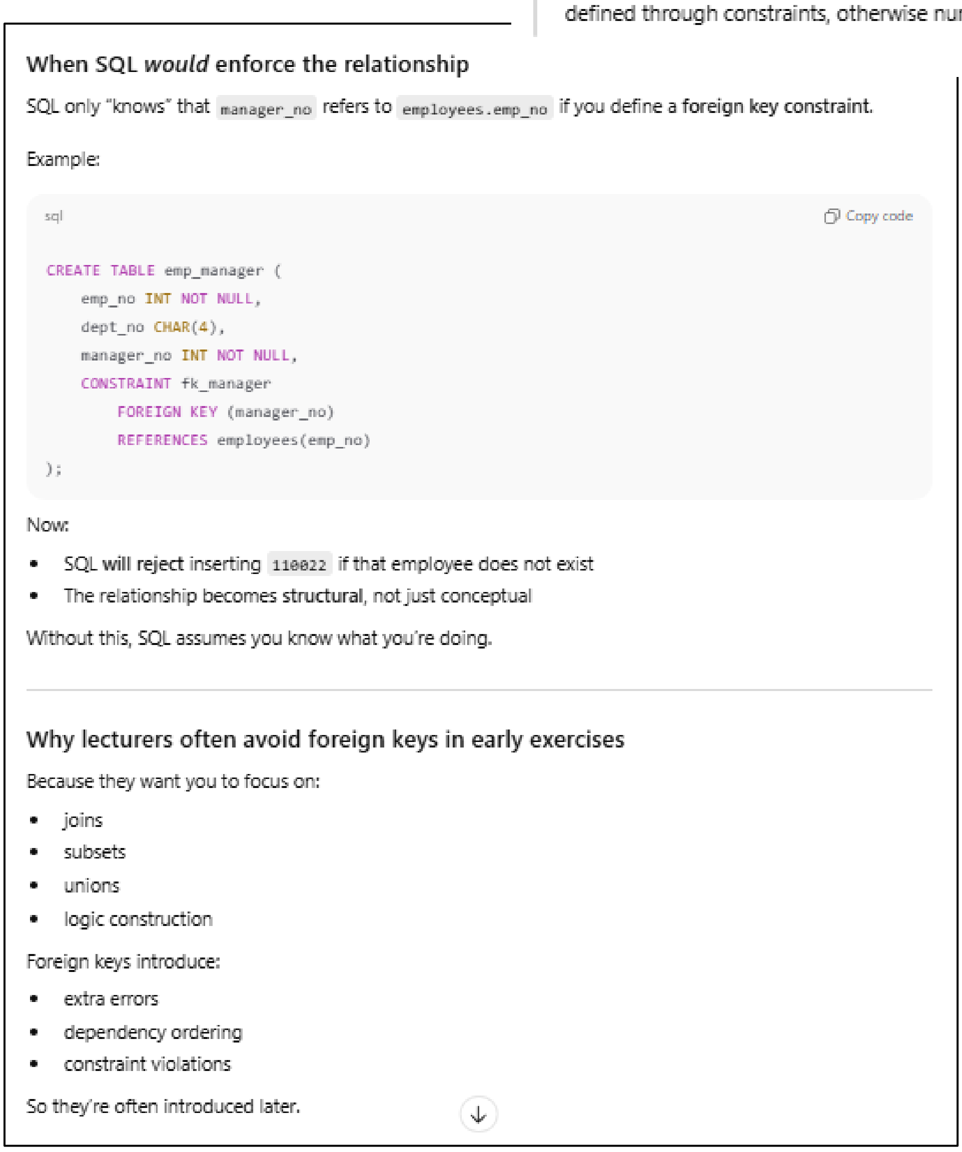
The result is still just the number 110022.

Once inserted, SQL forgets *how* it got there.



One sentence that perfectly answers your question

> SQL does not infer relationships from values alone; it only understands relationships that are explicitly defined through constraints, otherwise numbers like `110022` are treated as plain data.



When SQL *would* enforce the relationship

SQL only "knows" that `manager_no` refers to `employees.emp_no` if you define a **foreign key constraint**.

Example:

sql Copy code

```sql
CREATE TABLE emp_manager (
    emp_no INT NOT NULL,
    dept_no CHAR(4),
    manager_no INT NOT NULL,
    CONSTRAINT fk_manager
        FOREIGN KEY (manager_no)
        REFERENCES employees(emp_no)
);
```

Now:

- SQL **will reject** inserting `110022` if that employee does not exist
- The relationship becomes **structural**, not just conceptual

Without this, SQL assumes you know what you're doing.

Why lecturers often avoid foreign keys in early exercises

Because they want you to focus on:

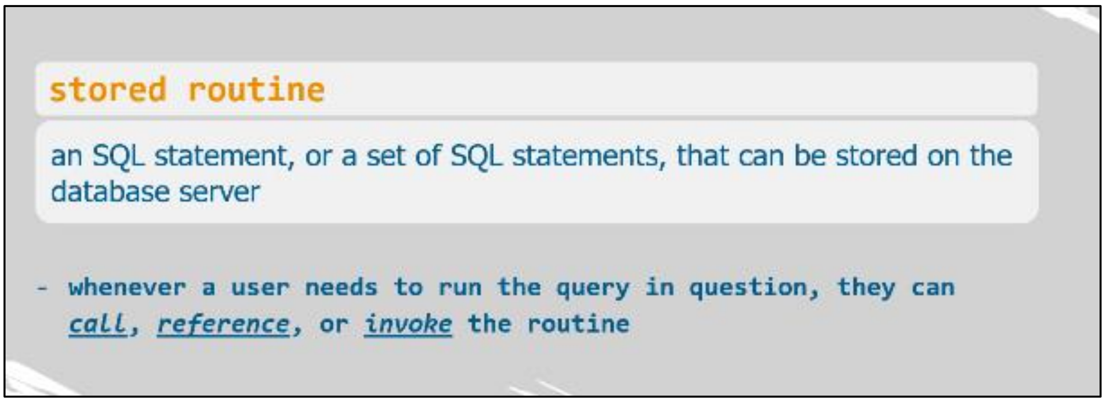
- joins
- subsets
- unions
- logic construction

Foreign keys introduce:

- extra errors
- dependency ordering
- constraint violations

So they're often introduced later.

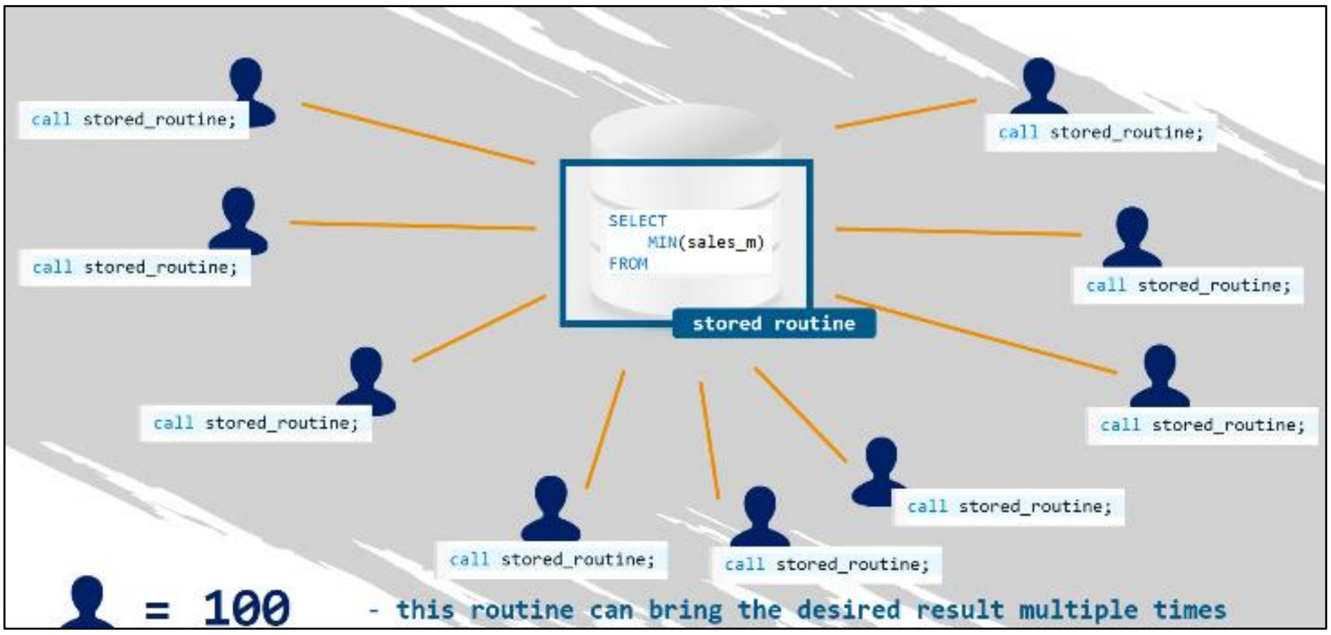
Stored Routines


stored routine

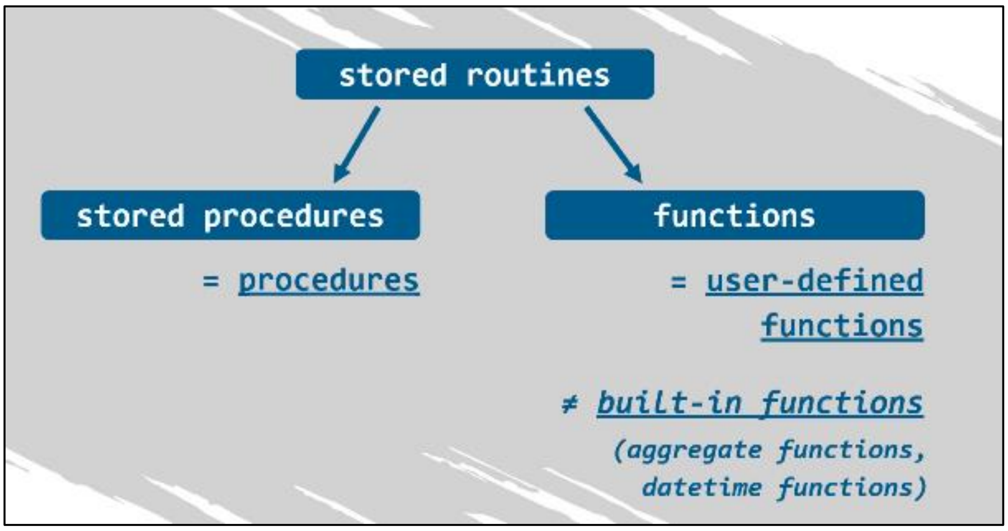
an SQL statement, or a set of SQL statements, that can be stored on the database server

