INTRODUCTION TO SQL

SQL Series Part 1

-Mayuri Dandekar

WHAT IS SQL?

SQL is **Structured Query Language.**

It is a programming language used to interact with database.

There are 4 **basic applications** in SQL

Create, Read, Update, Delete. These are also called as **CRUD** Statements.

Create - Inserts new data

Read (Select) – Reads the data

Update – Update existing data

Delete – Removes the data

SQL V/S NOSQL

SQL	NOSQL
It is Relational Database	It is Non-Relational Database
Data is stored in tables	Data stored as either key-value pair, document-based, graph database or wide-columns.
Database have fixed/ stable/ predefined schema.	Database have dynamic schema
Low performance with huge volume of data	Easily works with huge volumes of data.
Example- PostgreSQL, My-SQL	Example- MongoDB, Hbase

SQL COMMANDS

There are mainly 3 types of SQL commands:

- **DDL**

DDL is Data Definition Language. It includes create, alter, and drop

- **DML**

DML is Data Manipulation Language. It includes select, insert, update and delete

- **DCL**

DCL is Data Control Language. It includes grant and revoke permission to users

WHAT IS DATABASE?

Database is a system that allows users to **store and organize data**.

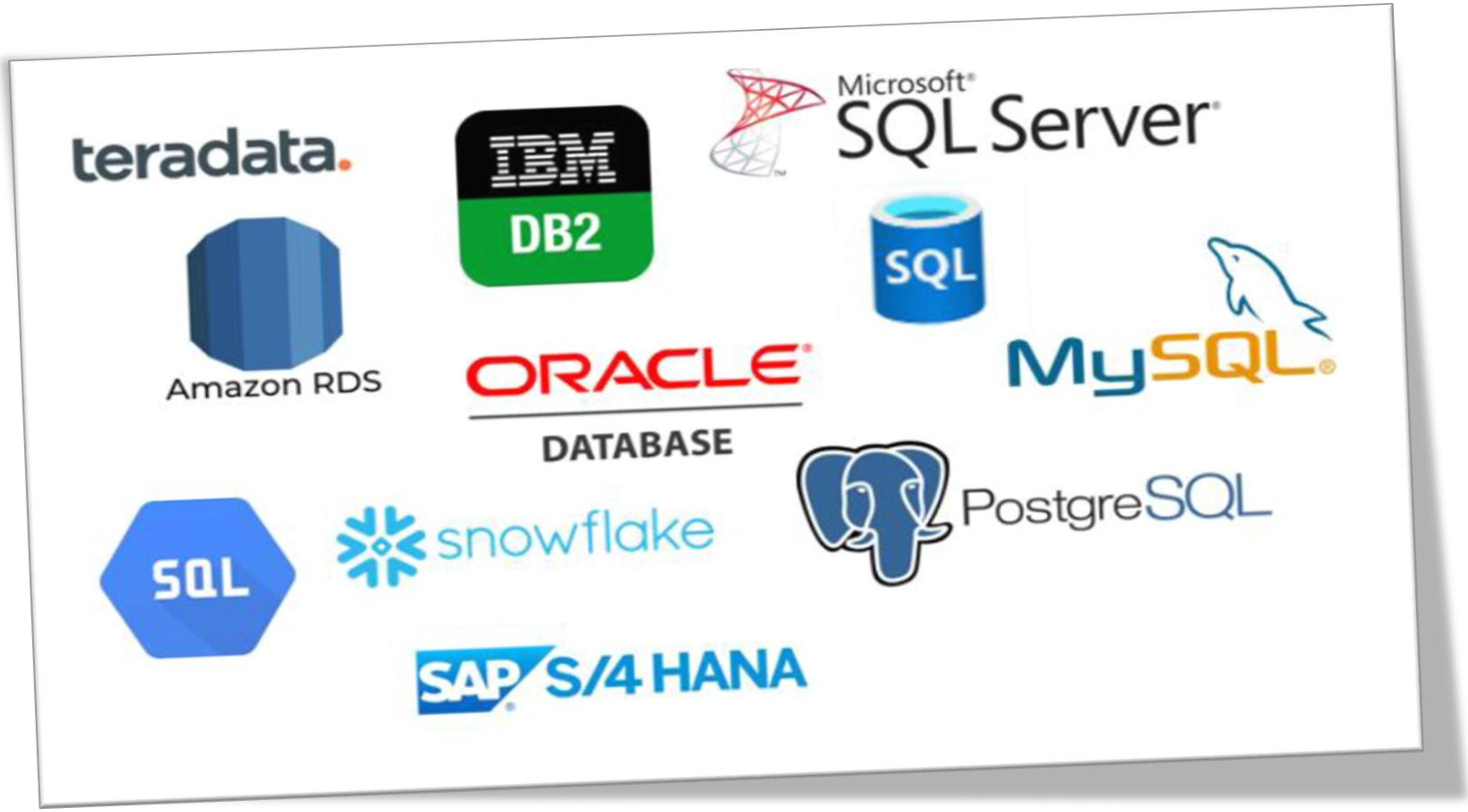
Predominant type of database is **Relational** database.

Relational database organize data in the **form of tables** also sometimes in the form of **queries, views and other elements** to help us interact with the data.

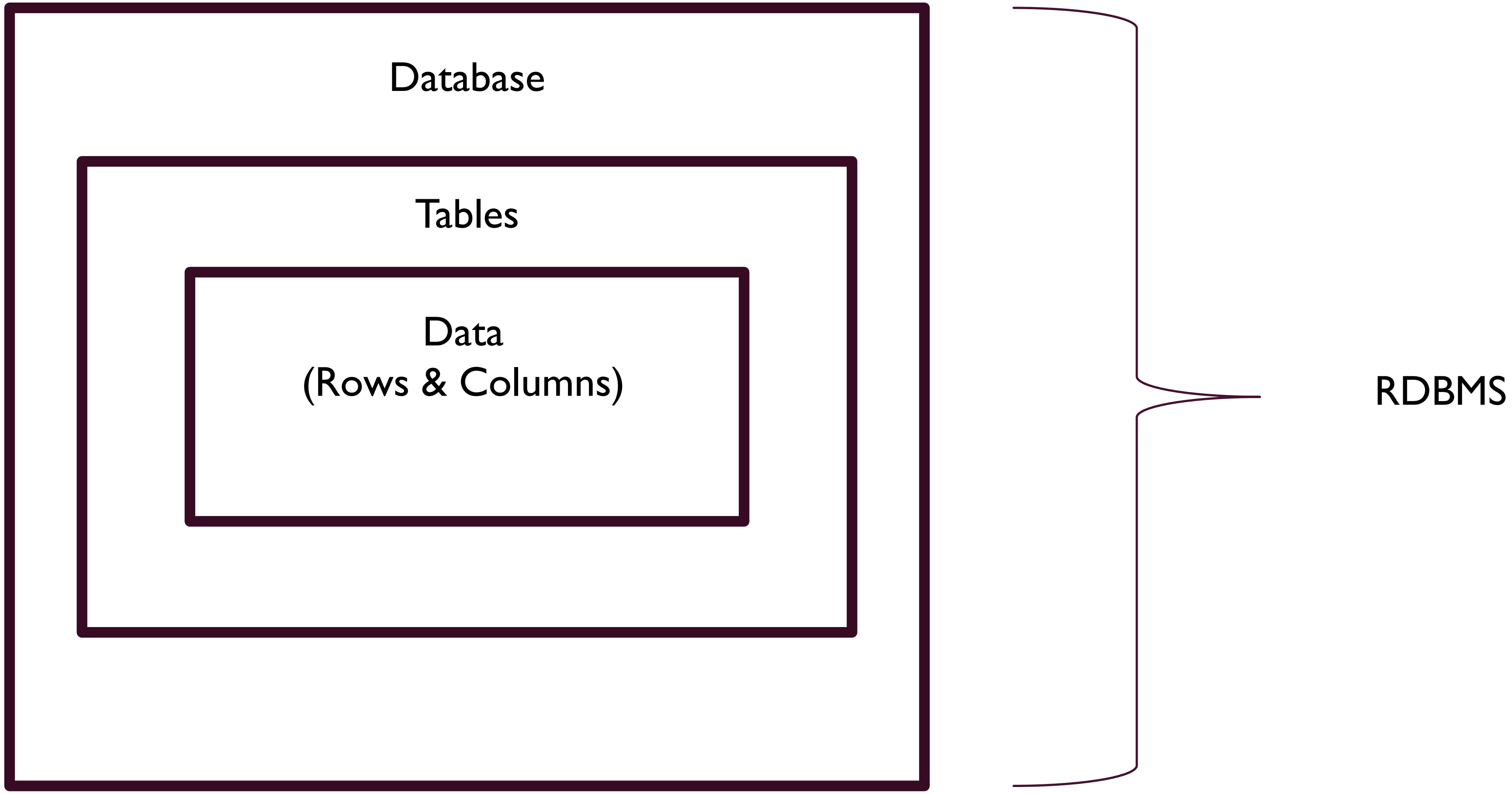
EXCEL V/S DATABASE

EXCEL	DATABASE
Excel is easy to use as untrained person can work.	In database trained person can work.
Excel stores less data.	Database can store large amount of data.
It is good for one time analysis/ quick charts.	Database can automate tasks.
There is no data integrity due to manual operations.	There is high data integrity.
There are low search/ filter capabilities.	There are high search/ filter capabilities.

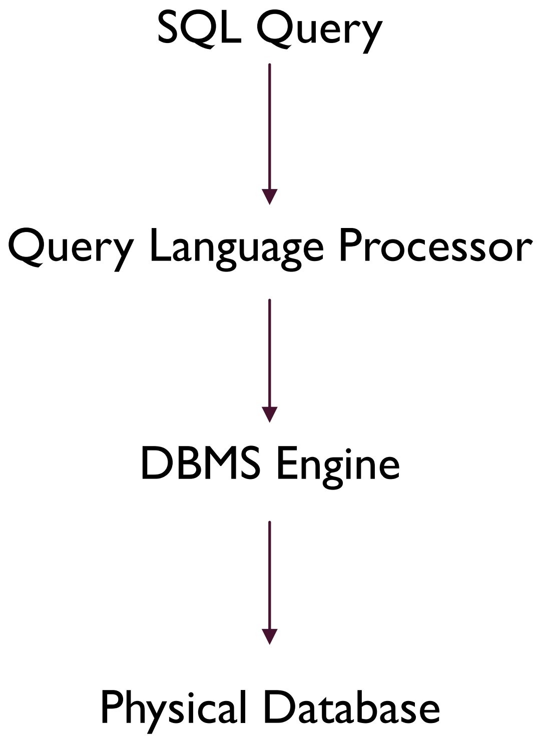
DIFFERENT SQL DATABASES



STRUCTURE OF SQL



FLOW OF SQL



SQL Query

Query Language Processor

DBMS Engine

Physical Database

The flow of SQL begins with a SQL query issued by a user or application.
The query is processed by the Query Language Processor, which parses and analyzes it.
The parsed query is then handed over to the DBMS engine, which includes components like the query optimizer, execution engine, and transaction manager.
The execution engine interacts with the physical database to retrieve or modify data as per the query, and the results are returned to the user or application.

CREATING DATABASE & TABLES IN SQL

SQL Series Part 2

-Mayuri Dandekar

CREATING DATABASE

A database is a collection of related tables, queries and views etc

To create a database in MySQL, we use the **CREATE DATABASE** keywords. A keyword is a word that has a predefined meaning in SQL. In other words, if you want to create a database, you have to type CREATE DATABASE, you cannot be creative and type other words like MAKE DATABASE or CREATE COLLECTION etc.

Keywords are generally **not case sensitive in SQL**. Hence, you can also write create database or CREATE DATABASE.

Syntax-

```
CREATE DATABASE name_of_database;
```

VIEW DATABASE

We have to let the DBMS know that we want to **use this database**.

This is because the DBMS may be managing more than one databases concurrently.

We have to let it know that all subsequent code that we write applies to the stated database.

Syntax:

```
USE name_of_database;
```

DELETE/DROP DATABASE

If after you create your database, you realize that you have typed the name wrongly. There is no easy way to rename a database in MySQL.

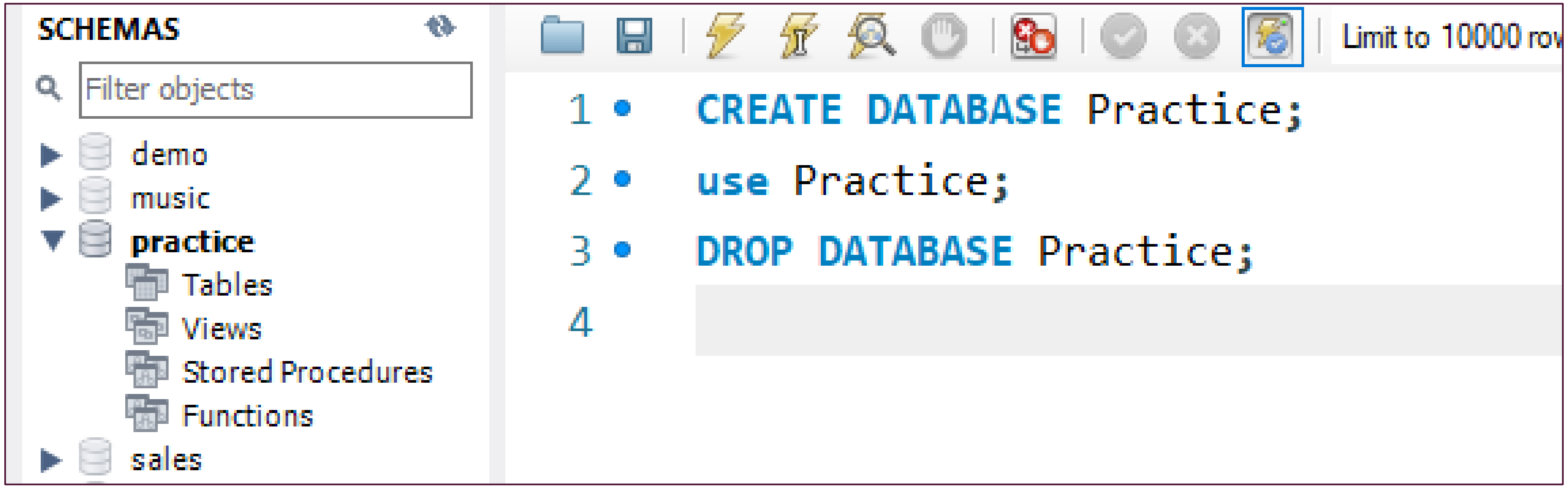
What you can do is create a new database and delete the old database.

Syntax-

```
DROP DATABASE [IF EXISTS] name_of_database;
```

When deleting a database, the **IF EXISTS keywords are optional**. We use them to prevent an error from occurring when we accidentally try to delete a database that does not exist.

OUTPUT



```
CREATE DATABASE Practice;
use Practice;
DROP DATABASE Practice;
```

CREATE TABLES

The CREATE TABLE statement in SQL is used to **create a new table** in a database.