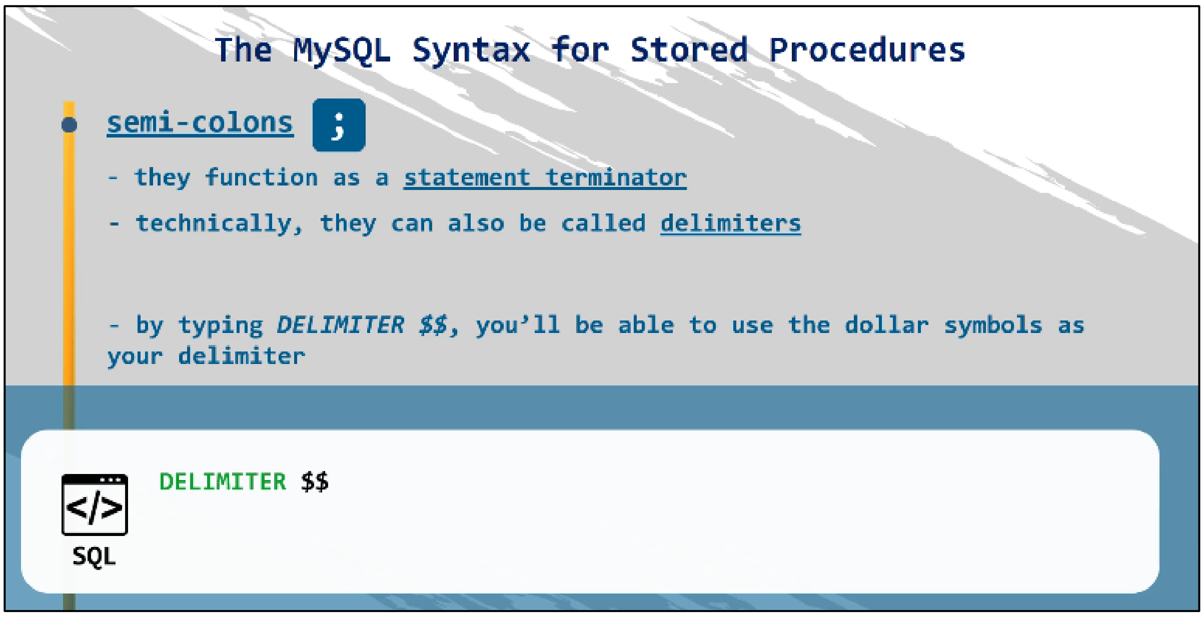
- whenever a user needs to run the query in question, they can *call*, *reference*, or *invoke* the routine

We essentially use this when there is a particular query we run multiple times a day – so instead of constantly having to rewrite it, we just call it instead.



We can have two types of stored routines: stored procedures and functions. Funny enough, all procedures are stored. Therefore, we could simply refer to them as procedures if we prefer. Functions, though, can be of various types; they could be programmed manually, and in that case, they will act like stored routines. These are the user-defined functions. Alternatively, we can have built-in functions in MySQL, which means that these functions have already been defined inside MySQL. We've already used some of them, like the aggregate functions or the date-time functions. So when we say functions, only the context will clarify which type of functions we are talking about.





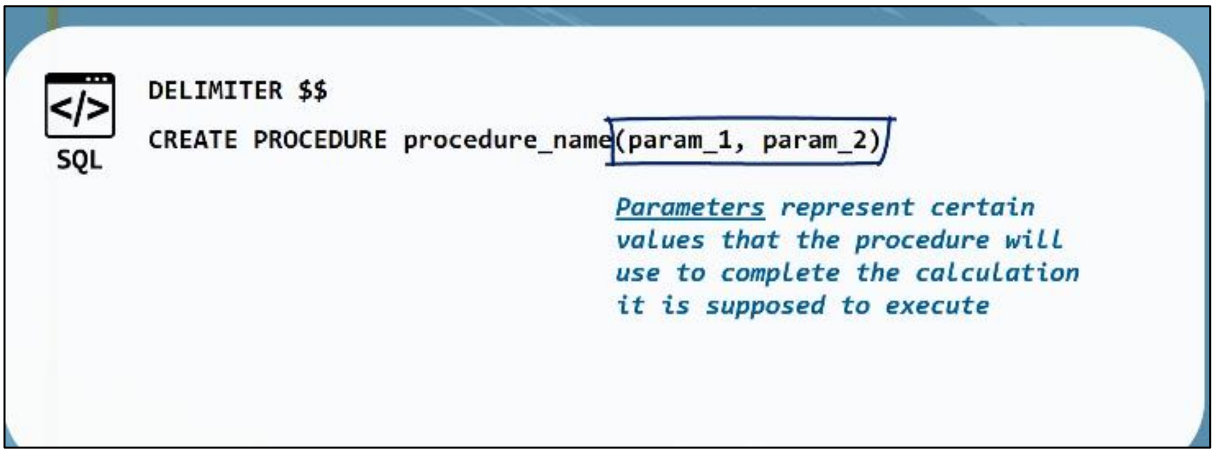
The MySQL Syntax for Stored Procedures

- semi-colons ;
  - they function as a statement terminator
  - technically, they can also be called delimiters
  - by typing *DELIMITER $$*, you'll be able to use the dollar symbols as your delimiter

```sql
DELIMITER $$
```

We are ready to move on. Think of how semicolons are used in SQL. Initially, we said the function was a statement terminator, but technically, they can also be called delimiters. And by typing 'delimiter' and the dollar symbol two times, we'll be able to use the dollar symbol as our delimiter. The semicolon isn't your dilemma anymore.

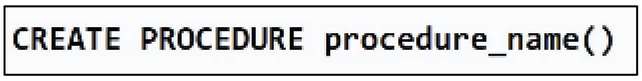
Well, think of the long sheets of code we can have in the SQL editor. There, every query is terminated by a semicolon, right? As we know, the SQL engine will run only the first of the statements in our procedure, and we'll move on to the next query that is beyond the procedure. It is not going to read the code after the first semicolon. To avoid this problem, we need a temporary delimiter different from the standard semicolon. There are various symbols you can use, a double dollar sign or a double forward slash, for instance. It doesn't really matter which one you choose. I will opt for the double dollar symbol.



```sql
DELIMITER $$
CREATE PROCEDURE procedure_name(param_1, param_2)
```

Parameters represent certain values that the procedure will use to complete the calculation it is supposed to execute

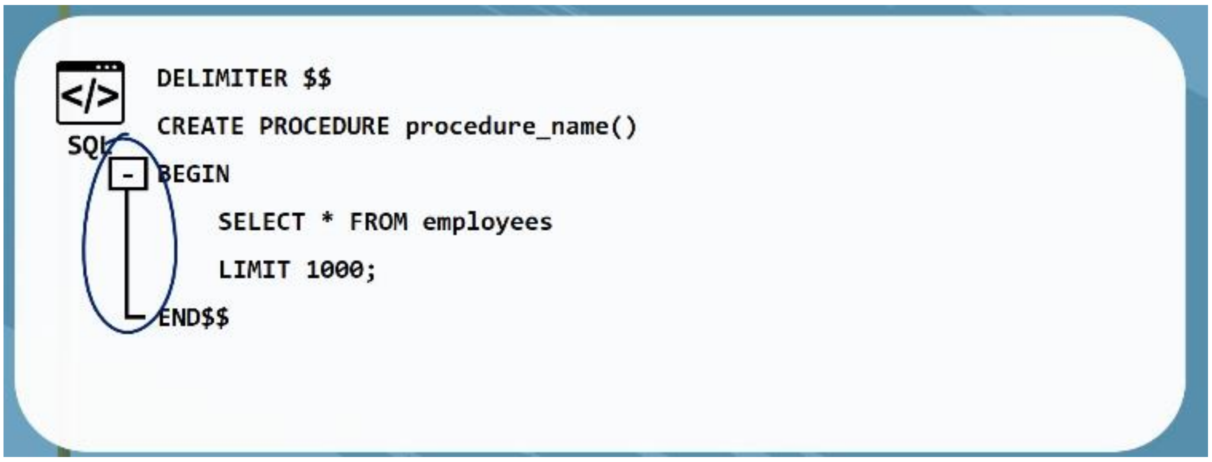
We must then write, 'create procedure' and attach the name we would like to assign to it. Next to the name, remember that we **must** always **open** and **close parentheses**. They are inherent to the syntax for creating a procedure because, within these parentheses, you would typically insert parameters. What do parameters do? They represent certain values that the procedure will use to complete the calculation it is supposed to execute. However please remember that a procedure can be created without parameters too. Nevertheless, the parentheses must always be attached to its name. Otherwise, My SQL will display an error message.



```sql
CREATE PROCEDURE procedure_name()
```

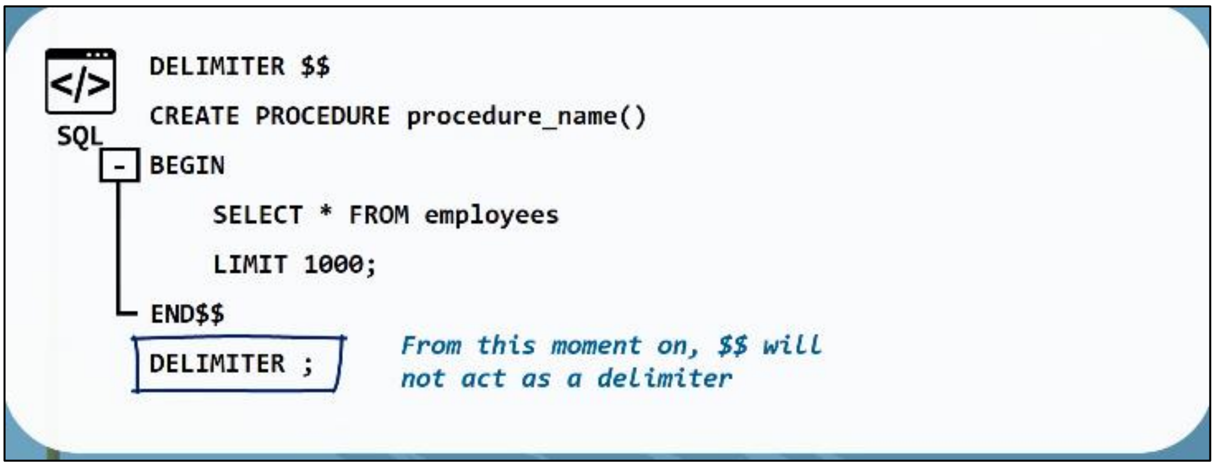
What follows is the body of the procedure. It is always enclosed between the keyword begin, the keyword end, and the temporary delimiter, which in our case is a double dollar sign. My SQL

Workbench will display a black vertical line along the left side of the body of the procedure with a tiny box on top. By clicking on the minus or the plus sign located within this box, you can hide or expand the code of the body.



```
DELIMITER $$
CREATE PROCEDURE procedure_name()
BEGIN
    SELECT * FROM employees
    LIMIT 1000;
END$$
```

The body of the procedure is composed of a query, and this query is the reason we are creating the entire procedure in the first place. It will be placed between the begin and end keywords. More importantly, however, at the end of this query, we'll have the usual delimiter: the semicolon – not the double dollar sign. How come? Well, if we use the delimiter again, the creation of the procedure will stop here, and MySQL will show an error.