Syntax

CREATE TABLE table_name (

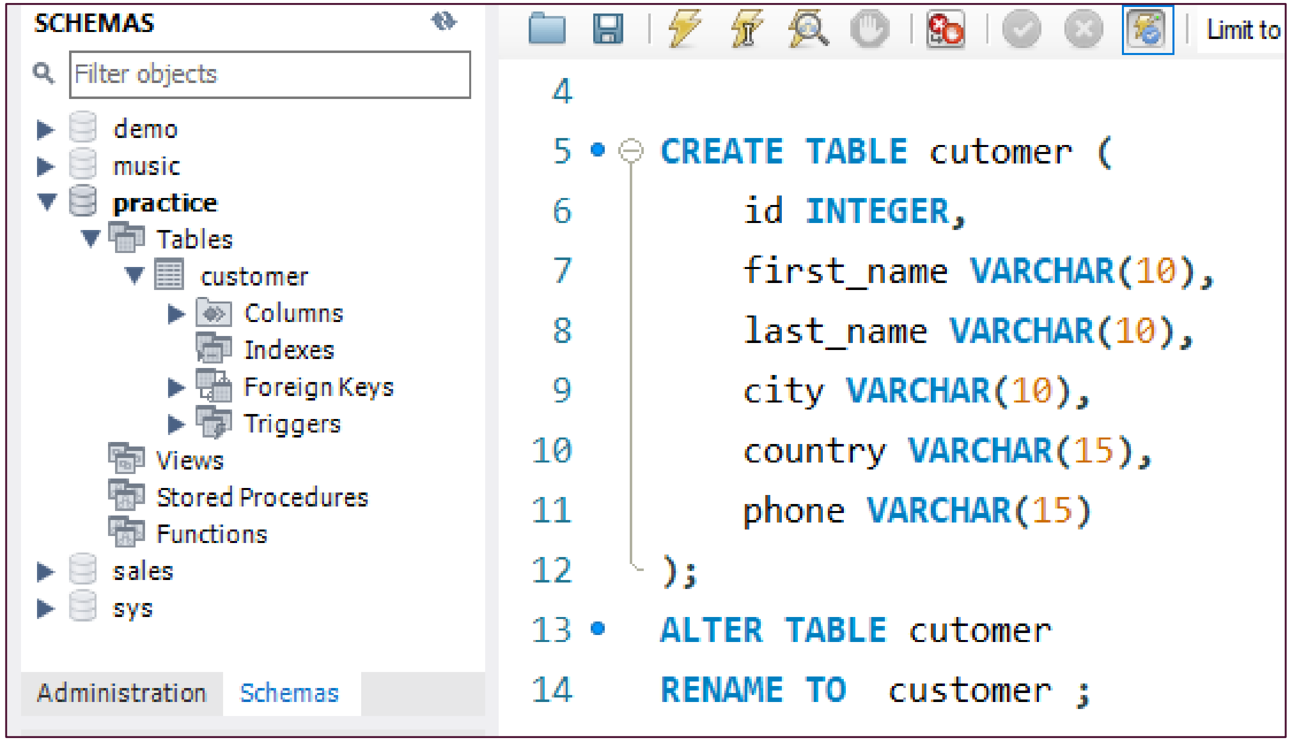
column1 data_type,

column2 data_type,

column3 data_type,

....

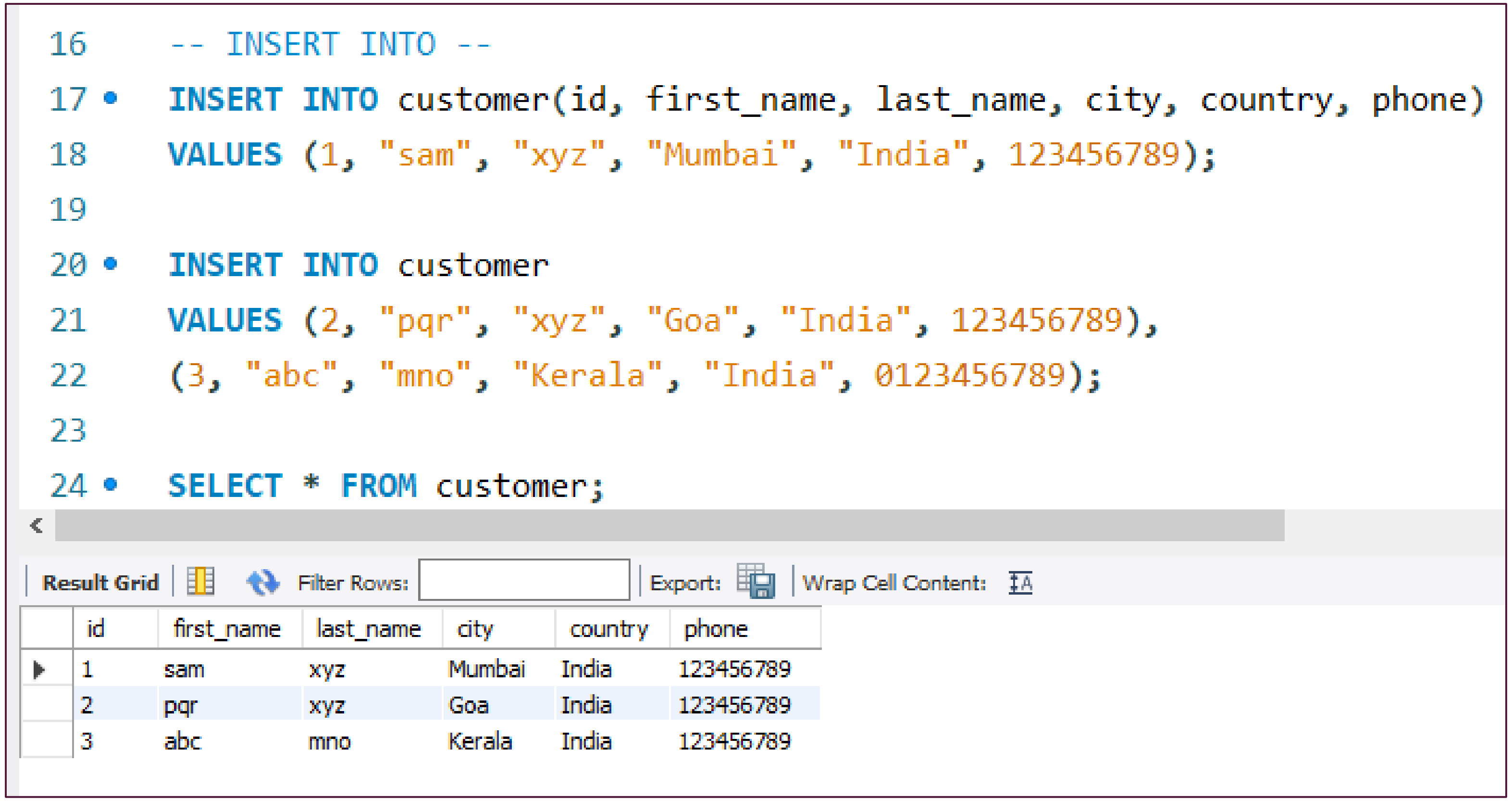
);



```sql

CREATE TABLE cutomer (
      id INTEGER,
      first_name VARCHAR(10),
      last_name VARCHAR(10),
      city VARCHAR(10),
      country VARCHAR(15),
      phone VARCHAR(15)
);
ALTER TABLE cutomer
RENAME TO  customer ;
```

INSERT RECORDS IN TABLES

The INSERT INTO statement in SQL is used to **insert new records** in a table. Below are two ways of inserting records.

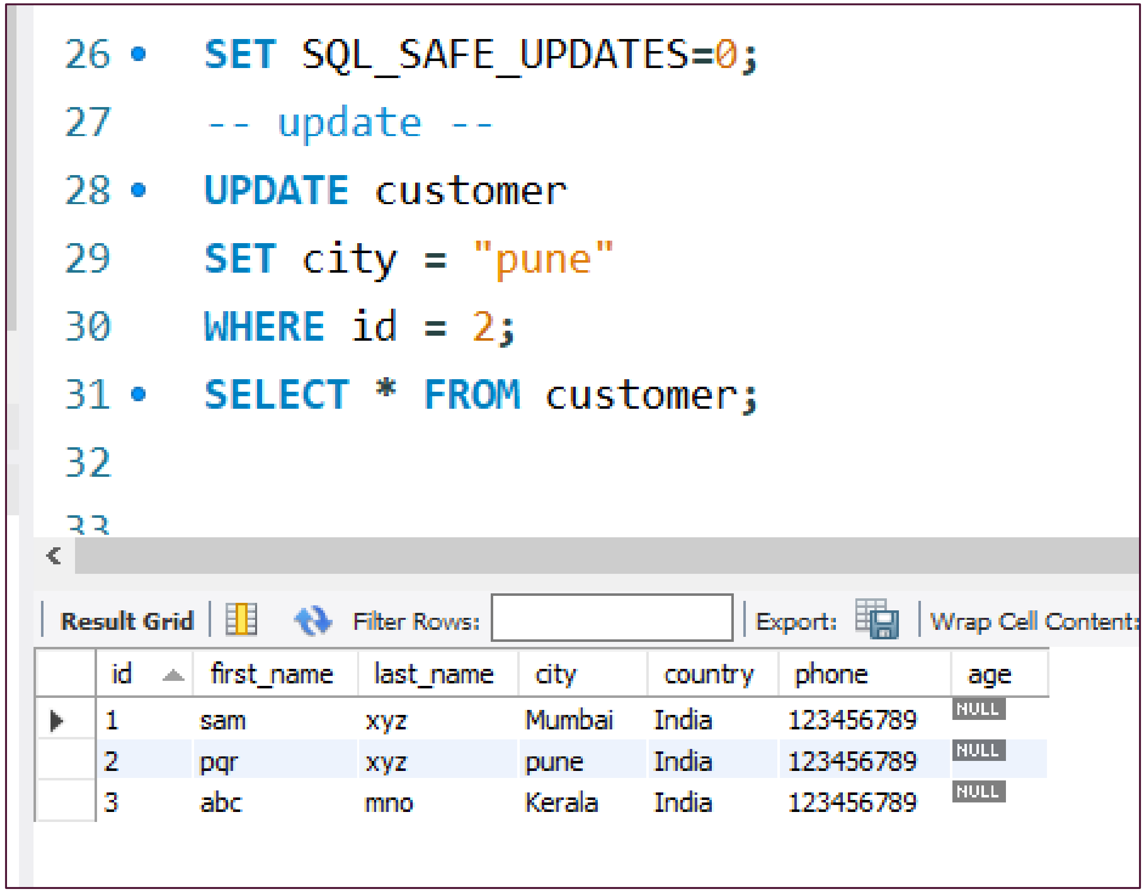


```
-- INSERT INTO --
INSERT INTO customer(id, first_name, last_name, city, country, phone)
VALUES (1, "sam", "xyz", "Mumbai", "India", 123456789);

INSERT INTO customer
VALUES (2, "pqr", "xyz", "Goa", "India", 123456789),
(3, "abc", "mno", "Kerala", "India", 0123456789);

SELECT * FROM customer;
```

id	first_name	last_name	city	country	phone
1	sam	xyz	Mumbai	India	123456789
2	pqr	xyz	Goa	India	123456789
3	abc	mno	Kerala	India	123456789

UPDATE RECORDS IN TABLES

The UPDATE statement in SQL is used to **modify the existing records** in a table

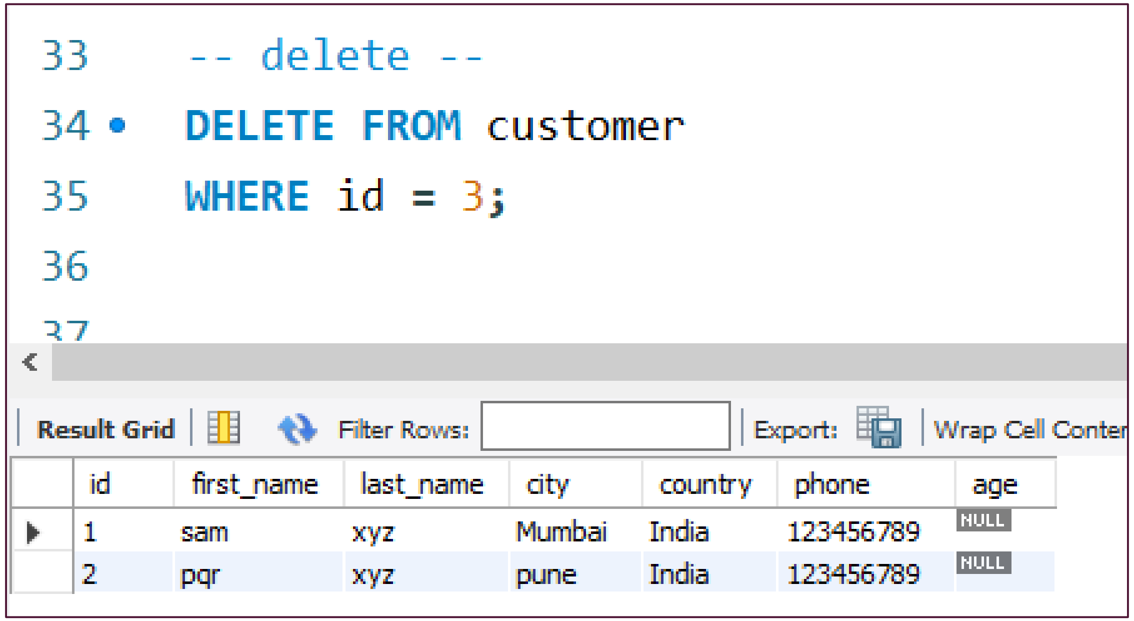


```
SET SQL_SAFE_UPDATES=0;
-- update --
UPDATE customer
SET city = "pune"
WHERE id = 2;
SELECT * FROM customer;
```

id	first_name	last_name	city	country	phone	age
1	sam	xyz	Mumbai	India	123456789	NULL
2	pqr	xyz	pune	India	123456789	NULL
3	abc	mno	Kerala	India	123456789	NULL