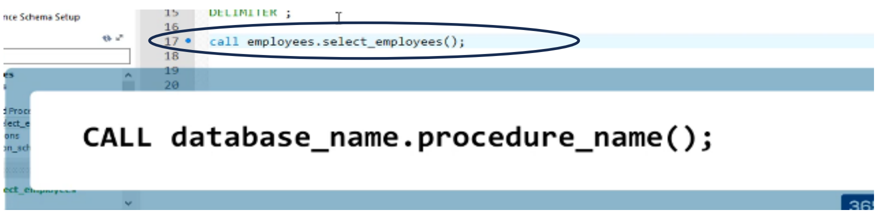


```
DELIMITER $$
CREATE PROCEDURE procedure_name()
BEGIN
    SELECT * FROM employees
    LIMIT 1000;
END$$
DELIMITER ;
```

From this moment on, $$ will not act as a delimiter

Finally, do not forget to reset the delimiter to the classical semicolon symbol. If you forget to do that, you risk making the opposite mistake; not running any of the code succeeding the line where we are calling the procedure. And from this moment on, the double dollar sign will not act as a delimiter. Once again, the semicolon will have this role.

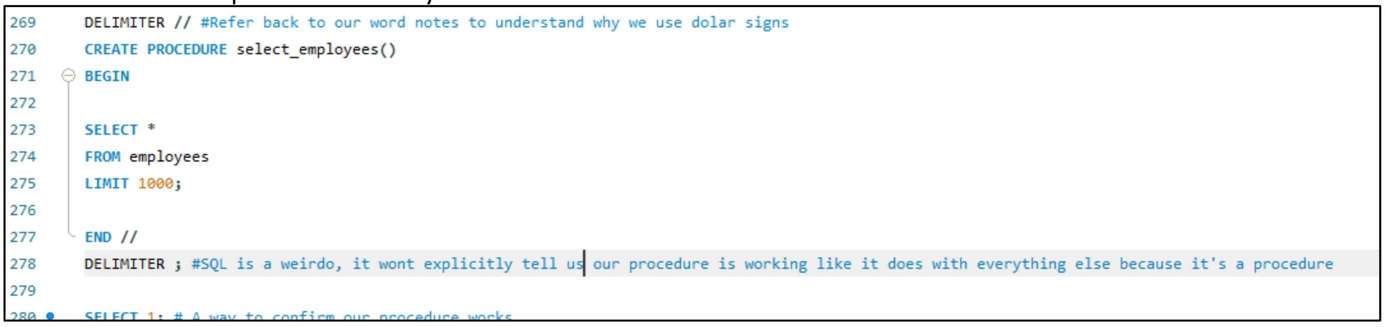
At this stage, we can talk about invoking the procedure. Essentially, there are three primary methods to achieve this. The first one involves the following syntax. Call the name of the database the stored procedure is applied to:



```
DELIMITER ;

call employees.select_employees();
```

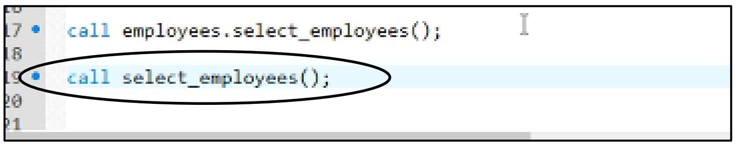
```
CALL database_name.procedure_name();
```

When executed, this line of code will deliver the first 1,000 rows from the employee's table, as outlined in the procedure's body of code.



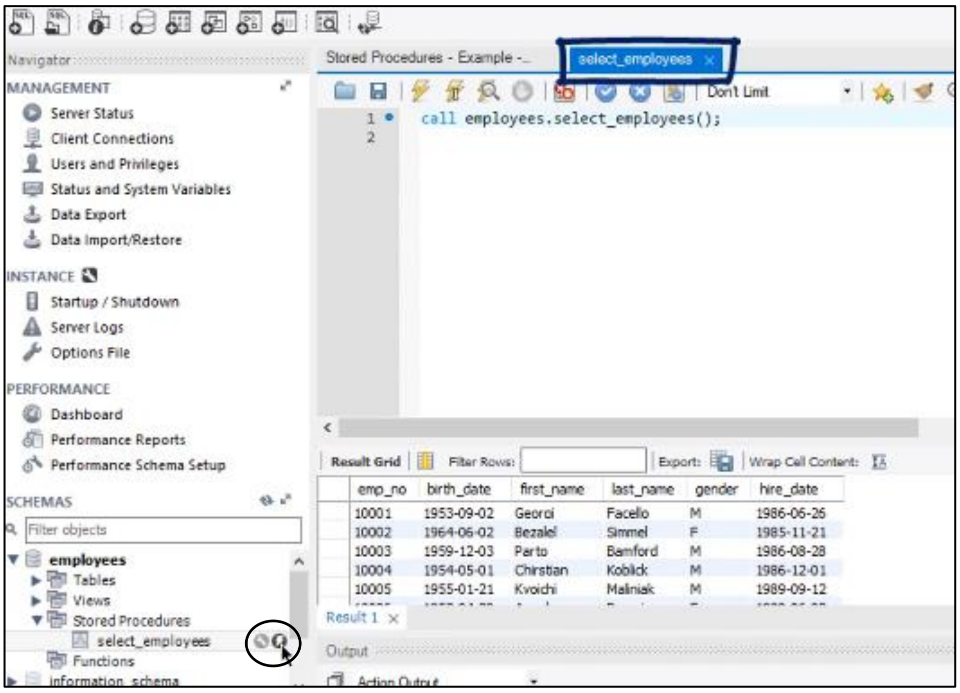
```
DELIMITER // #Refer back to our word notes to understand why we use dolar signs
CREATE PROCEDURE select_employees()
BEGIN

SELECT *
FROM employees
LIMIT 1000;

END //
DELIMITER ; #SQL is a weirdo, it wont explicitly tell us our procedure is working like it does with everything else because it's a procedure

SELECT 1; # A way to confirm our procedure works
```

The second way to invoke "select employees" would be to take advantage of the fact that we have already selected employees as our default database: "USE employees;". In other words, we can skip the database name part and call the procedure name directly.



```
call employees.select_employees();

call select_employees();
```

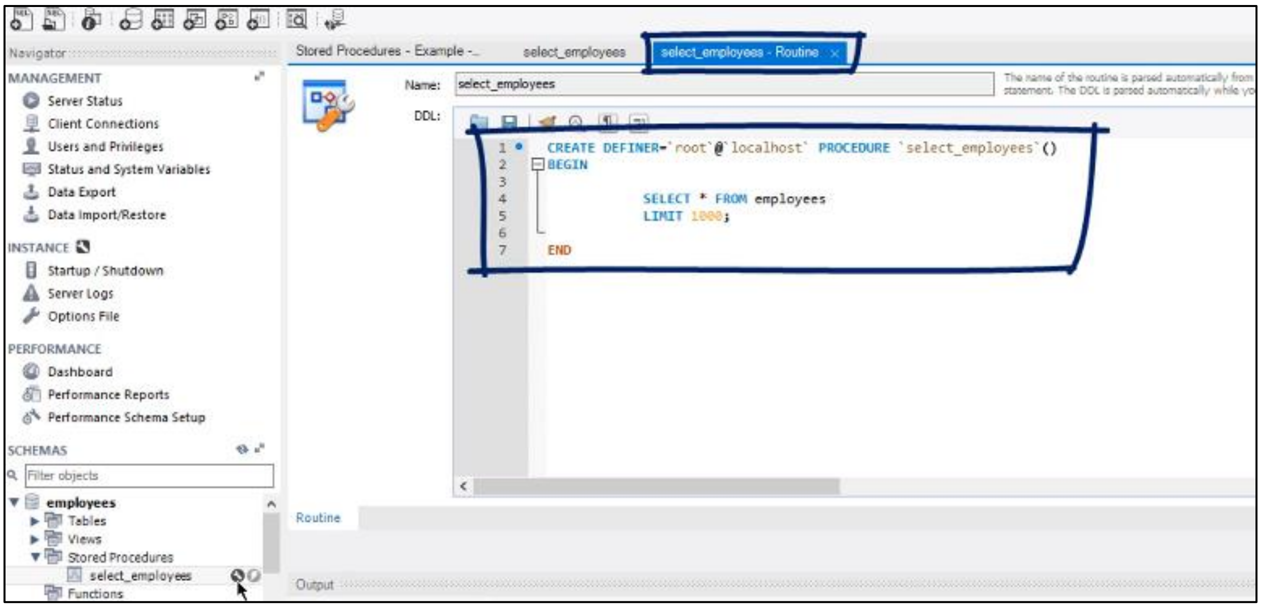
The third way to invoke a procedure is to click on this tiny lightning symbol that turns up as you hover over the name select employees in the schema section in workbench. After you press this button, a new tab will open to the right. In its top part, you see a newly started SQL window whose only line of code is identical to the first option for invoking the procedure we discussed. Logically, in the middle part of this tab, you can see the results set obtained, the first 1,000 rows of the employee's table, awesome. These are the three ways to invoke a procedure in MySQL.



```
call employees.select_employees();
```

emp_no	birth_date	first_name	last_name	gender	hire_date
10001	1953-09-02	Georgi	Facello	M	1986-06-26
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21
10003	1959-12-03	Parto	Bamford	M	1986-08-28
10004	1954-05-01	Chirstian	Koblick	M	1986-12-01
10005	1955-01-21	Kyoichi	Maliniak	M	1989-09-12

<u>Next to the lightning symbol we used, you can see a tiny icon depicting a wrench. If you click on it, a new separate tab will appear. It shows the whole procedure in question. This is an amazingly useful tool, as it channels your search directly towards the code of the stored procedure and allows for a quicker and more purposeful correction of its structure whenever necessary.</u>

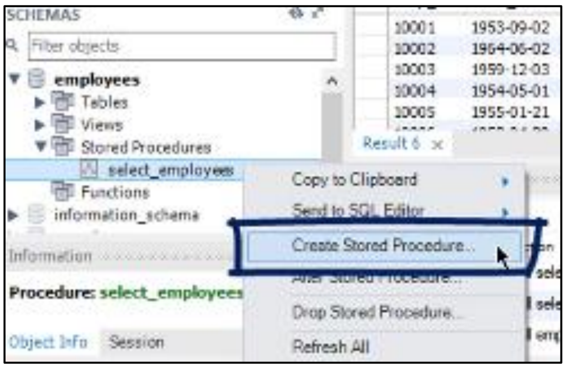


Name: select_employees

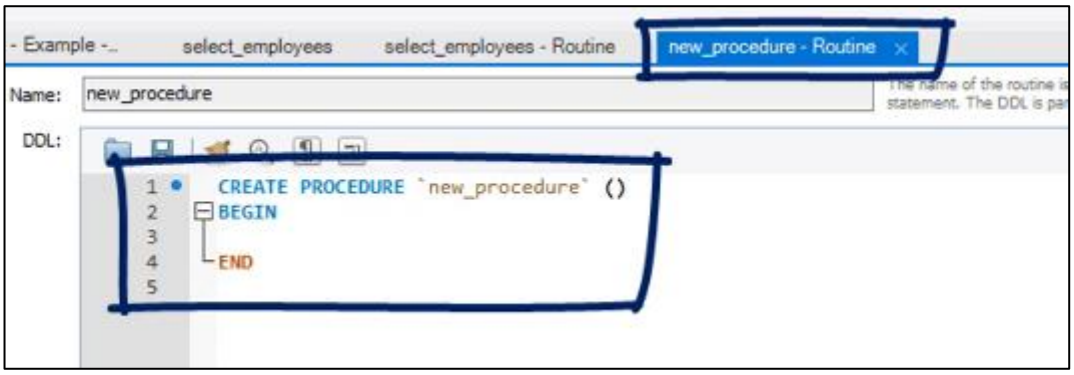
DDL:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `select_employees`()
BEGIN

        SELECT * FROM employees
        LIMIT 1000;

END
```

Another way of creating procedures is by doing it through the schemas tab:



A new table will then pop up with the skeleton of a procedure for you to fill in:



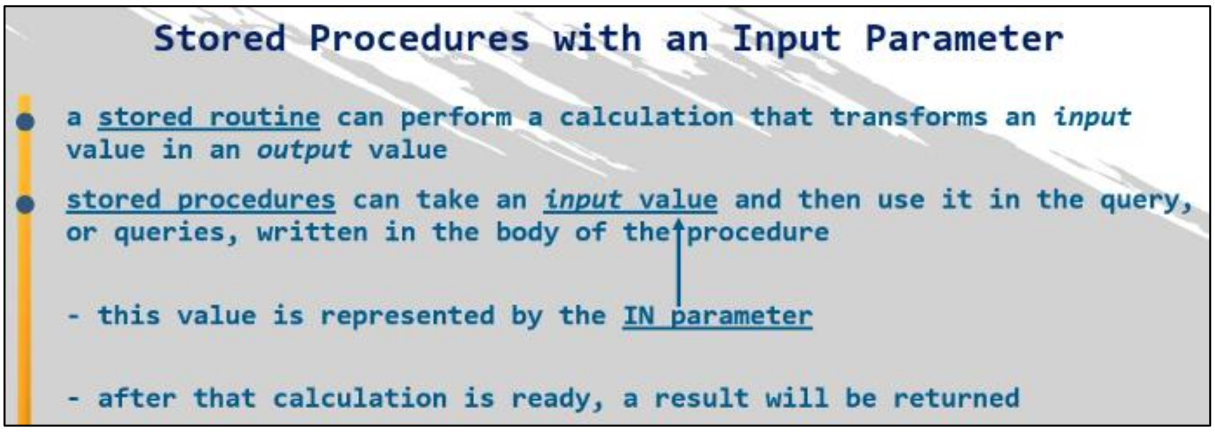
Name: new_procedure

DDL:

```
CREATE PROCEDURE `new_procedure` ()
BEGIN

END
```

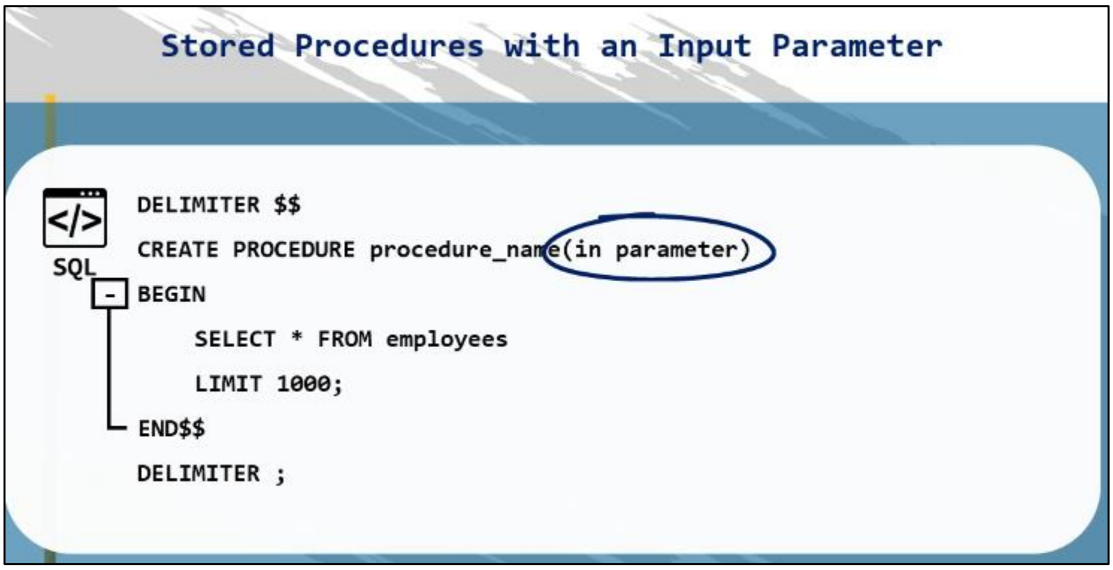
So much easierrrrr!

<u>NEW EXPANSION ON TOPIC:</u>



Stored Procedures with an Input Parameter

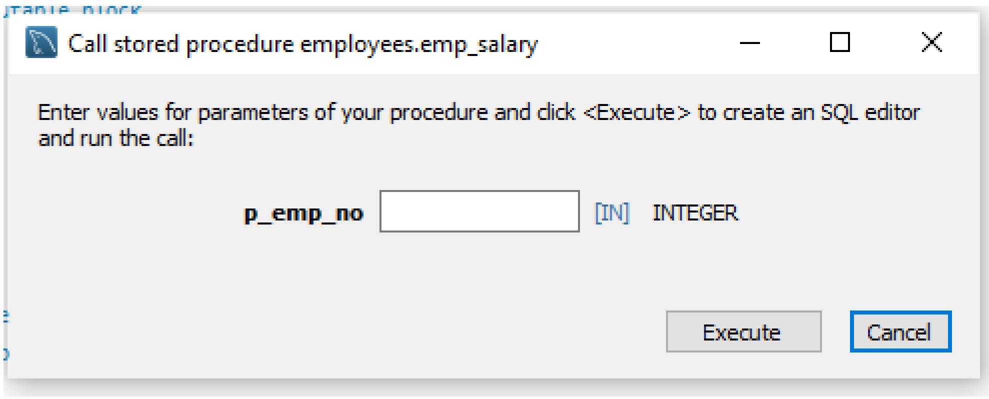
- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an *input value* and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter
  - after that calculation is ready, a result will be returned



Stored Procedures with an Input Parameter