DELETE RECORDS IN TABLES

The DELETE statement is used to **delete existing records** in a table

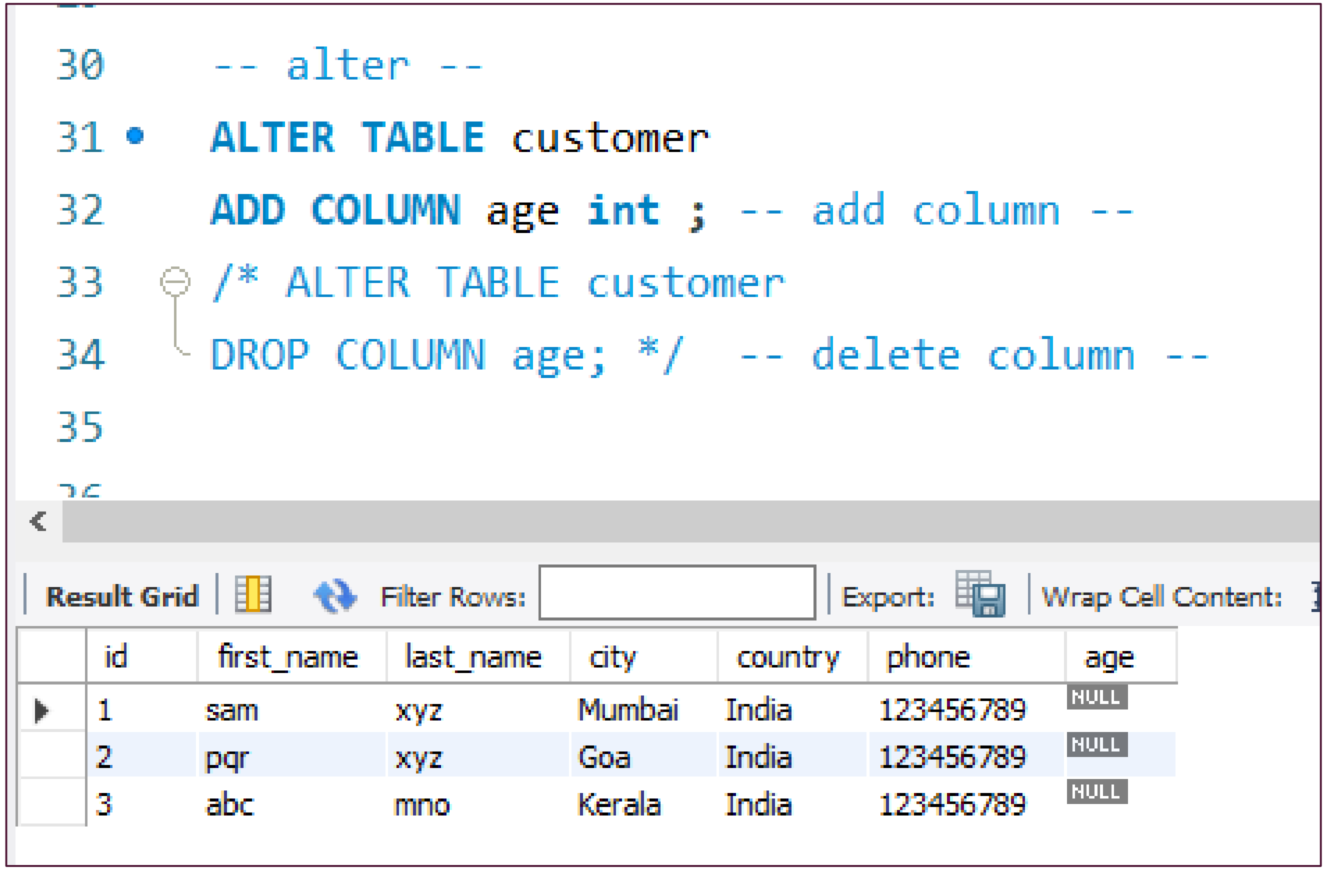


```sql
-- delete --
DELETE FROM customer
WHERE id = 3;
```

id	first_name	last_name	city	country	phone	age
1	sam	xyz	Mumbai	India	123456789	NULL
2	pqr	xyz	pune	India	123456789	NULL

ALTER TABLES

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table

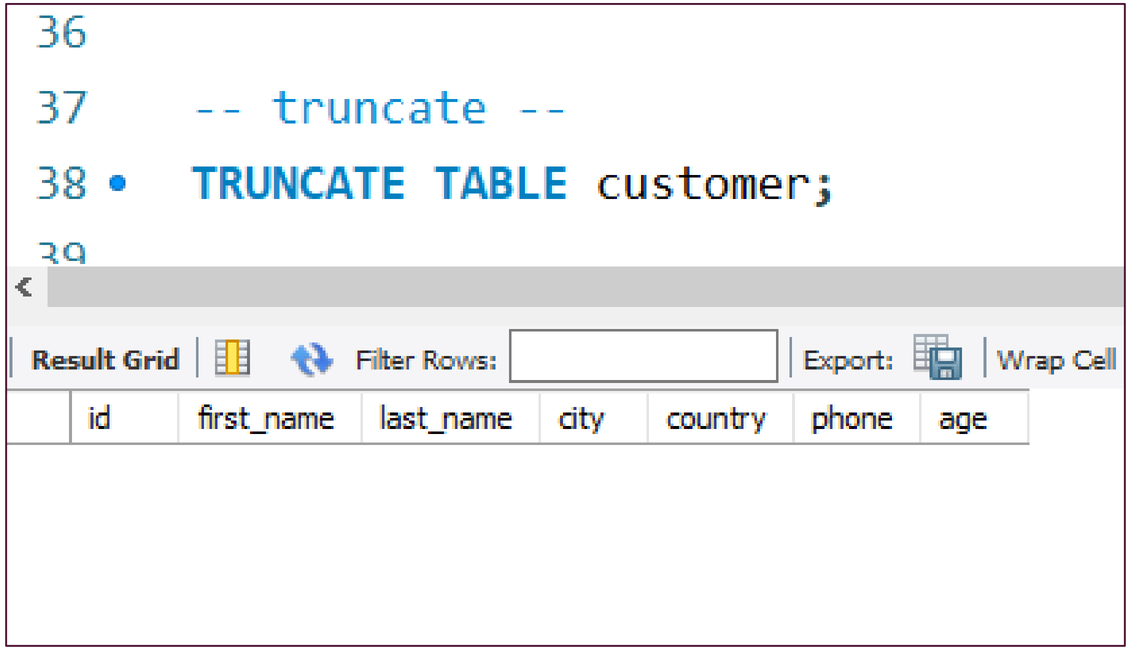


```sql
-- alter --
ALTER TABLE customer
ADD COLUMN age int ; -- add column --
/* ALTER TABLE customer
DROP COLUMN age; */  -- delete column --
```

id	first_name	last_name	city	country	phone	age
1	sam	xyz	Mumbai	India	123456789	NULL
2	pqr	xyz	Goa	India	123456789	NULL
3	abc	mno	Kerala	India	123456789	NULL

TRUNCATE TABLES

The TRUNCATE TABLE command **deletes the data inside a table, but not the table** itself

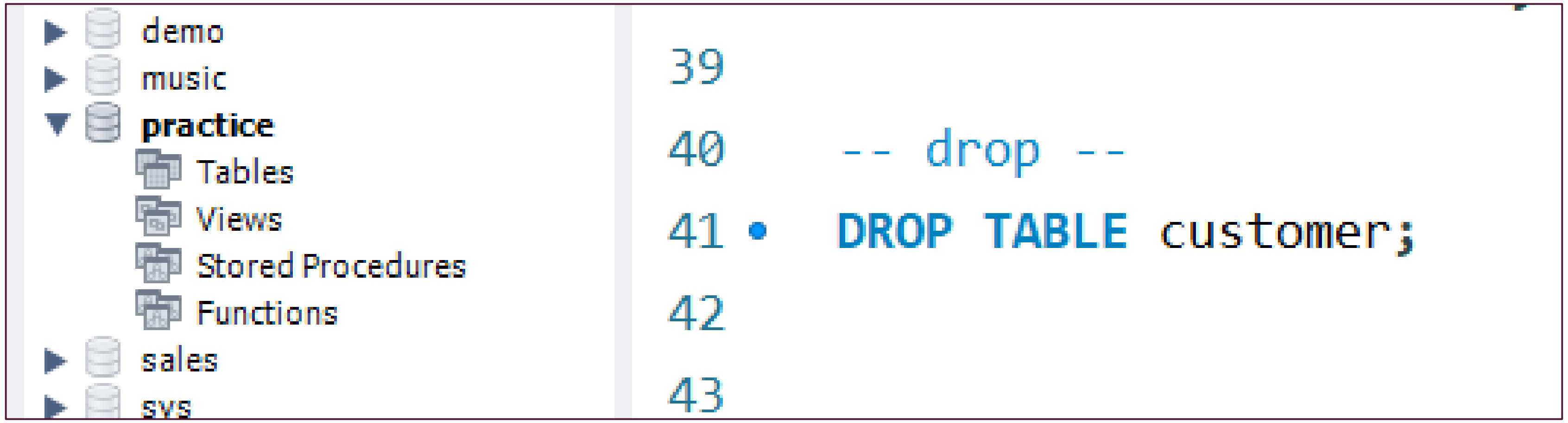


```
-- truncate --
TRUNCATE TABLE customer;
```

id	first_name	last_name	city	country	phone	age

DROP TABLES

The DROP TABLE command **deletes a table** in the database



```
-- drop --
DROP TABLE customer;
```

DATA TYPES, CONSTRAINTS IN SQL

SQL Series Part 3

-Mayuri Dandekar

WHAT IS DATA TYPE?

Data type of a column defines what **value the column can store** in table.

Data types are defined while creating tables in database.

Data types are mainly classified into **three categories**

- **String**: char, varchar, etc
- **Numeric**: int, float, bool, etc
- **Date and time**: date, datetime, etc

COMMONLY USED DATA TYPES

- **Int**: used for the integer value (1,2,3,…)
- **Float**: used to specify a decimal point number (1.2, 2.5, 5.0,…)
- **Bool**: used to specify Boolean values true and false
- **Char**: fixed length string that can contain numbers, letters, and special characters
- **Varchar**: variable length string that can contain numbers, letters, and special characters
- **Date**: date format YYYY-MM-DD
- **Datetime**: date & time combination, format is YYYY-MM-DD hh:mm:ss

WHAT IS CONSTRAINTS?

Constraints are used to **specify rules for data** in a table. This ensures the **accuracy and reliability** of the data in the table

Constraints can be specified when the table is created with the **CREATE TABLE statement**, or after the table is created with the **ALTER TABLE statement.**

If there is any violation between the constraint and the record action, the **action is aborted**.

Constraints can be **column level or table leve**l. Column level constraints apply to a column, and table-level constraints apply to the whole table.

Syntax -

```
CREATE TABLE table_name (
        column1 datatype constraint,
        column2 datatype constraint,
        .... );
```

COMMONLY USED CONSTRAINTS

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE

FOREIGN KEY - Prevents actions that would destroy links between tables (used to link multiple tables together)

CHECK - Ensures that the values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column if no value is specified

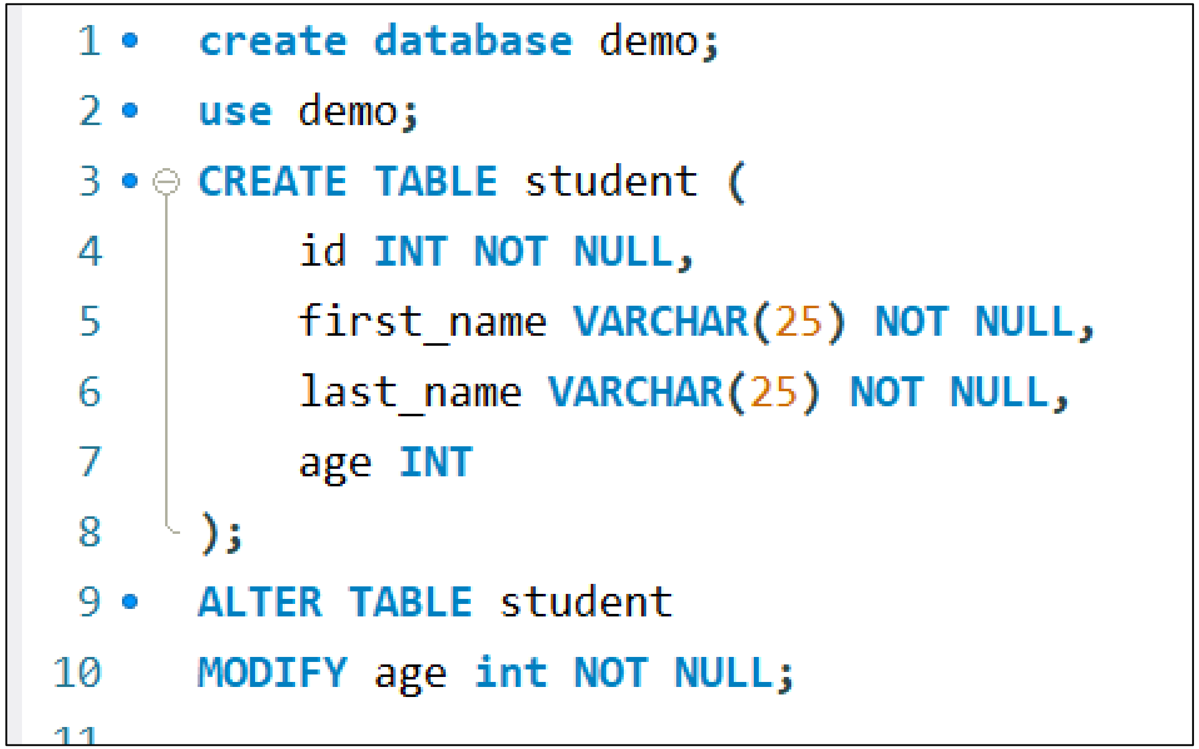
CREATE INDEX - Used to create and retrieve data from the database very quickly

NOT NULL CONSTRAINT

The NOT NULL constraint enforces a column **NOT to accept NULL values**.

This imposes a field always to contain a value, which means that the user cannot insert a new record in a table or update a record without adding a value to this field.

NOTE: By default, a column can hold NULL values



```
create database demo;
use demo;
CREATE TABLE student (
    id INT NOT NULL,
    first_name VARCHAR(25) NOT NULL,
    last_name VARCHAR(25) NOT NULL,
    age INT
);
ALTER TABLE student
MODIFY age int NOT NULL;
```

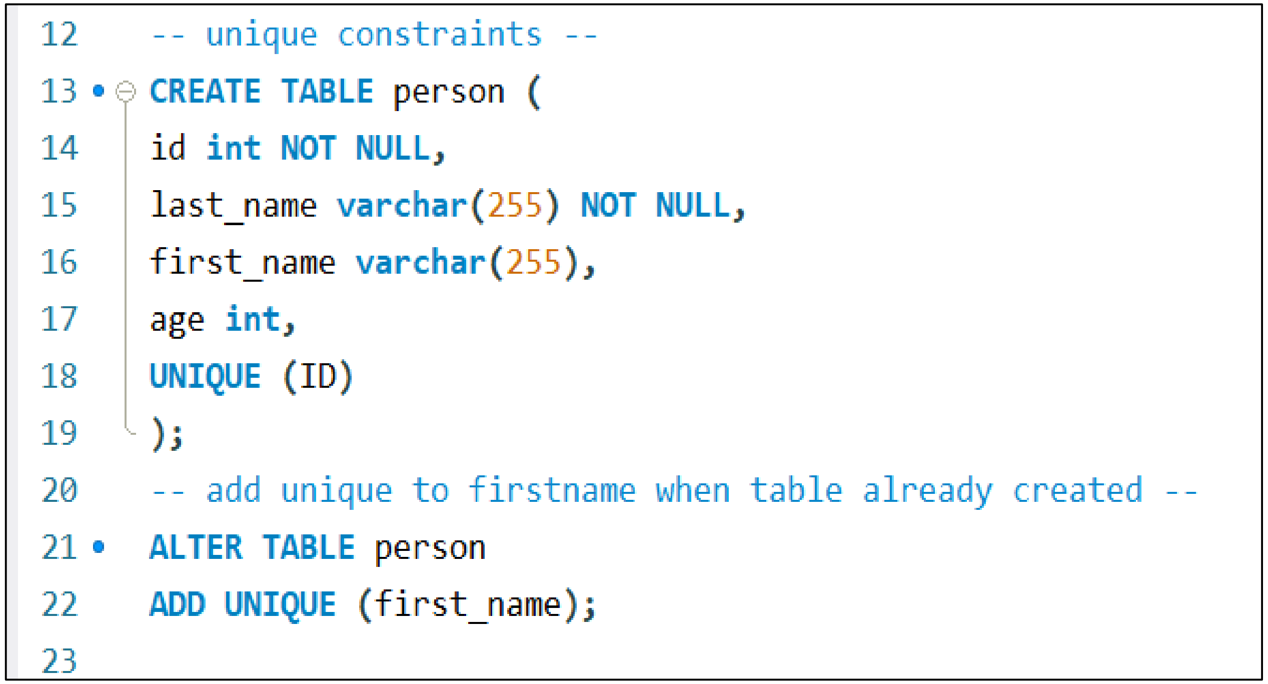
UNIQUE CONSTRAINT

The UNIQUE constraint in SQL ensures that all **values in a column are distinct**.