```
DELIMITER $$
CREATE PROCEDURE procedure_name(in parameter)
BEGIN
    SELECT * FROM employees
    LIMIT 1000;
END$$
DELIMITER ;
```

This is essentially the result of using an IN parameter:

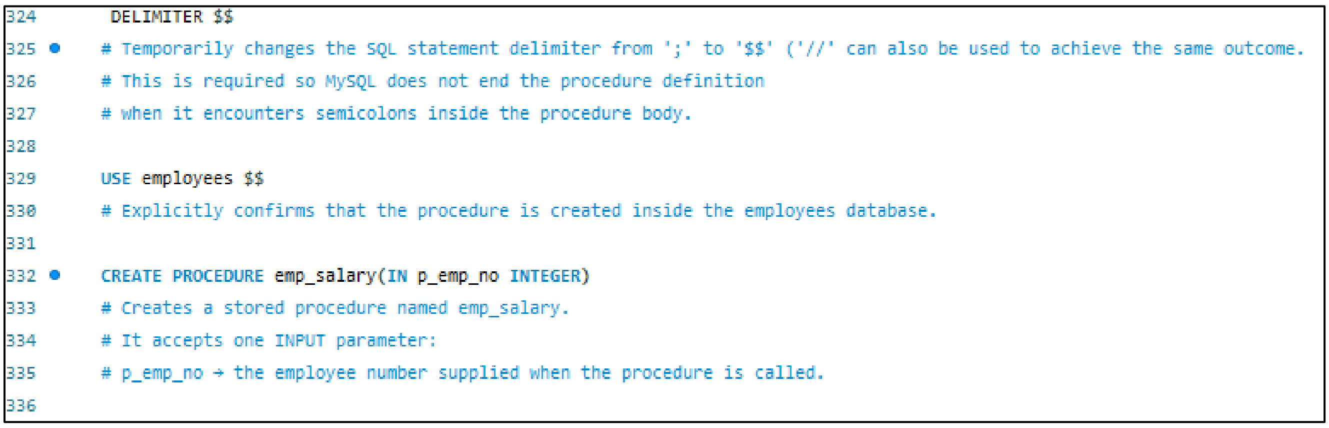


Call stored procedure employees.emp_salary

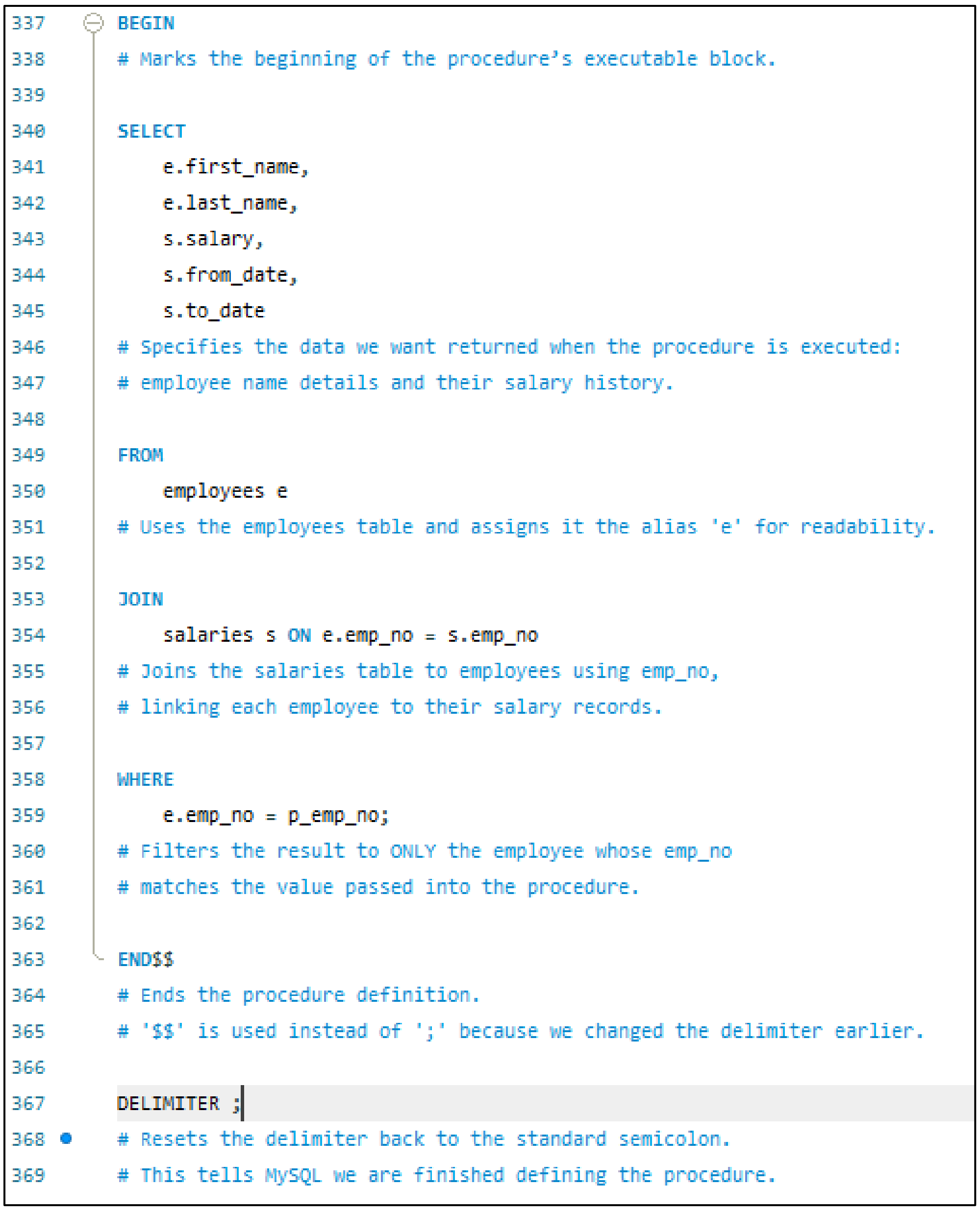
Enter values for parameters of your procedure and click <Execute> to create an SQL editor and run the call:

p_emp_no [IN] INTEGER

The code along with the breakdown is below – incredibly interesting and shows how we're now moving from writing static queries to creating **reusable, parameter-driven logic.** This is exactly how SQL starts behaving like **a real programming language**.



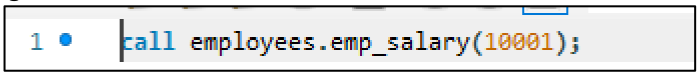
```
 DELIMITER $$
# Temporarily changes the SQL statement delimiter from ';' to '$$' ('//' can also be used to achieve the same outcome.
# This is required so MySQL does not end the procedure definition
# when it encounters semicolons inside the procedure body.

USE employees $$
# Explicitly confirms that the procedure is created inside the employees database.

CREATE PROCEDURE emp_salary(IN p_emp_no INTEGER)
# Creates a stored procedure named emp_salary.
# It accepts one INPUT parameter:
# p_emp_no → the employee number supplied when the procedure is called.

```



```
BEGIN
# Marks the beginning of the procedure's executable block.

SELECT
    e.first_name,
    e.last_name,
    s.salary,
    s.from_date,
    s.to_date
# Specifies the data we want returned when the procedure is executed:
# employee name details and their salary history.

FROM
    employees e
# Uses the employees table and assigns it the alias 'e' for readability.

JOIN
    salaries s ON e.emp_no = s.emp_no
# Joins the salaries table to employees using emp_no,
# linking each employee to their salary records.

WHERE
    e.emp_no = p_emp_no;
# Filters the result to ONLY the employee whose emp_no
# matches the value passed into the procedure.

END$$
# Ends the procedure definition.
# '$$' is used instead of ';' because we changed the delimiter earlier.

DELIMITER ;
# Resets the delimiter back to the standard semicolon.
# This tells MySQL we are finished defining the procedure.
```

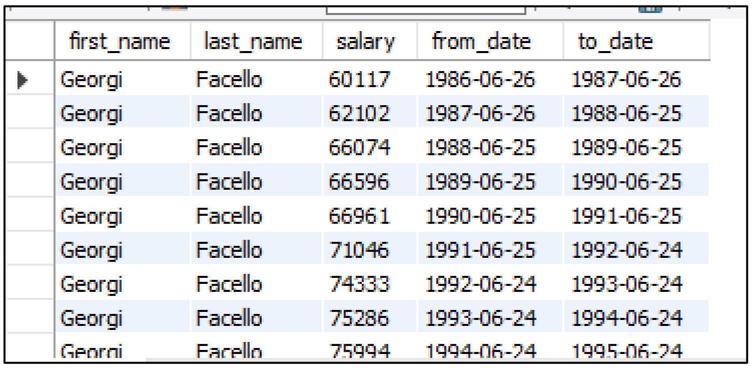
Stored procedures such as this allow businesses to centralise logic in the database, accept dynamic input, and expose controlled, reusable operations instead of raw data access. In a business environment, HR analysts and payroll teams frequently need to retrieve salary histories for individual employees without exposing the full salary table or rewriting complex SQL queries.

By creating a stored procedure such as emp_salary, the organisation encapsulates this logic within the database itself. The procedure accepts an employee number as an input parameter and returns only the relevant salary information for that employee. So if we wanted salary information about employee 10001, we'd enter it into the parentheses of the stored procedure, and this is what we'd get as a result:

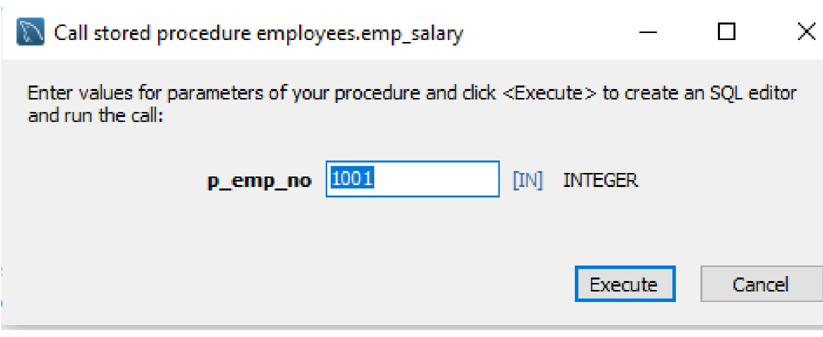


```
call employees.emp_salary(10001);
```

We then get all the contracts Georgi Facello has had since joining the company:



first_name	last_name	salary	from_date	to_date
Georgi	Facello	60117	1986-06-26	1987-06-26
Georgi	Facello	62102	1987-06-26	1988-06-25
Georgi	Facello	66074	1988-06-25	1989-06-25
Georgi	Facello	66596	1989-06-25	1990-06-25
Georgi	Facello	66961	1990-06-25	1991-06-25
Georgi	Facello	71046	1991-06-25	1992-06-24
Georgi	Facello	74333	1992-06-24	1993-06-24
Georgi	Facello	75286	1993-06-24	1994-06-24
Georgi	Facello	75994	1994-06-24	1995-06-24



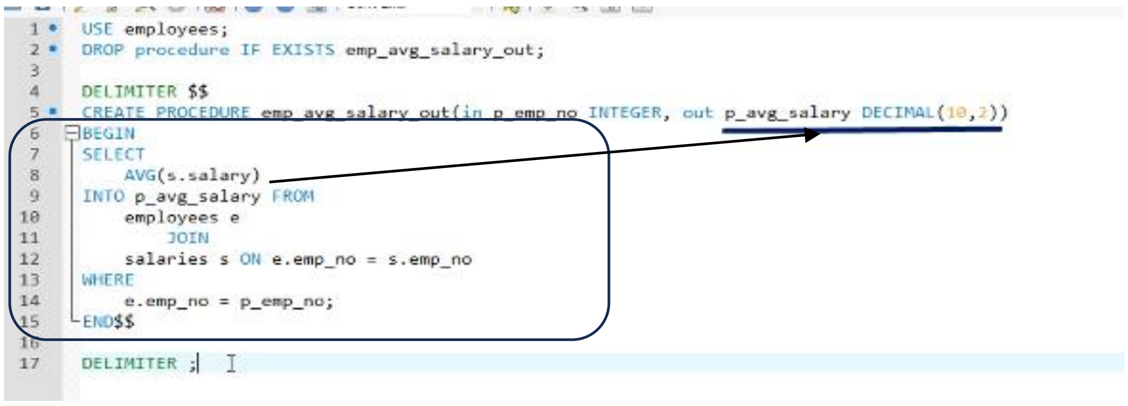
Call stored procedure employees.emp_salary

Enter values for parameters of your procedure and click <Execute> to create an SQL editor and run the call:

p_emp_no 1001 [IN] INTEGER

<u>Stored Procedures with INPUT AND OUTPUT PROCEDURES:</u>

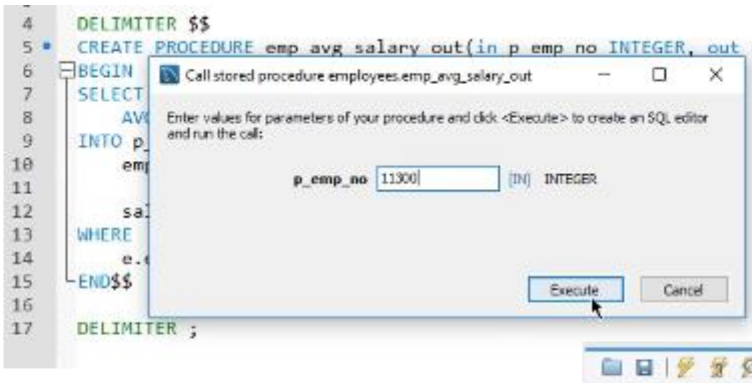
Okay, so it is clear how procedures with zero or one parameters work. In cases where the select statement is used in the procedure's body, the output will be displayed to the user readily, and they will be able to treat the result as they like. However, if the outcome is supposed to be stored in another variable that can be recorded in the database and that can be used in outside applications, a second parameter must be defined within the parentheses. This parameter will be called an OUT parameter. It will represent the variable containing the output value of the operation executed by the query of the stored procedure.



```sql
USE employees;
DROP procedure IF EXISTS emp_avg_salary_out;

DELIMITER $$
CREATE PROCEDURE emp_avg_salary_out(in p_emp_no INTEGER, out p_avg_salary DECIMAL(10,2))
BEGIN
SELECT
    AVG(s.salary)
INTO p_avg_salary FROM
    employees e
        JOIN
    salaries s ON e.emp_no = s.emp_no
WHERE
    e.emp_no = p_emp_no;
END$$

DELIMITER ;
```

We will use P employee number as an in parameter again, and we will add P average salary as an out parameter. It will be of the DECIMAL type because it will define a monetary value, let it be of a precision of 10, and more importantly, a scale of two. Second, look at the body of the procedure. The query must reflect our idea to store just a single value in our parameter. That's why we will need only one selection: The average amount obtained from the salary column in the salaries table. We must then insert this value into the out parameter we just declared. This is the philosopher stone of my SQL stored procedures. When out parameters are in play. Every time you create a procedure containing both an in and an out parameter, remember that you must use the SELECT INTO structure in the query of this object's body. And **this time outcome of 48,193.80 cents is not just displayed for the user but is also stored in the P average salary parameter.**

The result of the IN & OUT parameters:

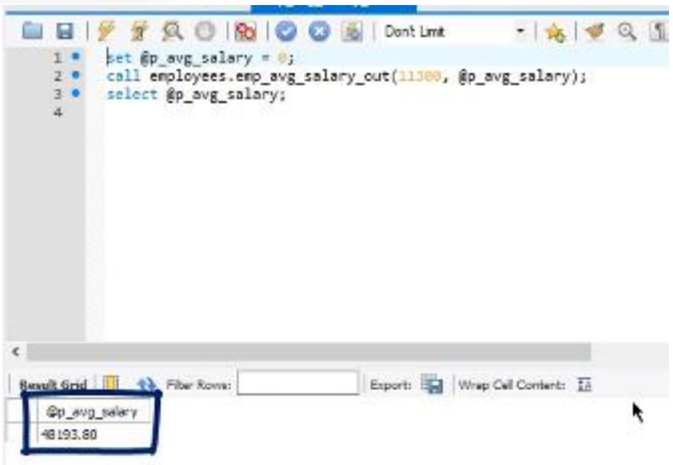


```sql
DELIMITER $$
CREATE PROCEDURE emp_avg_salary_out(in p_emp_no INTEGER, out p_avg_salary DECIMAL(10,2))
BEGIN
SELECT
    AV
INTO p
    em
    
    sa
WHERE
    e.
END$$

DELIMITER ;
```

Call stored procedure employees.emp_avg_salary_out

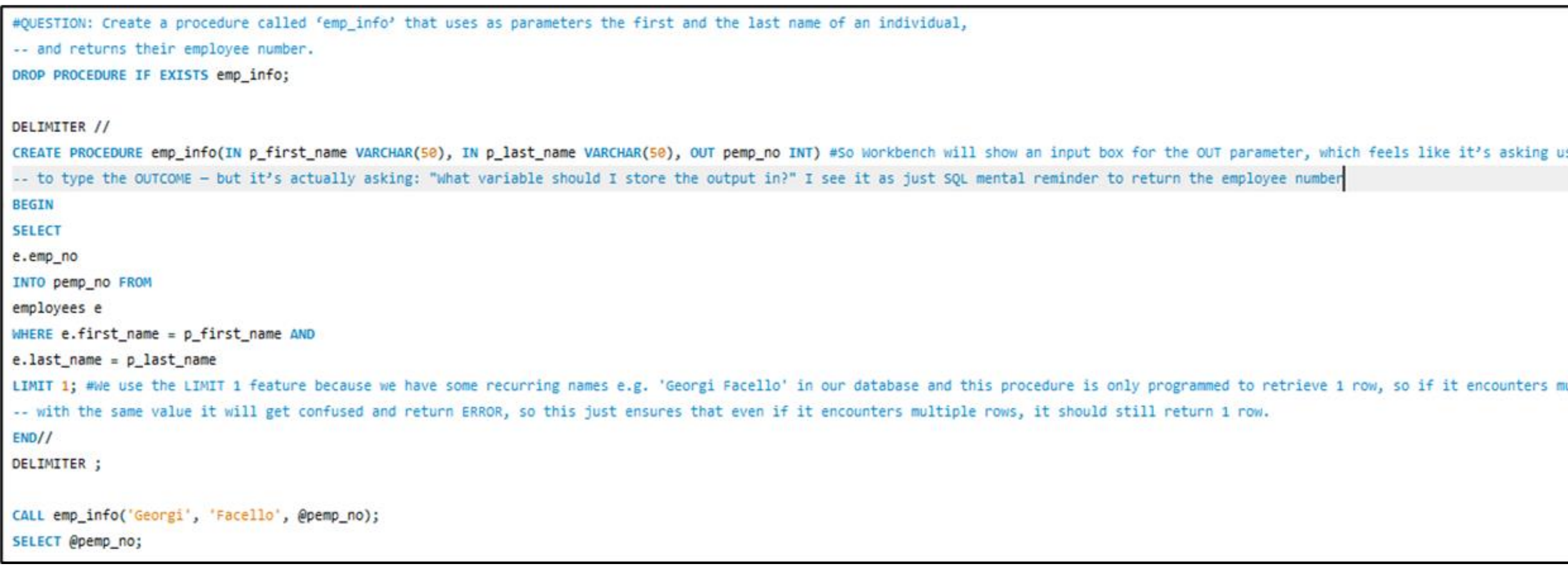
Enter values for parameters of your procedure and click <Execute> to create an SQL editor and run the call:

p_emp_no 11300 [IN] INTEGER

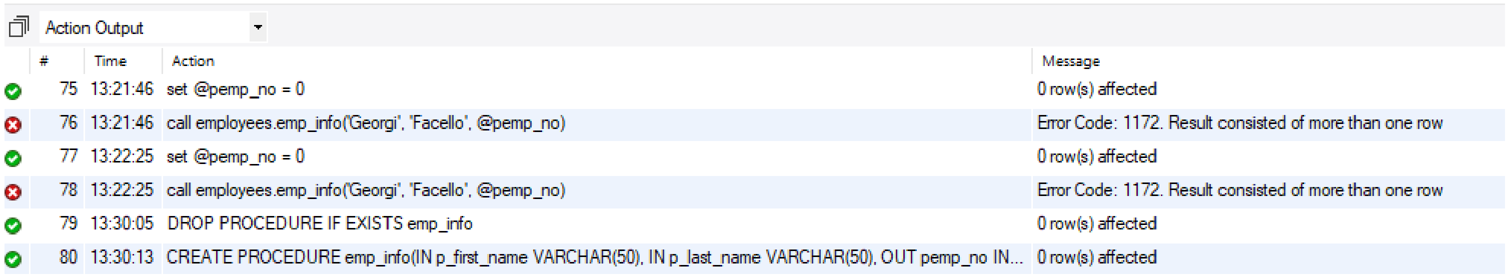


```sql
set @p_avg_salary = 0;
call employees.emp_avg_salary_out(11300, @p_avg_salary);
select @p_avg_salary;
```

```
@p_avg_salary
48193.80
```



```
#QUESTION: Create a procedure called ‘emp_info’ that uses as parameters the first and the last name of an individual,
-- and returns their employee number.
DROP PROCEDURE IF EXISTS emp_info;

DELIMITER //
CREATE PROCEDURE emp_info(IN p_first_name VARCHAR(50), IN p_last_name VARCHAR(50), OUT pemp_no INT) #So Workbench will show an input box for the OUT parameter, which feels like it’s asking u
-- to type the OUTCOME – but it’s actually asking: "What variable should I store the output in?" I see it as just SQL mental reminder to return the employee number
BEGIN
SELECT
e.emp_no
INTO pemp_no FROM
employees e
WHERE e.first_name = p_first_name AND
e.last_name = p_last_name
LIMIT 1; #We use the LIMIT 1 feature because we have some recurring names e.g. 'Georgi Facello' in our database and this procedure is only programmed to retrieve 1 row, so if it encounters m
-- with the same value it will get confused and return ERROR, so this just ensures that even if it encounters multiple rows, it should still return 1 row.
END//
DELIMITER ;