UNIQUE and PRIMARY KEY constraints both provides a **guarantee for uniqueness** for a column or group of columns.

A PRIMARY KEY constraint, by default, has a UNIQUE constraint.

However, the user can have **many UNIQUE constraints per table**, but **only one PRIMARY KEY** constraint per table.



```
-- unique constraints --
CREATE TABLE person (
id int NOT NULL,
last_name varchar(255) NOT NULL,
first_name varchar(255),
age int,
UNIQUE (ID)
);
-- add unique to firstname when table already created --
ALTER TABLE person
ADD UNIQUE (first_name);

```

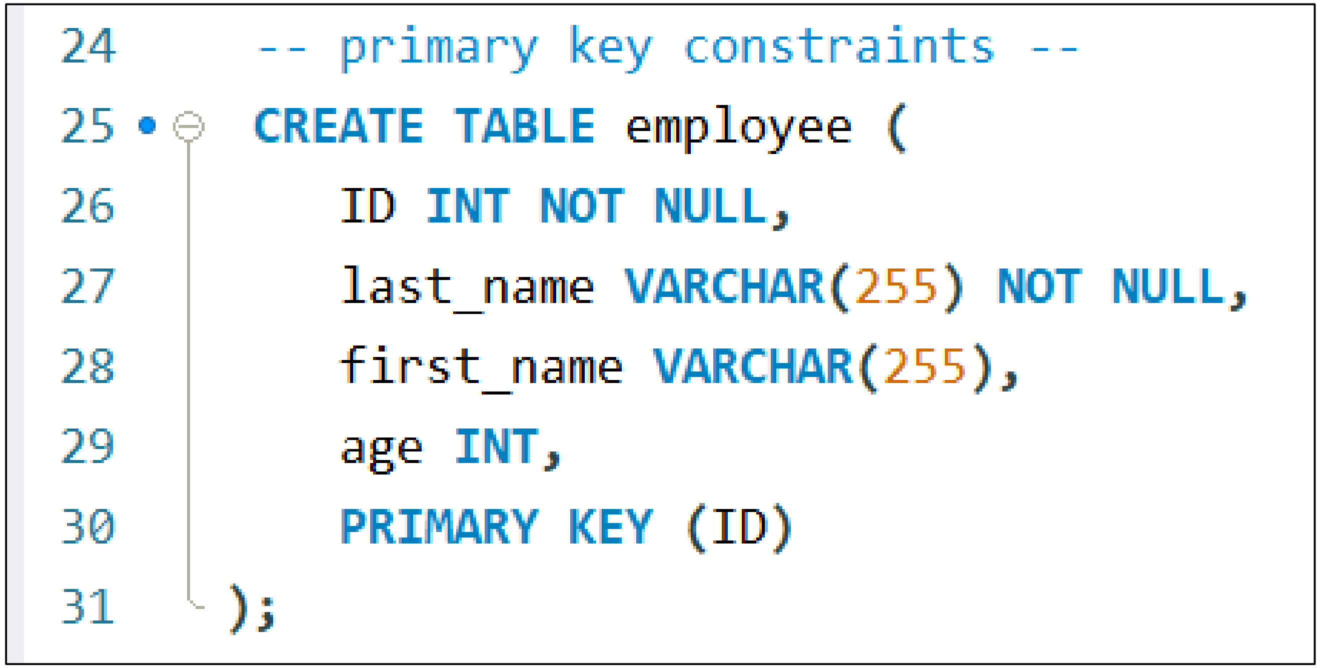
PRIMARY KEY CONSTRAINT

The **PRIMARY KEY** constraint uniquely identifies each of the records in a table.

Only ONE primary key can have in a table.

And also, in the table, this primary key can consist of **single or multiple columns** (fields).

Primary keys should **contain UNIQUE values**, and **cannot contain NULL values**.



```
-- primary key constraints --
CREATE TABLE employee (
    ID INT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    age INT,
    PRIMARY KEY (ID)
);
```

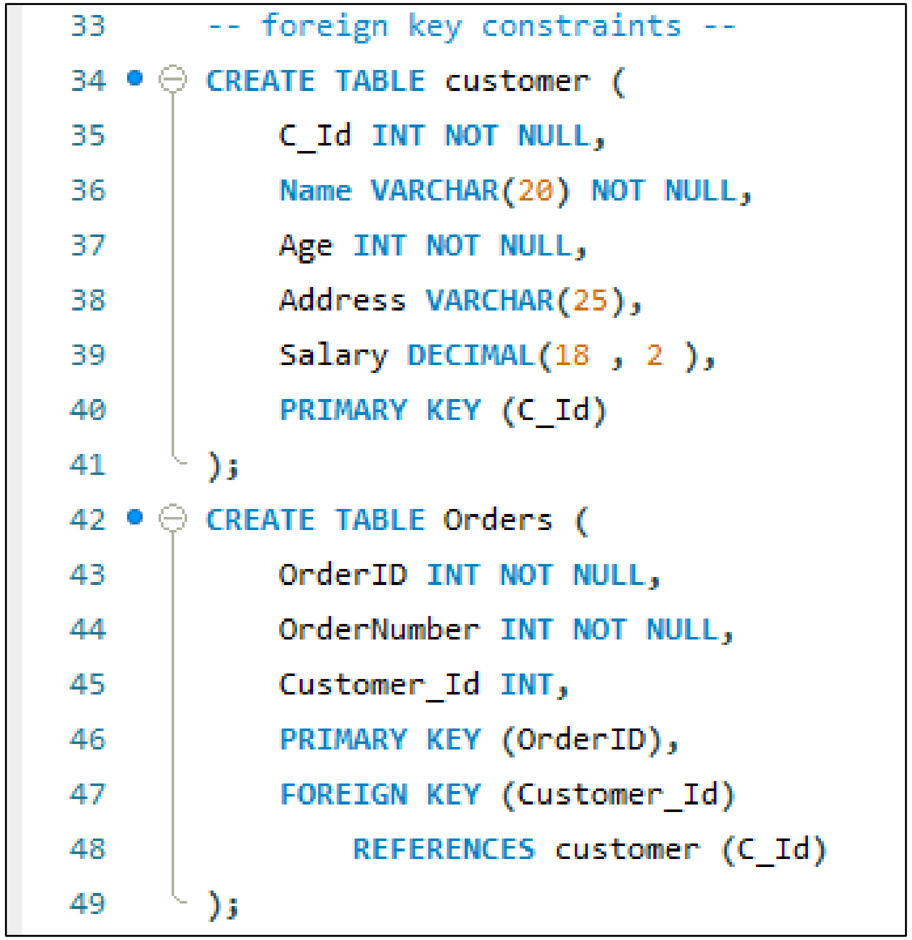
FOREIGN KEY CONSTRAINT

A FOREIGN KEY is used to **link two tables** together. It is also called a **referencing key**.

Foreign Key is a combination of columns (can be single column) whose **value matches a Primary Key** in the different tables.

The relationship between two tables matches the **Primary Key in one of the tables with a Foreign Key in the second table.**

If the table contains a primary key defined on any field, then the user should not have two records having the equal value of that field.

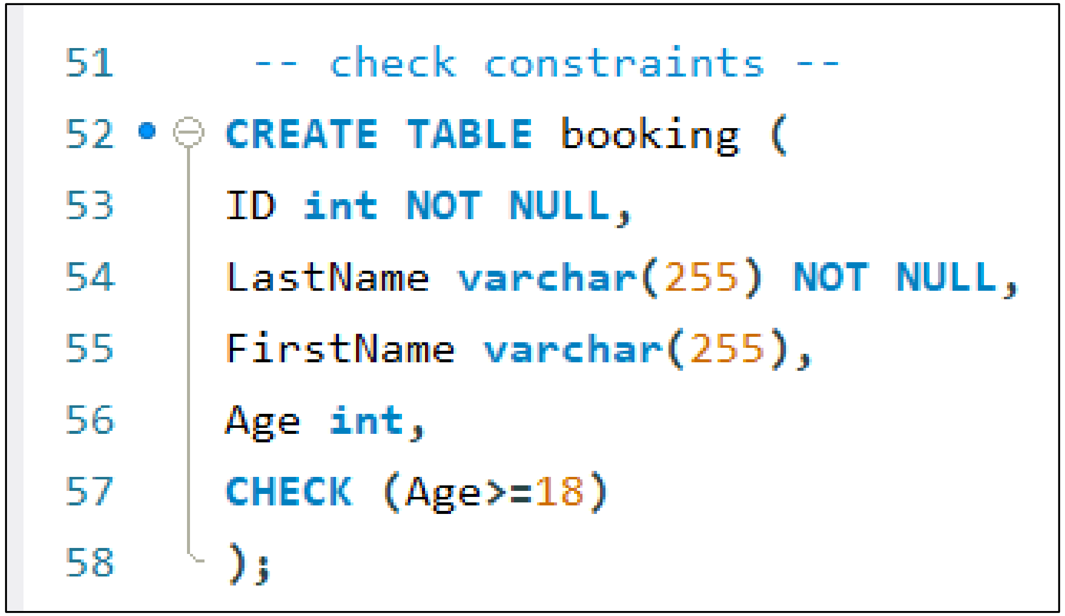


```
-- foreign key constraints --
CREATE TABLE customer (
    C_Id INT NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Age INT NOT NULL,
    Address VARCHAR(25),
    Salary DECIMAL(18 , 2 ),
    PRIMARY KEY (C_Id)
);
CREATE TABLE Orders (
    OrderID INT NOT NULL,
    OrderNumber INT NOT NULL,
    Customer_Id INT,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (Customer_Id)
        REFERENCES customer (C_Id)
);
```

CHECK CONSTRAINT

The CHECK CONSTRAINTS is used to **limit the range of value** that can be placed in a column if the user defines a CHECK constraint on a single column, it **allows only specific values** for the column.

If the user defines a CHECK constraint on a table, it can limit the values in particular columns based on values in another column in the row.

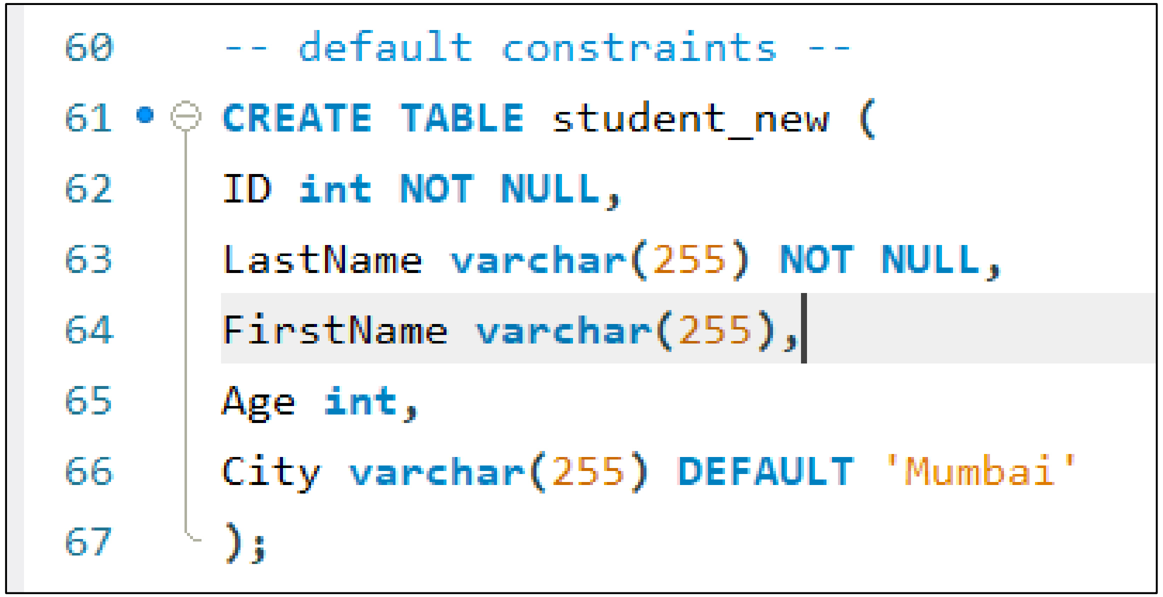


```
 -- check constraints --
CREATE TABLE booking (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int,
CHECK (Age>=18)
);
```

DEFAULT CONSTRAINT

The DEFAULT constraint in SQL is used to provide a **default value for a column** of the table.

The default value will be **added to every new record if no other value is mentioned.**



```sql
-- default constraints --
CREATE TABLE student_new (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int,
City varchar(255) DEFAULT 'Mumbai'
);
```

DEFAULT CONSTRAINT

CREATE INDEX statement in SQL is used to **create indexes in tables**.

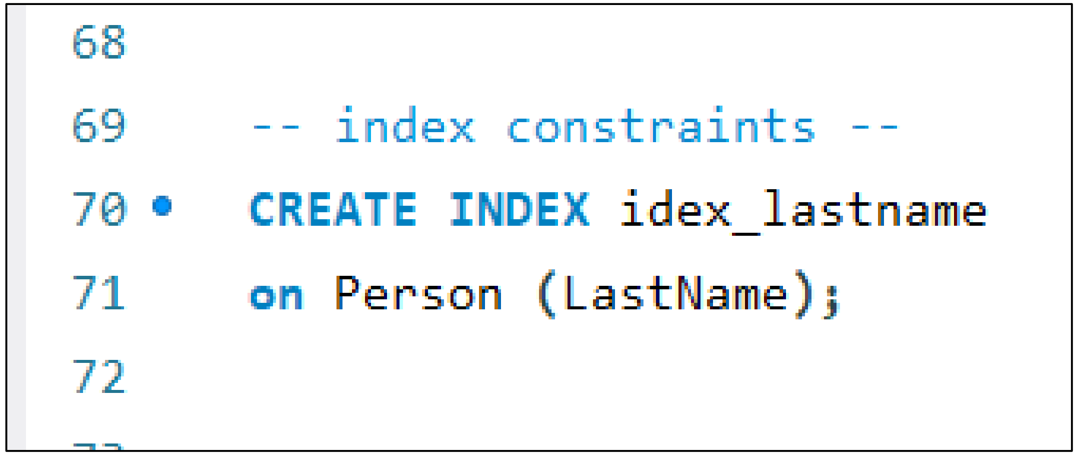
The indexes are used to **retrieve data from the database more quickly than others**.

The user can not see the indexes, and they are just used to **speed up queries /searches**.

Note: Updating the table with indexes takes a lot of time than updating a table without indexes. It is because the indexes also need an update. So, only create indexes on those columns that will be frequently searched against.