CALL emp_info('Georgi', 'Facello', @pemp_no);
SELECT @pemp_no;
```

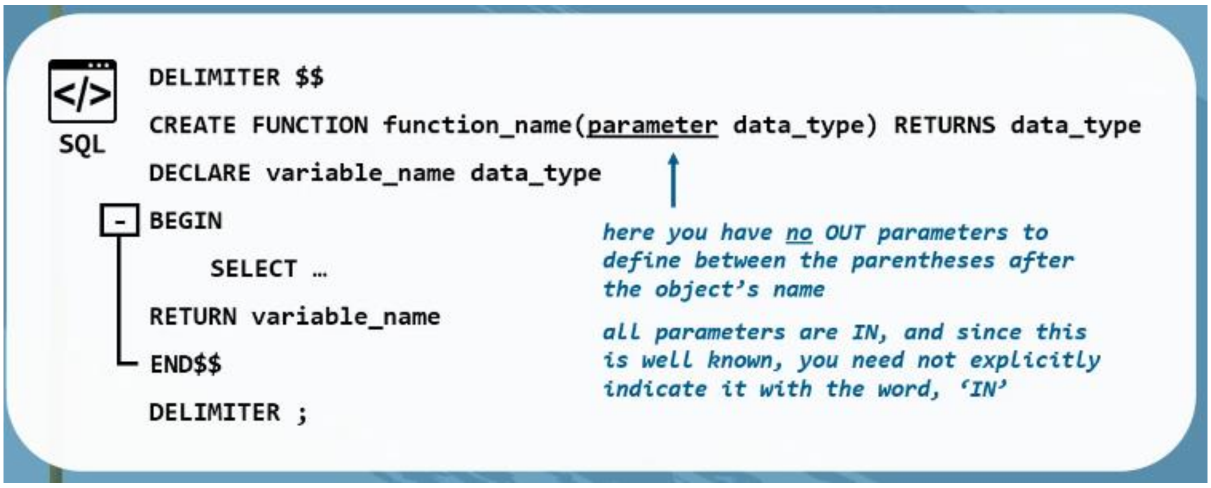


Action Output

	#	Time	Action	Message
✓	75	13:21:46	set @pemp_no = 0	0 row(s) affected
✗	76	13:21:46	call employees.emp_info('Georgi', 'Facello', @pemp_no)	Error Code: 1172. Result consisted of more than one row
✓	77	13:22:25	set @pemp_no = 0	0 row(s) affected
✗	78	13:22:25	call employees.emp_info('Georgi', 'Facello', @pemp_no)	Error Code: 1172. Result consisted of more than one row
✓	79	13:30:05	DROP PROCEDURE IF EXISTS emp_info	0 row(s) affected
✓	80	13:30:13	CREATE PROCEDURE emp_info(IN p_first_name VARCHAR(50), IN p_last_name VARCHAR(50), OUT pemp_no IN...	0 row(s) affected

T**<u>he Difference Between Stored Procedures & Functions:</u>**

What a function's syntax looks like:

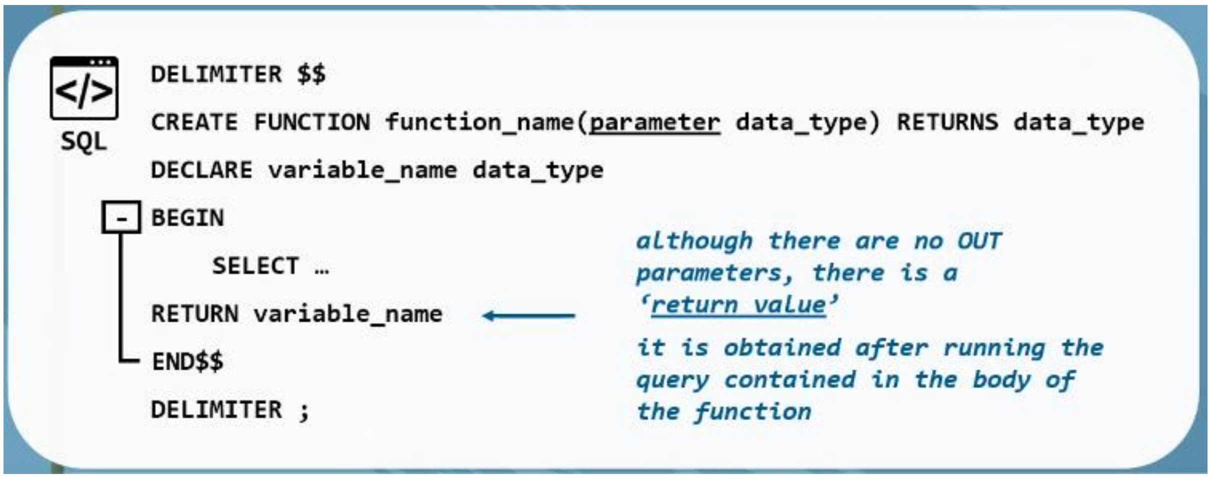


```
DELIMITER $$
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type
DECLARE variable_name data_type
BEGIN
    SELECT …
RETURN variable_name
END$$
DELIMITER ;
```

here you have no OUT parameters to define between the parentheses after the object’s name

all parameters are IN, and since this is well known, you need not explicitly indicate it with the word, ‘IN’

Writing the parameters name and its data type is enough.



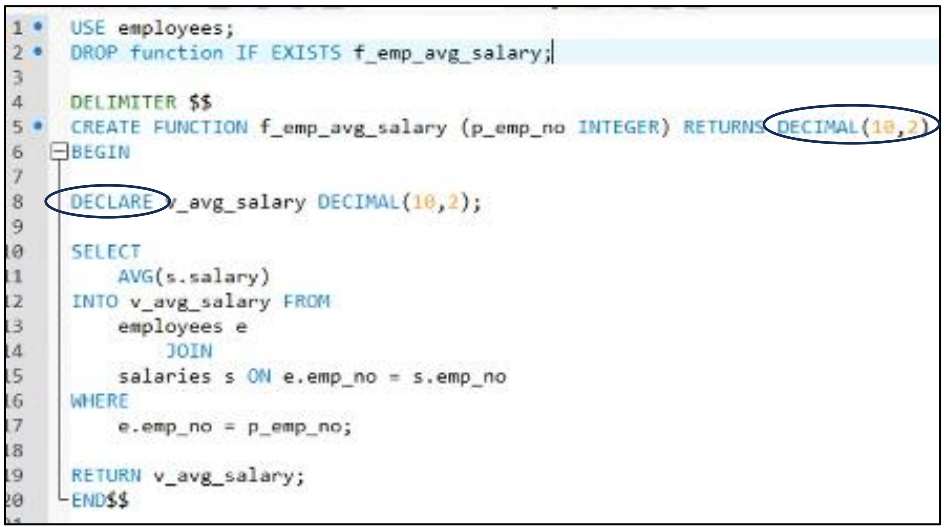
```
DELIMITER $$
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type
DECLARE variable_name data_type
BEGIN
    SELECT …
RETURN variable_name
END$$
DELIMITER ;
```

although there are no OUT parameters, there is a ‘return value’

it is obtained after running the query contained in the body of the function

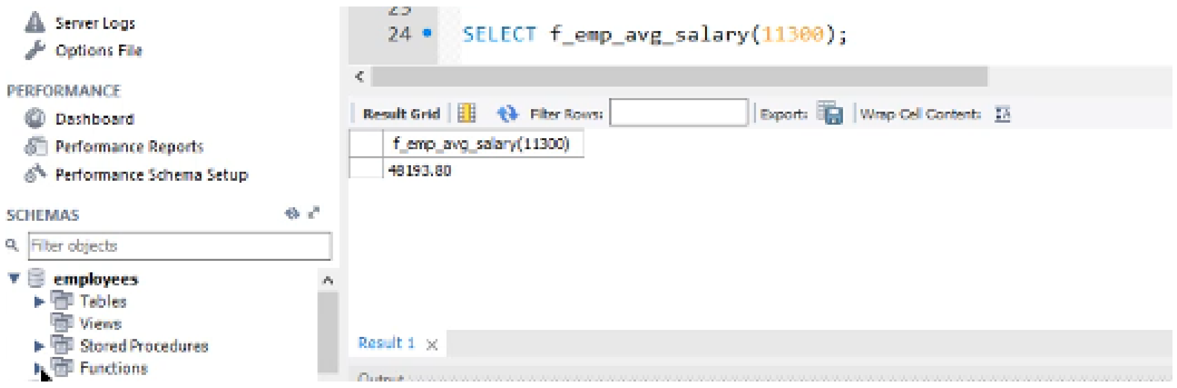
It can be of any data type, so the approach is almost identical to the one used when creating procedures. That's why the code structure is similar, too.

What it would look like:



```
USE employees;
DROP function IF EXISTS f_emp_avg_salary;

DELIMITER $$
CREATE FUNCTION f_emp_avg_salary (p_emp_no INTEGER) RETURNS DECIMAL(10,2)
BEGIN

DECLARE v_avg_salary DECIMAL(10,2);

SELECT
    AVG(s.salary)
INTO v_avg_salary FROM
    employees e
        JOIN
    salaries s ON e.emp_no = s.emp_no
WHERE
    e.emp_no = p_emp_no;