Syntax-

CREATE INDEX index_name
ON table_name (column1, column2, ...);



```
-- index constraints --
CREATE INDEX idex_lastname
on Person (LastName);
```

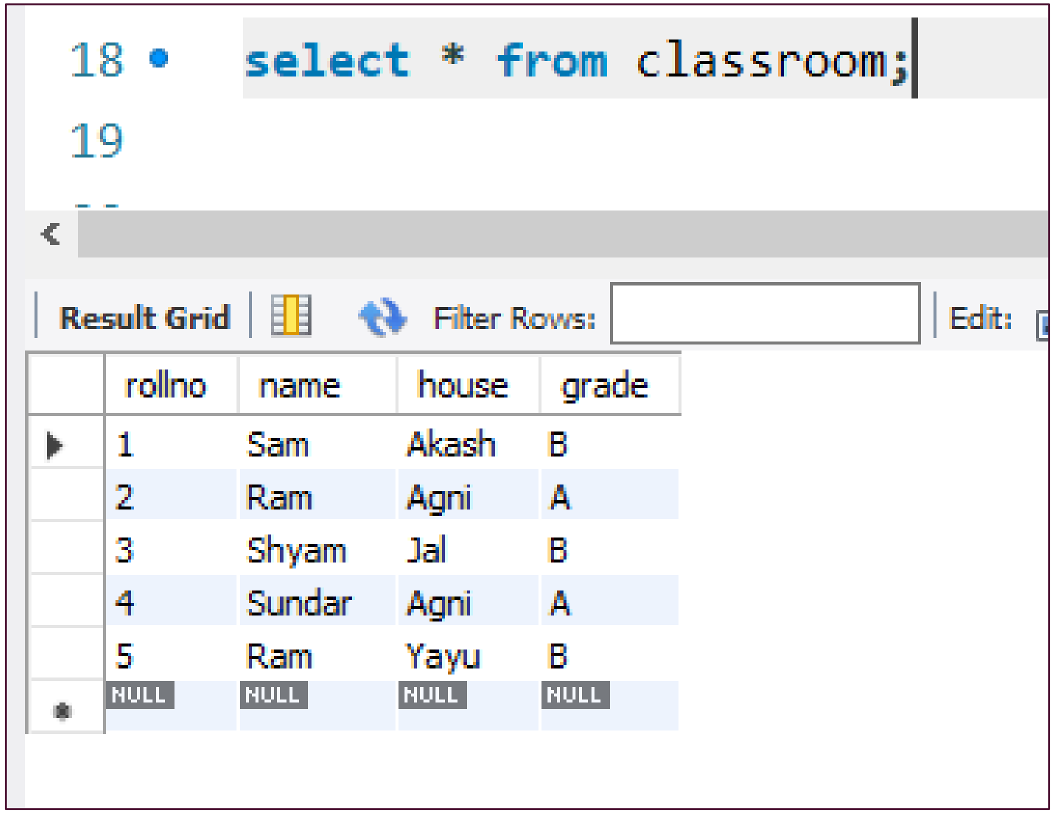
SELECT STATEMENT & WHERE, ORDER BY, LIMIT CLAUSE IN SQL

SQL Series Part 4

-Mayuri Dandekar

SELECT STATEMENT – (SELECT ALL)

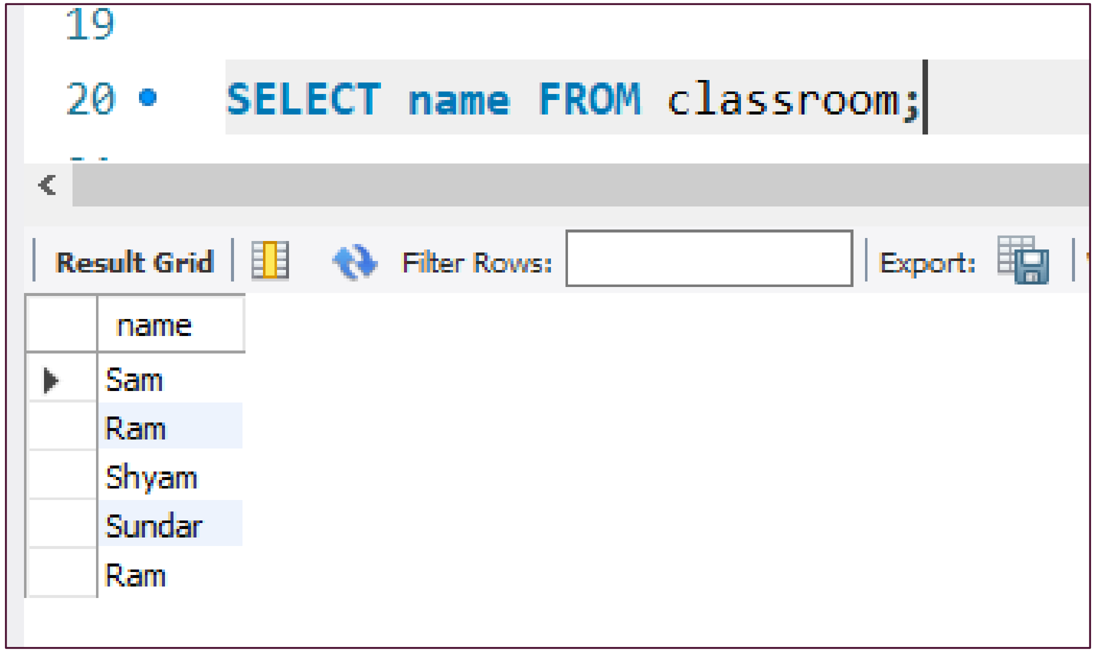
The SELECT statement permits you to read data from tables.



```
select * from classroom;
```

rollno	name	house	grade
1	Sam	Akash	B
2	Ram	Agni	A
3	Shyam	Jal	B
4	Sundar	Agni	A
5	Ram	Yayu	B
NULL	NULL	NULL	NULL

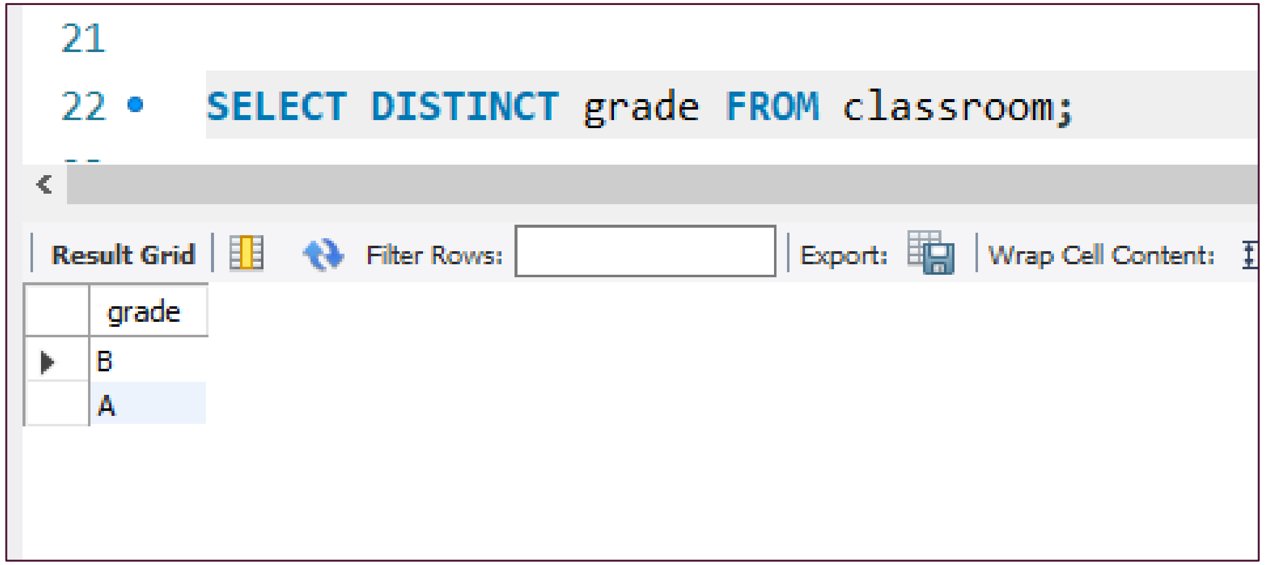
SELECT STATEMENT – (SELECT SPECIFIC COLUMN)



```sql
SELECT name FROM classroom;
```

name
Sam
Ram
Shyam
Sundar
Ram

SELECT STATEMENT – (SELECT DISTINCT FIELDS)

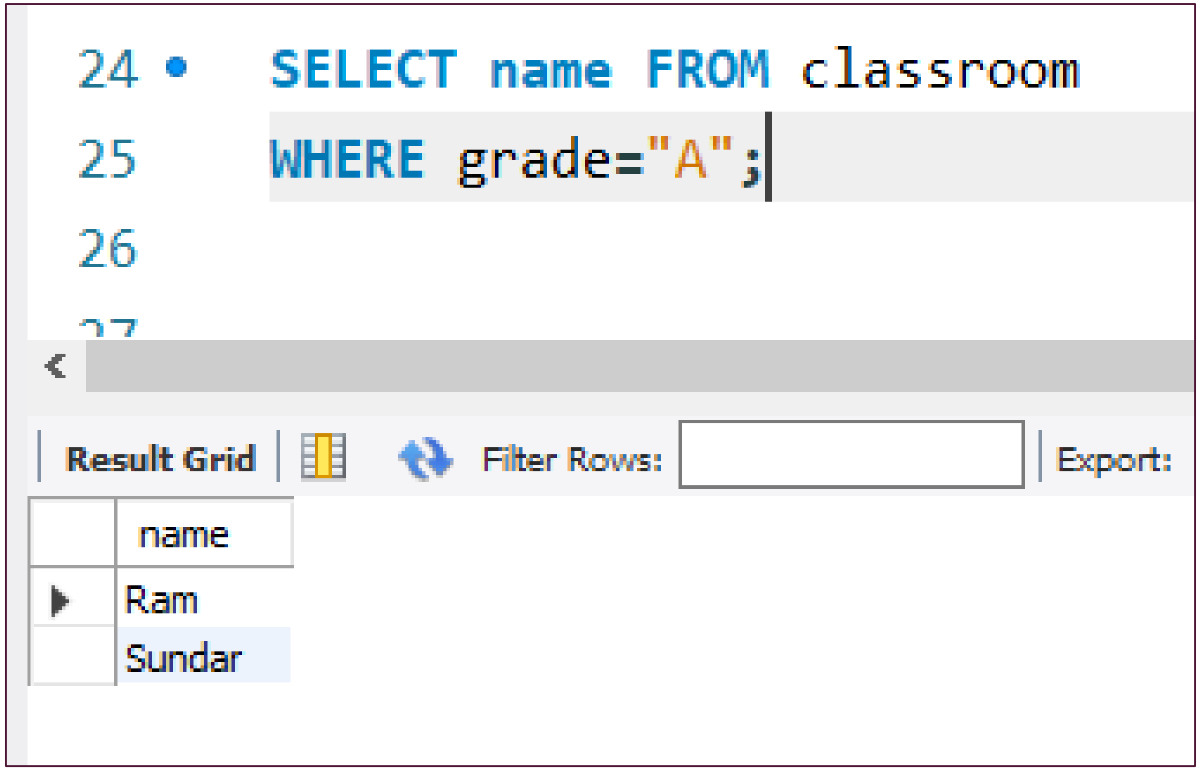


```
SELECT DISTINCT grade FROM classroom;
```

```
grade
B
A
```

WHERE CLAUSE

The WHERE clause allows the user to filter the data from the table. The WHERE clause allows the user to extract only those records that satisfy a specified condition.

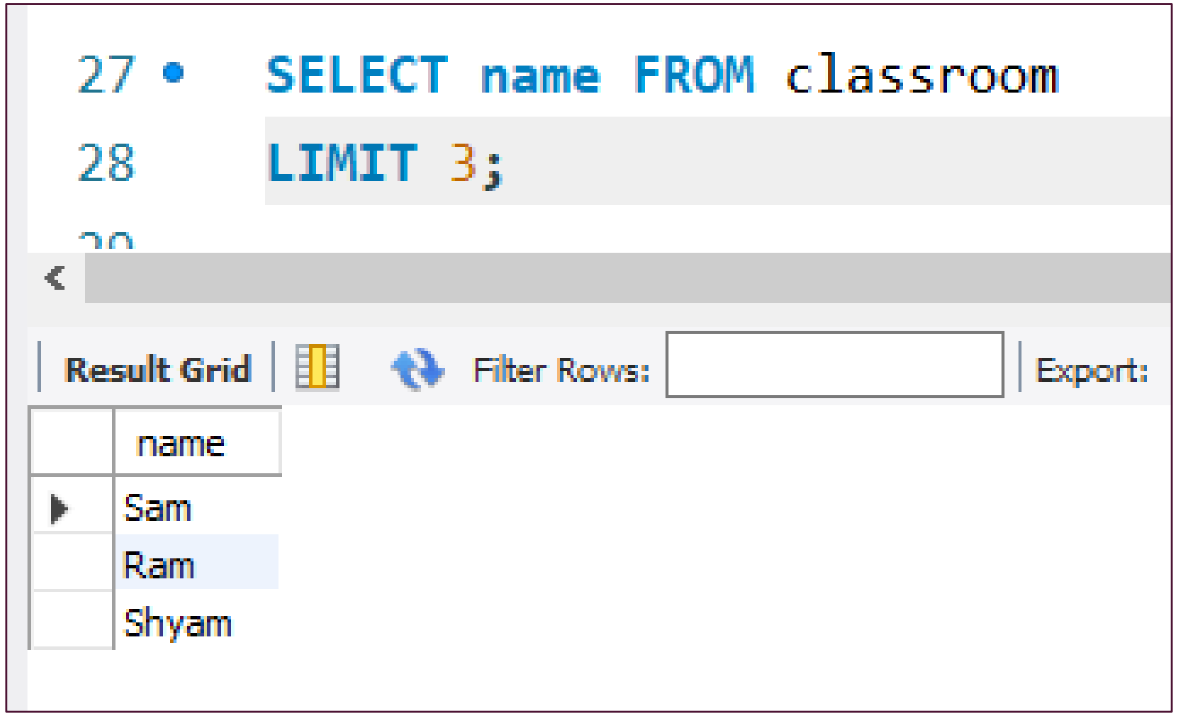


```
SELECT name FROM classroom
WHERE grade="A";
```

```
name
Ram
Sundar
```

LIMIT CLAUSE

The LIMIT clause is used to set an upper limit on the number of tuples returned by SQL.