RETURN v_avg_salary;
END$$
```

Then, instead of designating an OUT parameter, the keyword RETURNS must be written outside the parentheses. After that, we should not indicate an object name, but a data type instead, e.g. DECIMAL. We’re still talking about a return value and not a variable. The new variable will be created on the next row, not with the set keyword, but with DECLARE, because this is the word used to create variables visible to the body of the object they belong to. Then, we must indicate the name and the data type of the variable: V_average salary, where V stands for variable. The data type used in the variable line must coincide with the one specified in the create function statement, so DECIMAL (10,2). Lastly, we’ll have to insert a return statement, which merely returns the V average salary value. If we omit this line from the function's body, MySQL will display an error because, conceptually, we would not have satisfied the requirement to set a return value when creating a function.

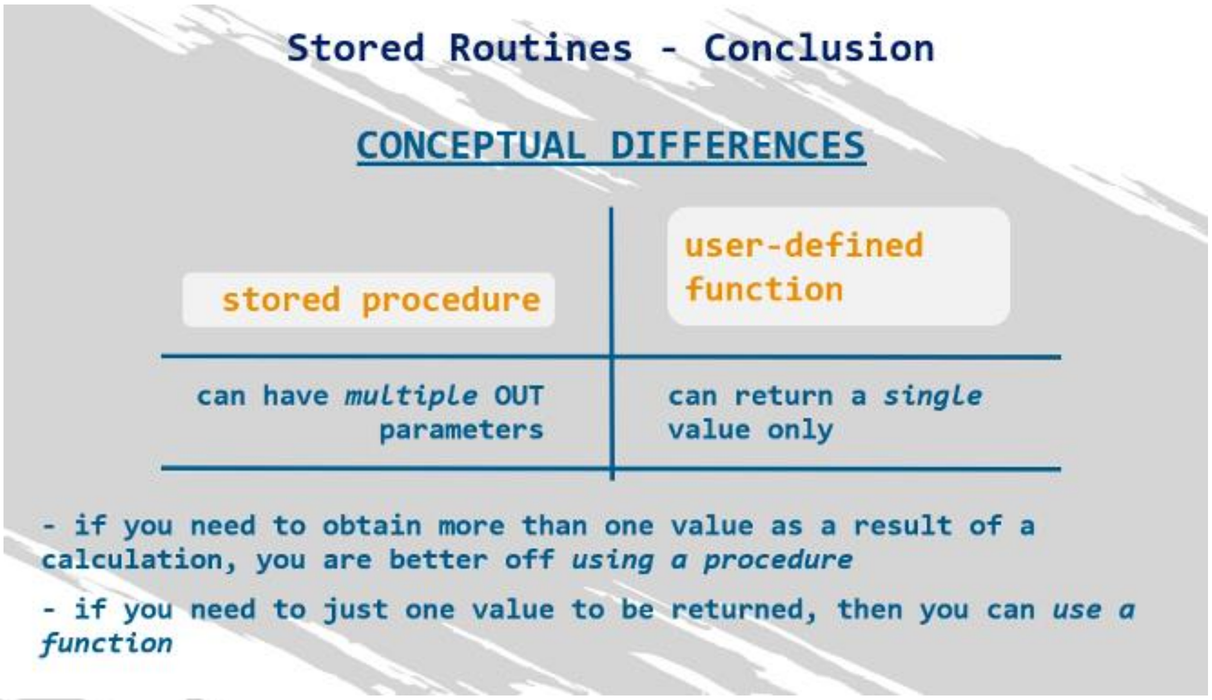


```
SELECT f_emp_avg_salary(11300);
```

```
f_emp_avg_salary(11300)
48193.80
```

Also, we can’t call a function. We can select it, indicating an input value within parentheses.

When we now execute this little query, we will obtain the well-known output of approximately $48,000. Or we can just run it from the Schemas sidebar.



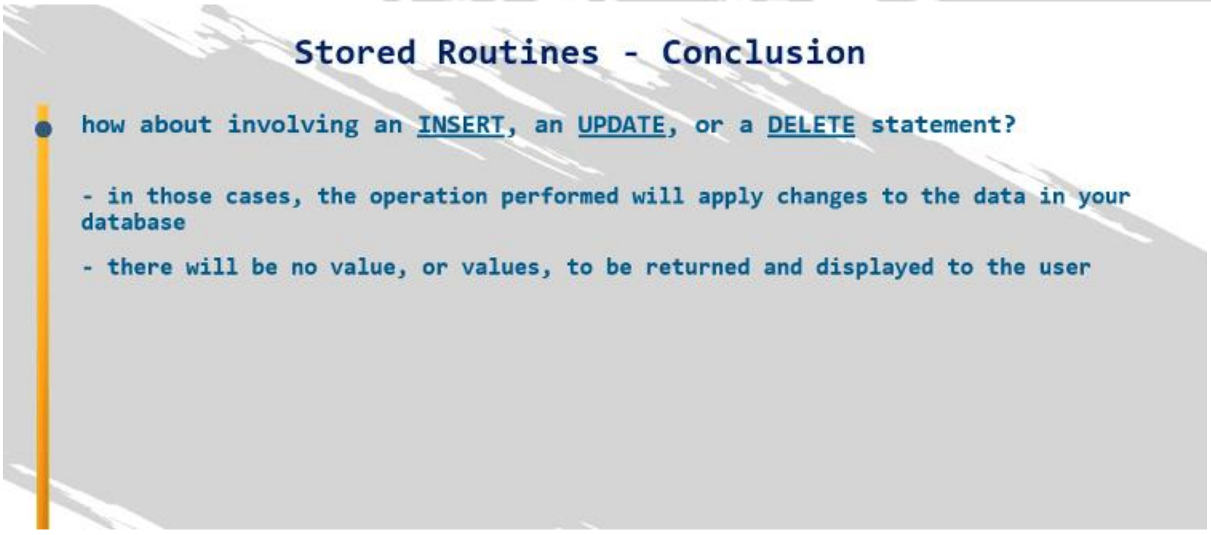
Stored Routines - Conclusion

CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have *multiple* OUT parameters	can return a *single* value only

- if you need to obtain more than one value as a result of a calculation, you are better off *using a procedure*

- if you need to just one value to be returned, then you can *use a function*

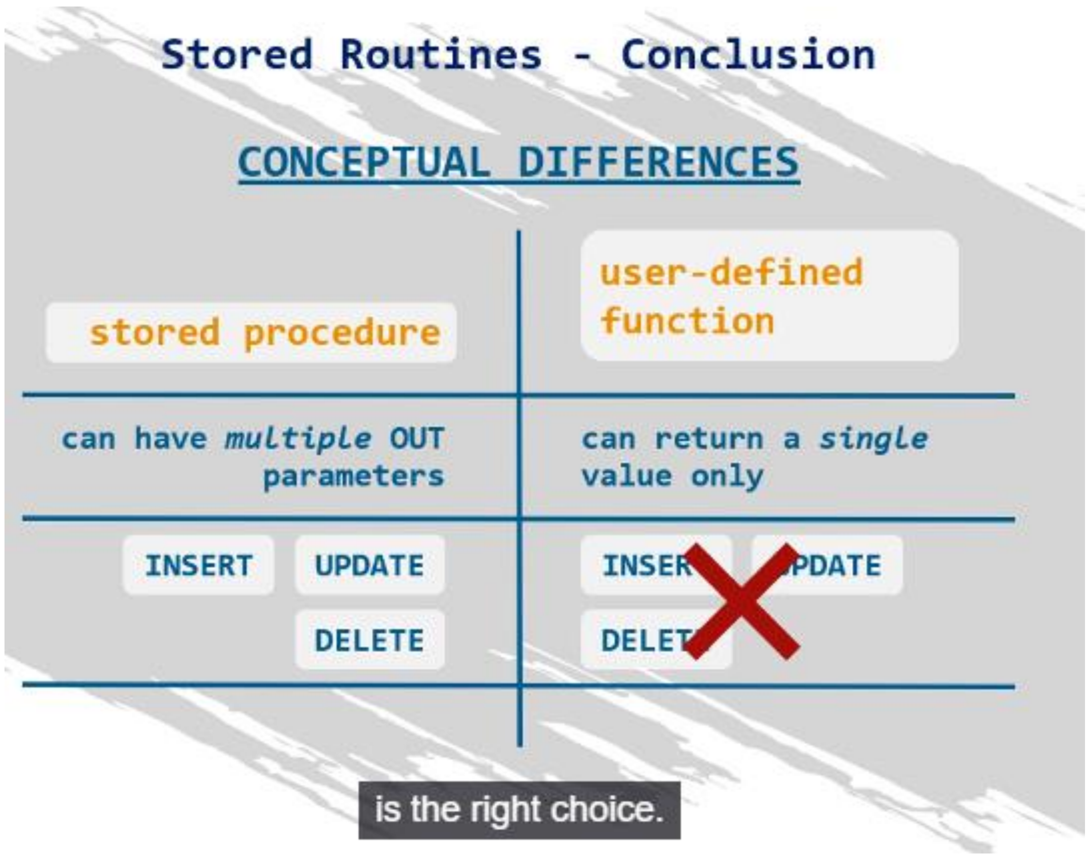


Stored Routines - Conclusion

● how about involving an INSERT, an UPDATE, or a DELETE statement?

- in those cases, the operation performed will apply changes to the data in your database

- there will be no value, or values, to be returned and displayed to the user



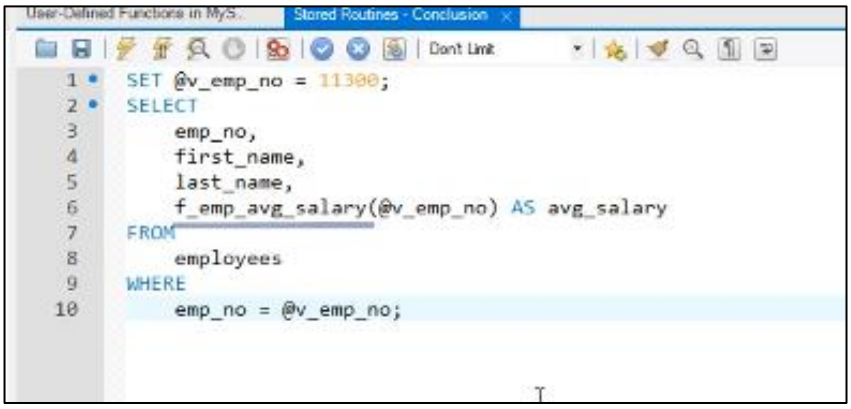
Stored Routines - Conclusion

CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have *multiple* OUT parameters	can return a *single* value only
INSERT UPDATE DELETE	~~INSER PDATE DELET~~

is the right choice.

The third substantial difference between procedures and user defined functions regards the way they can be called in a select statement. This refers to a technical distinction we discussed earlier in the section, which was mentioned at the beginning of this lecture as well. Procedures are invoked using the call keyword, whereas functions are referenced in a select statement. What this means is that you can easily include a function as one of the columns inside a select statement. For example, we can include our function calculating the average employee's salary after the employee's last name. Of course, we can use an alias to rename the column to average salary, then our query would look like this.


User-Defined Functions in MyS...
Stored Routines - Conclusion
Don't Limit

```
SET @v_emp_no = 11300;
SELECT
    emp_no,
    first_name,
    last_name,
    f_emp_avg_salary(@v_emp_no) AS avg_salary
FROM
    employees
WHERE
    emp_no = @v_emp_no;
```

Well remember that including a procedure

in a select statement is impossible.

Once you have become an advanced SQL user

and have gained a lot of practice,

you will appreciate the advantages

and disadvantages of both types of programs.