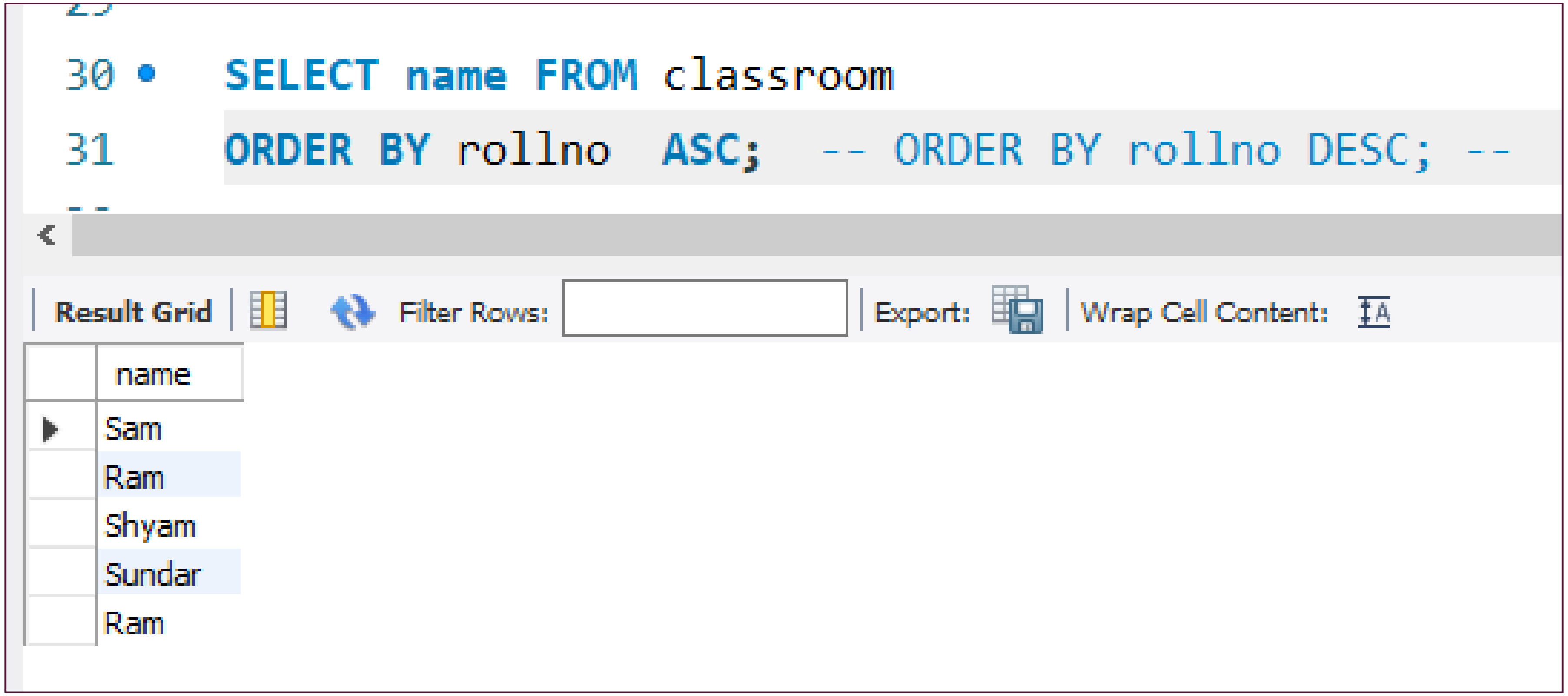


```
SELECT name FROM classroom
LIMIT 3;
```

name
Sam
Ram
Shyam

ORDER BY CLAUSE

The ORDER BY is used to sort the result-set in ascending (ASC) or descending order (DESC).



```sql
SELECT name FROM classroom
ORDER BY rollno  ASC;  -- ORDER BY rollno DESC; --
```

name
Sam
Ram
Shyam
Sundar
Ram

OPERATORS IN SQL

SQL Series Part 5

-Mayuri Dandekar

WHAT IS OPERATORS?

The SQL **reserved words and characters** are called operators, which are **used with a WHERE** clause in a SQL query.

Most used operators:

- **Arithmetic operators** : arithmetic operations on numeric values

  Example: Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%)

- **Comparison operators**: compare two different data of SQL table

  Example: Equal (=), Not Equal (!=), Greater Than (>), Greater Than Equals to (>=)

- **Logical operators**: perform the Boolean operations

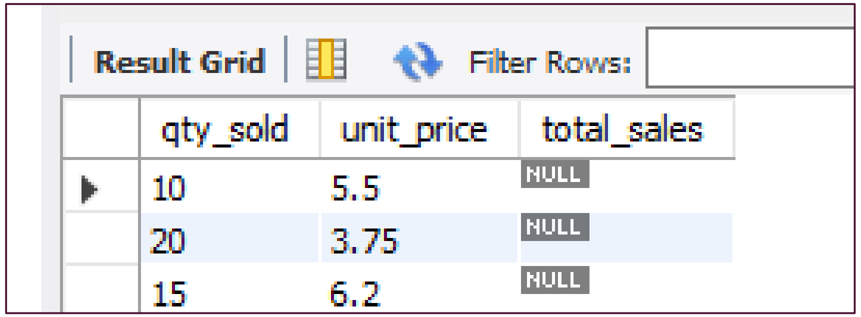
  Example: ALL, IN, BETWEEN, LIKE, AND, OR, NOT, ANY

- **Bitwise operators**: perform the bit operations on the Integer values

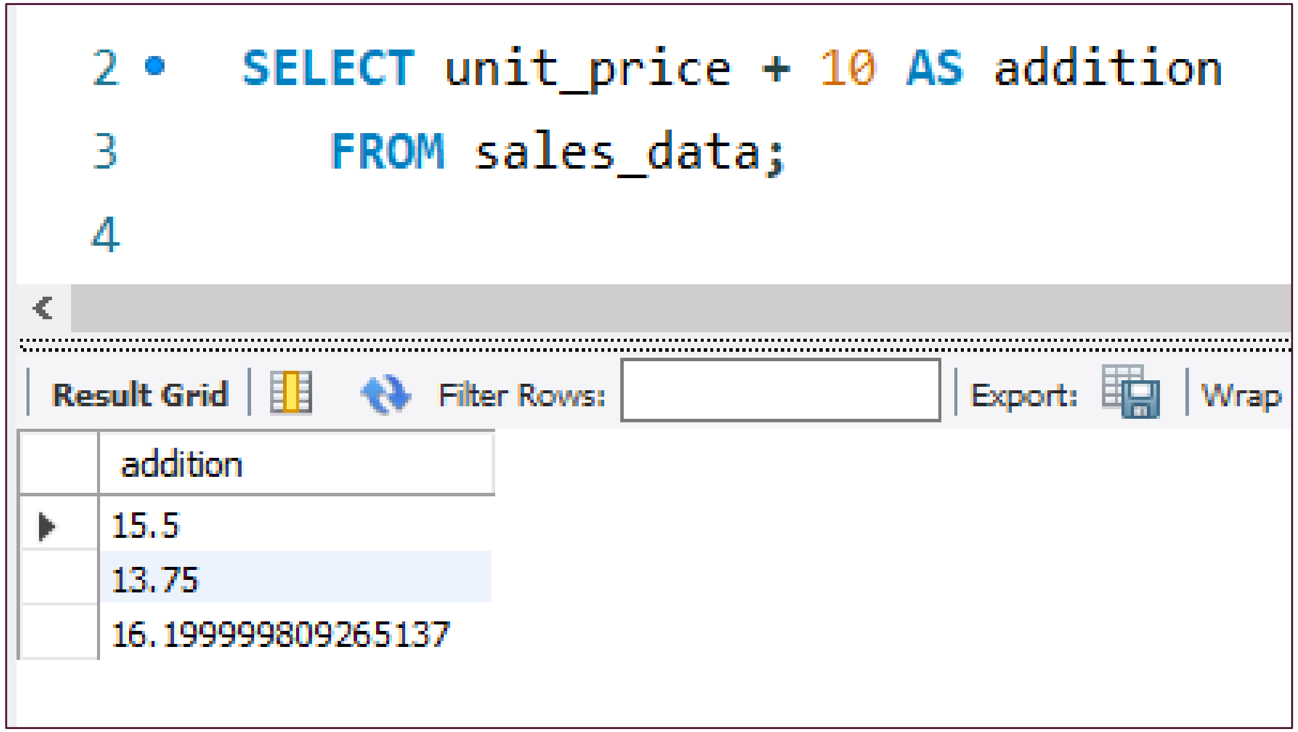
  Example: Bitwise AND (&), Bitwise OR(|)

ARITHMETIC OPERATOR -- ADDITION

Adds two numeric values.



qty_sold	unit_price	total_sales
10	5.5	NULL
20	3.75	NULL
15	6.2	NULL

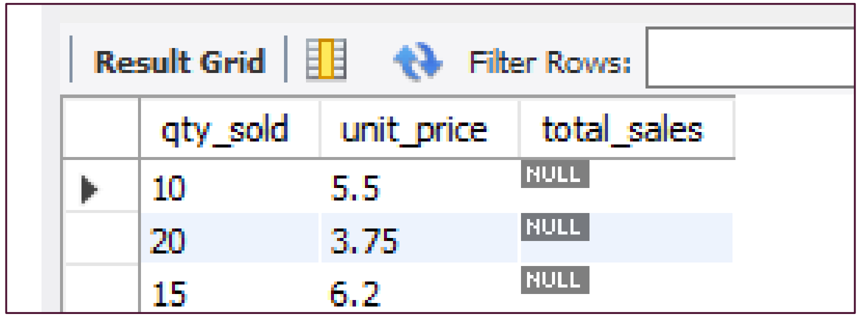


```sql
SELECT unit_price + 10 AS addition
    FROM sales_data;
```

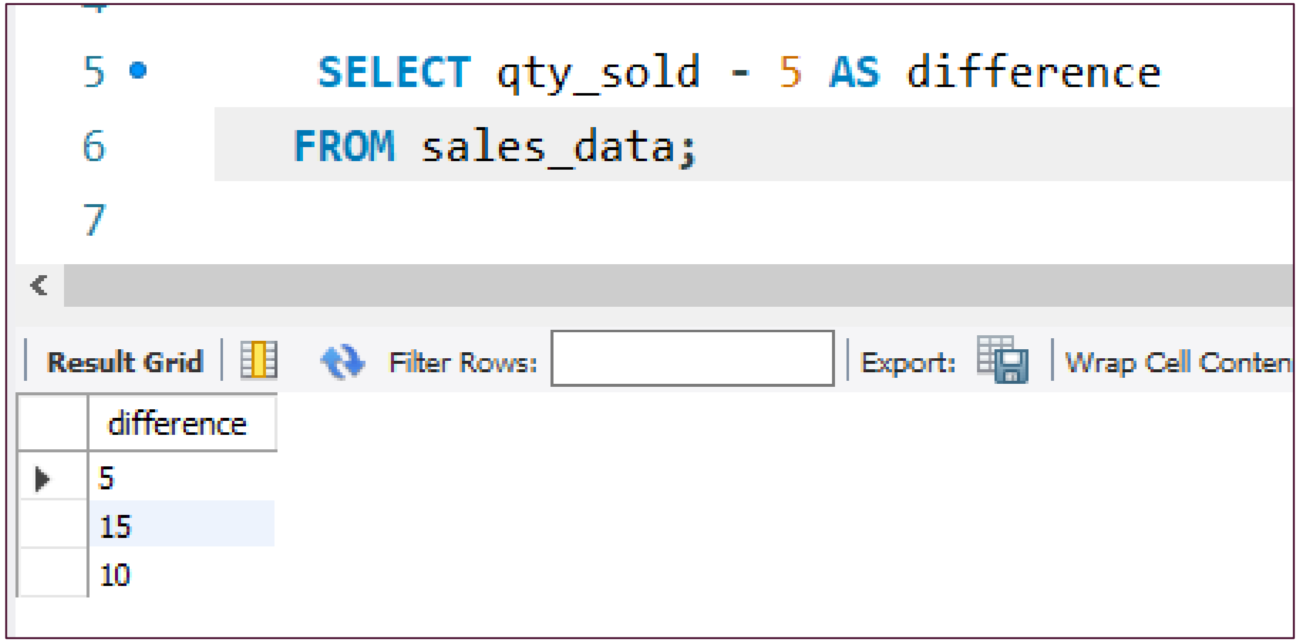
addition
15.5
13.75
16.199999809265137

ARITHMETIC OPERATOR -- SUBTRACTION

Subtracts one numeric value from another.



qty_sold	unit_price	total_sales
10	5.5	NULL
20	3.75	NULL
15	6.2	NULL

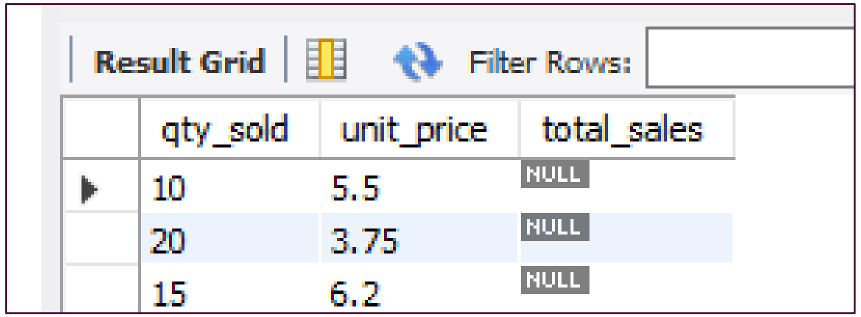


```
        SELECT qty_sold - 5 AS difference
    FROM sales_data;
```

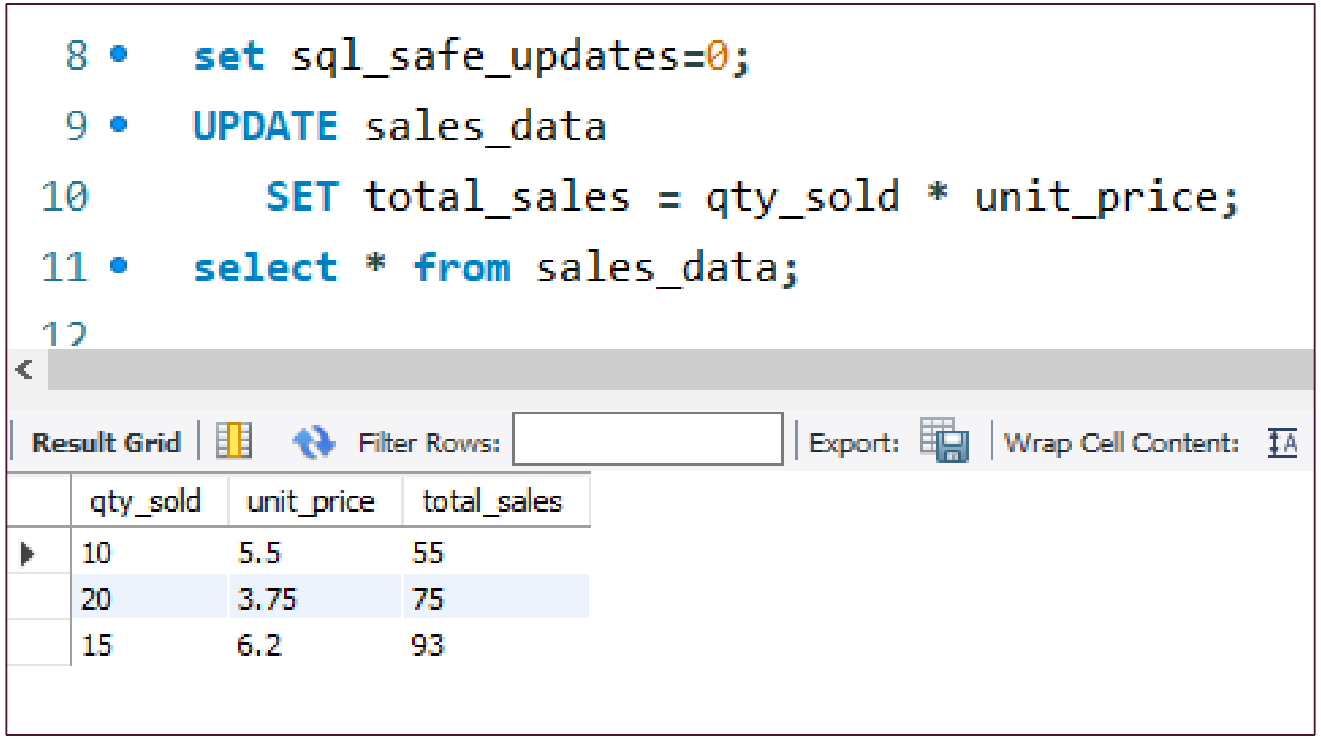
difference
5
15
10

ARITHMETIC OPERATOR -- MULTIPLICATION

Multiplies two numeric values.



qty_sold	unit_price	total_sales
10	5.5	NULL
20	3.75	NULL
15	6.2	NULL

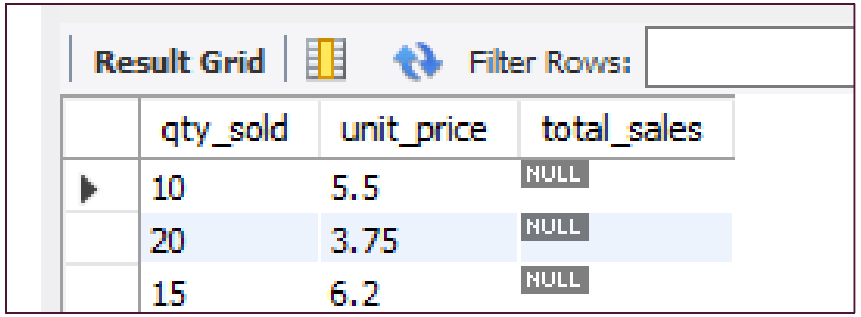


```
set sql_safe_updates=0;
UPDATE sales_data
    SET total_sales = qty_sold * unit_price;
select * from sales_data;
```

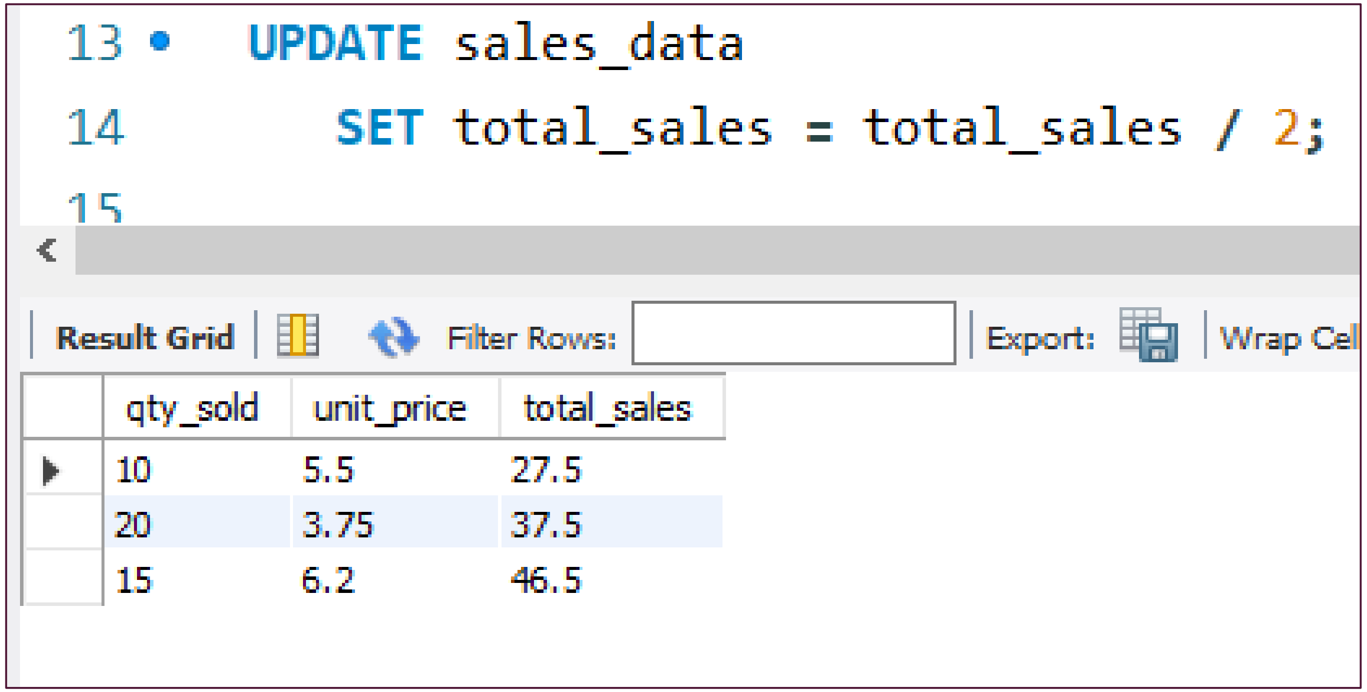
qty_sold	unit_price	total_sales
10	5.5	55
20	3.75	75
15	6.2	93

ARITHMETIC OPERATOR -- DIVISION

Divides one numeric value by another.



qty_sold	unit_price	total_sales
10	5.5	NULL
20	3.75	NULL
15	6.2	NULL

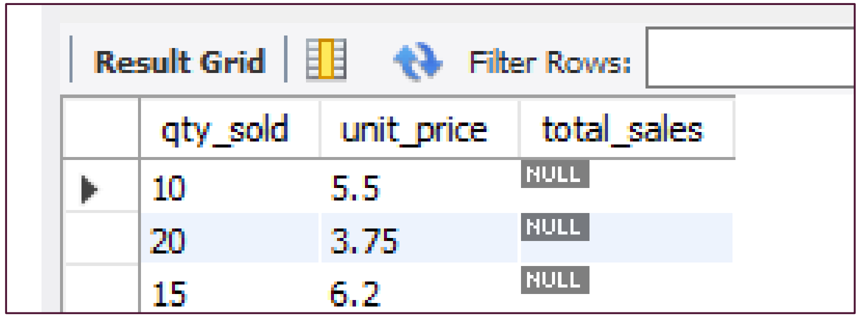


```
UPDATE sales_data
    SET total_sales = total_sales / 2;
```

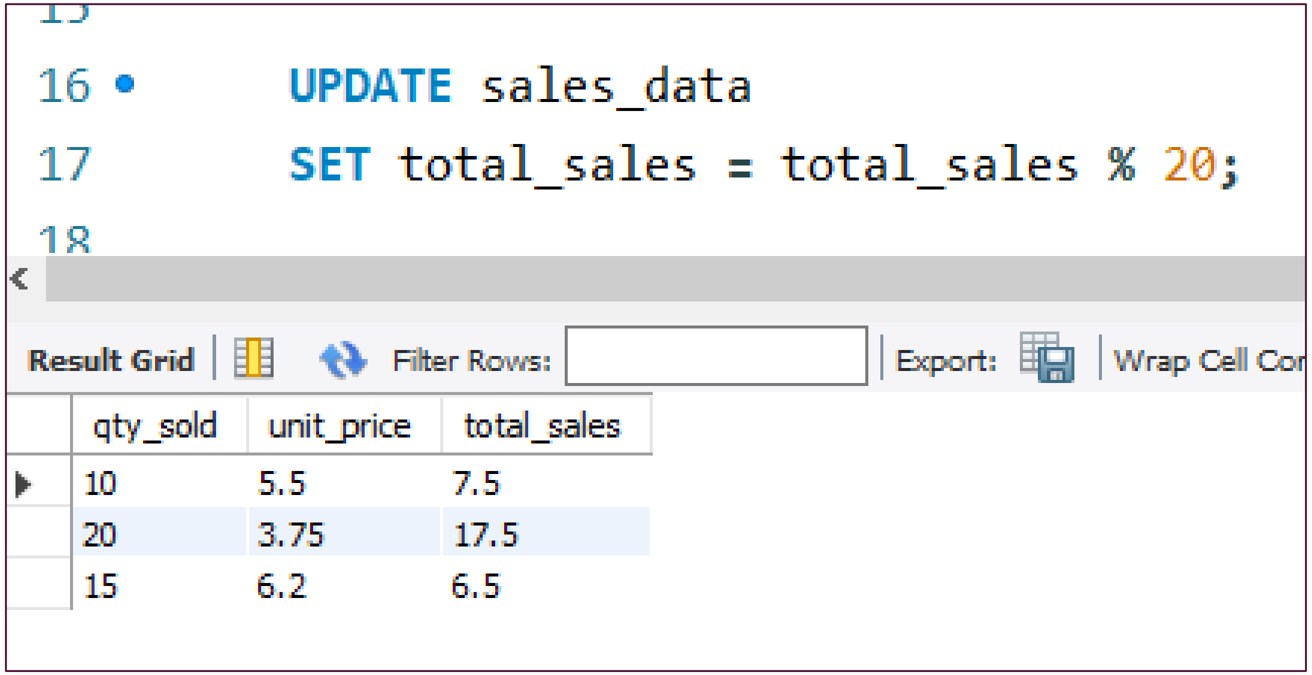
qty_sold	unit_price	total_sales
10	5.5	27.5
20	3.75	37.5
15	6.2	46.5

ARITHMETIC OPERATOR -- MODULUS

Returns the remainder of a division operation.



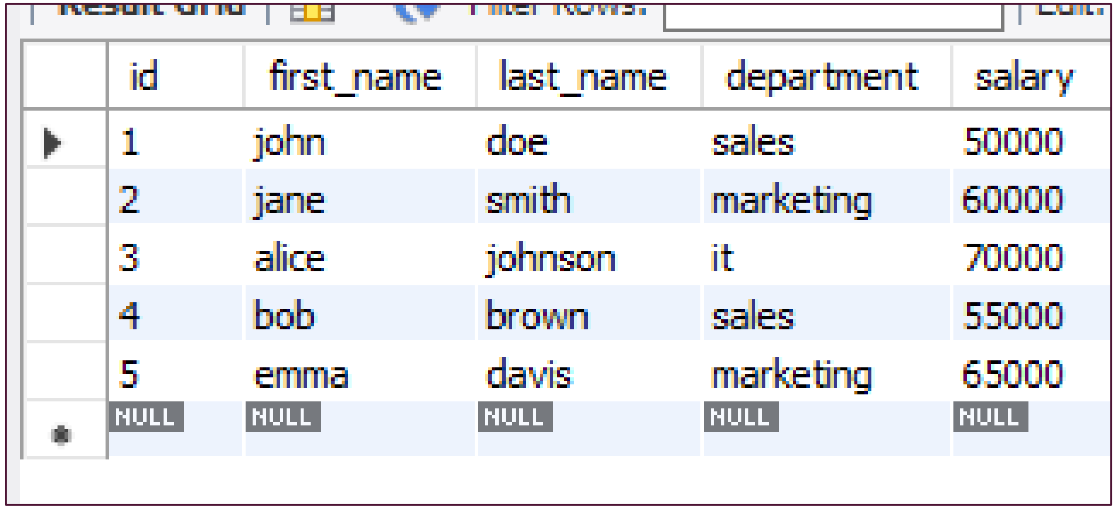
qty_sold	unit_price	total_sales
10	5.5	NULL
20	3.75	NULL
15	6.2	NULL



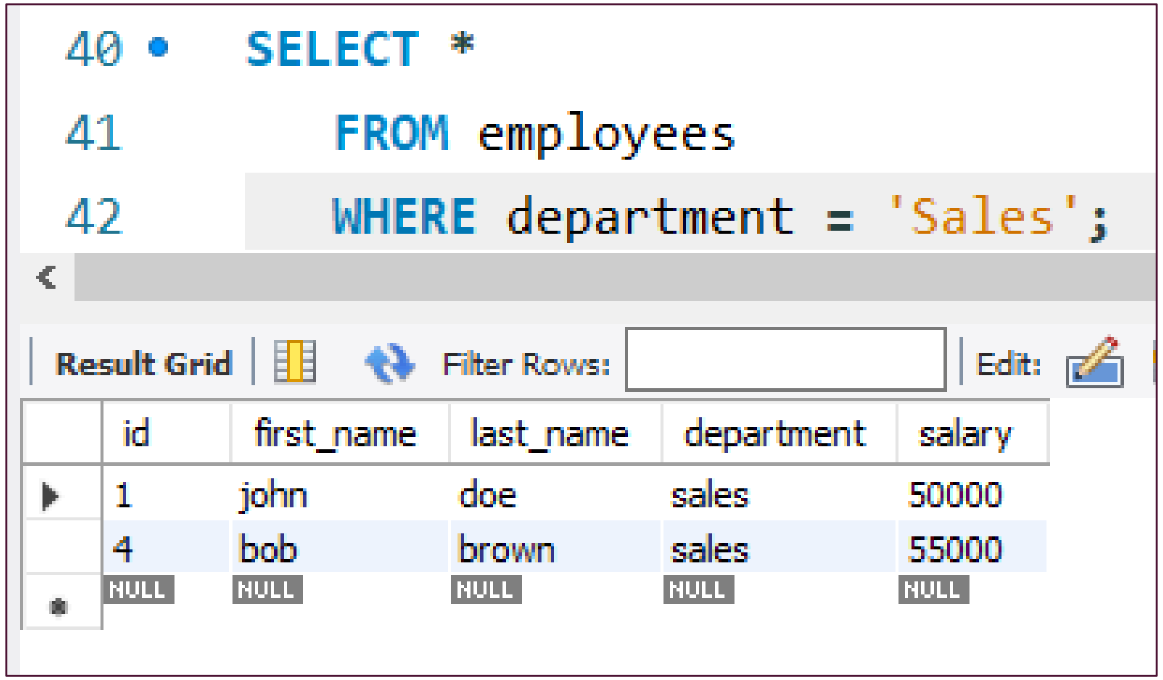
```
        UPDATE sales_data
SET total_sales = total_sales % 20;
```

qty_sold	unit_price	total_sales
10	5.5	7.5
20	3.75	17.5
15	6.2	6.5

COMPARISON OPERATOR – EQUAL TO(=)



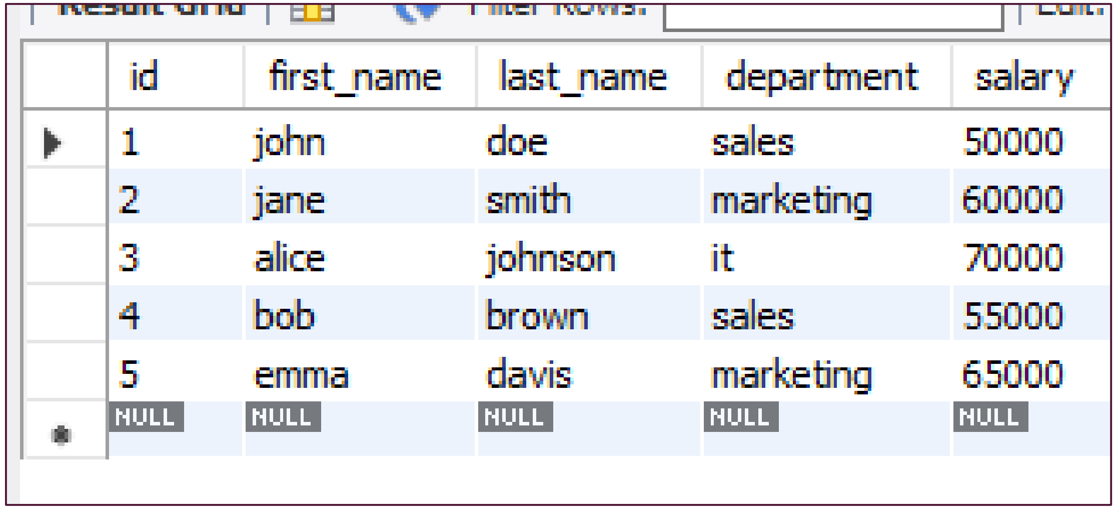
	id	first_name	last_name	department	salary
▶	1	john	doe	sales	50000
	2	jane	smith	marketing	60000
	3	alice	johnson	it	70000
	4	bob	brown	sales	55000
	5	emma	davis	marketing	65000
●	NULL	NULL	NULL	NULL	NULL



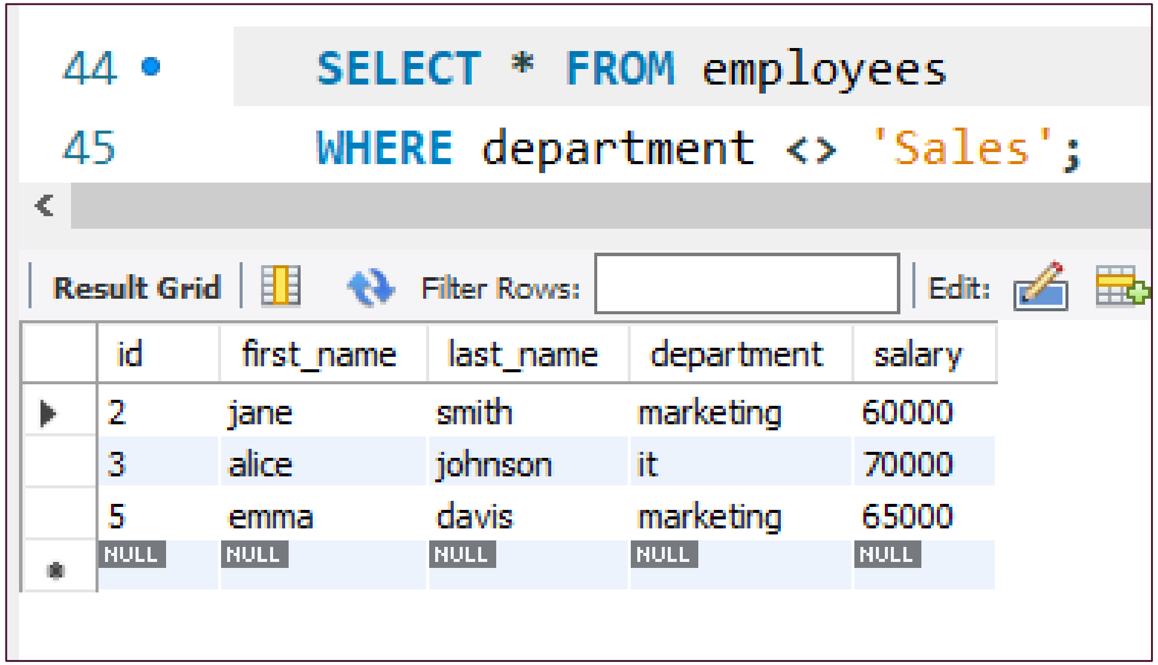
```
SELECT *
    FROM employees
    WHERE department = 'Sales';
```

	id	first_name	last_name	department	salary
▶	1	john	doe	sales	50000
	4	bob	brown	sales	55000
●	NULL	NULL	NULL	NULL	NULL

COMPARISON OPERATOR – NOT EQUAL TO(<> OR !=)



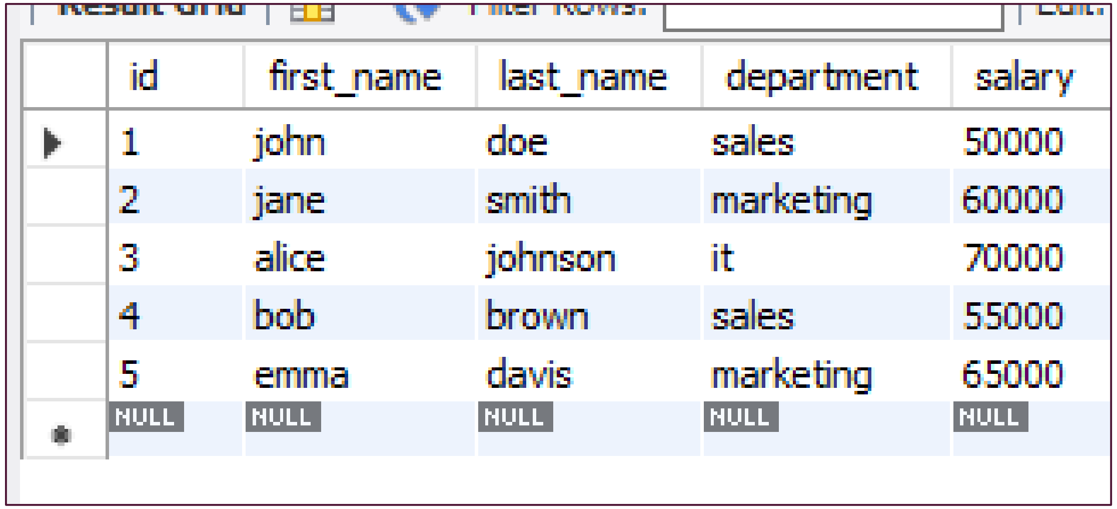
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



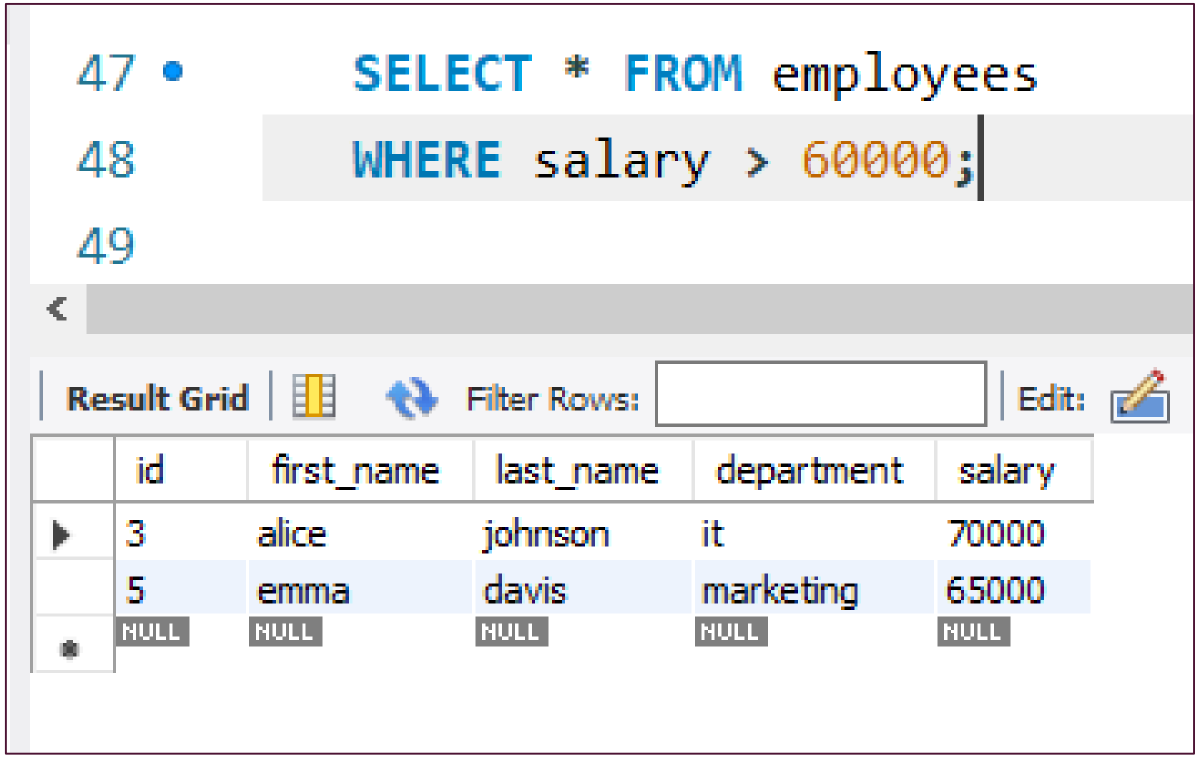
```
SELECT * FROM employees
WHERE department <> 'Sales';
```

id	first_name	last_name	department	salary
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

COMPARISON OPERATOR – GREATER THAN(>)



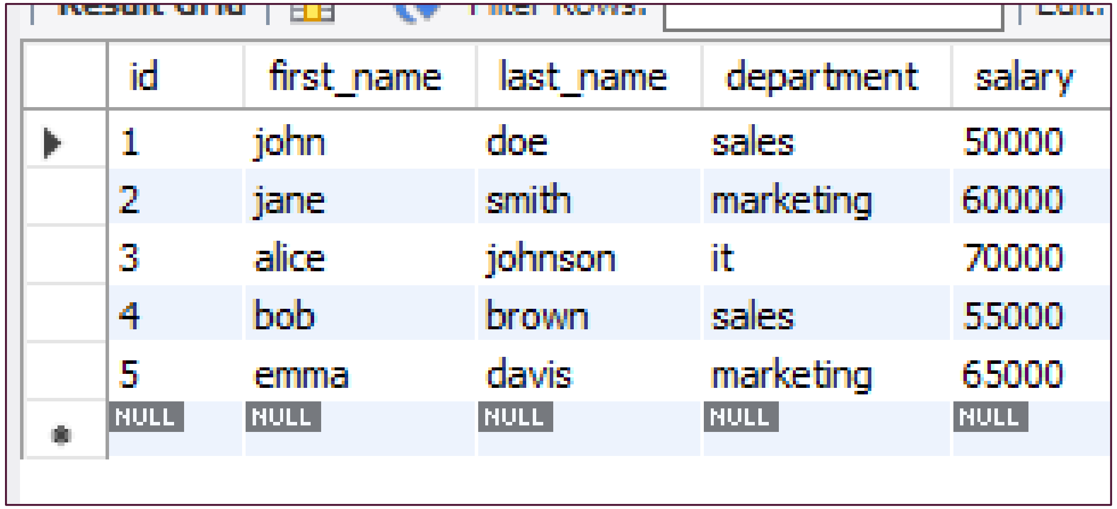
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



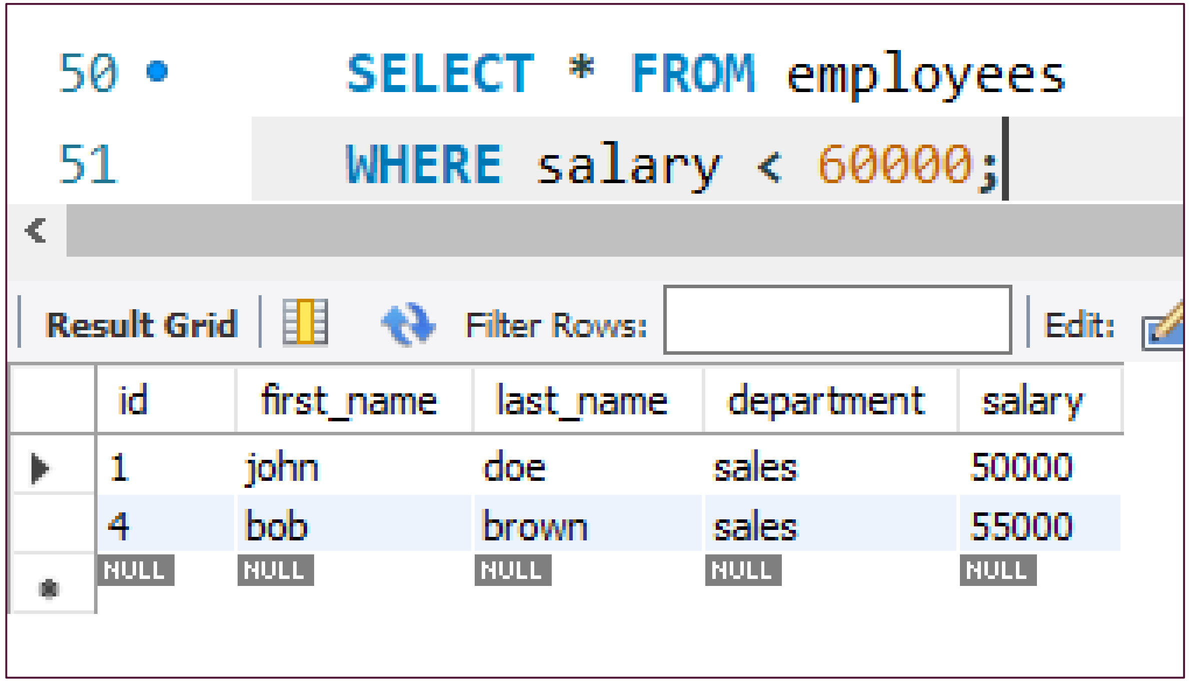
```sql
SELECT * FROM employees
WHERE salary > 60000;
```

id	first_name	last_name	department	salary
3	alice	johnson	it	70000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

COMPARISON OPERATOR – LESS THAN(<)



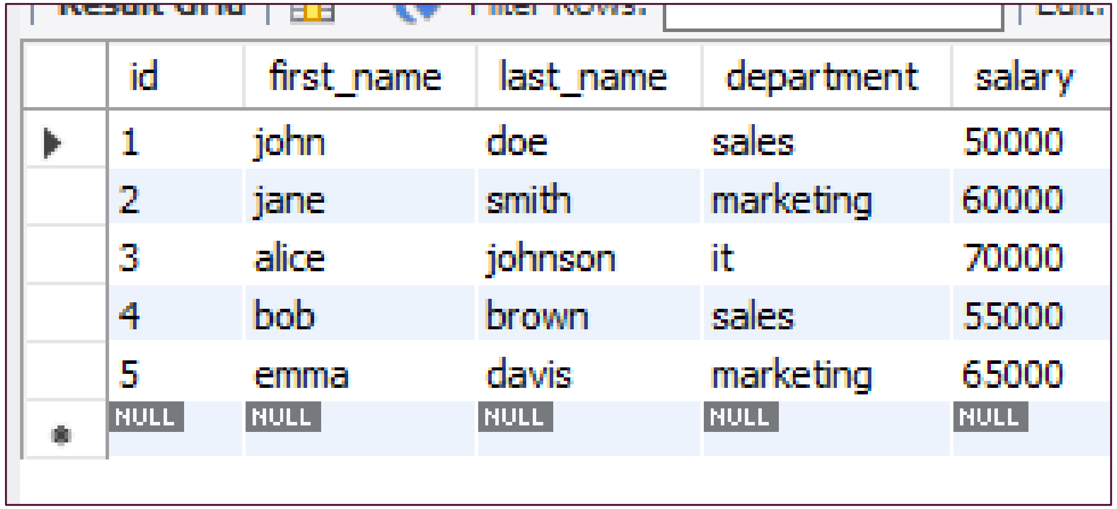
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



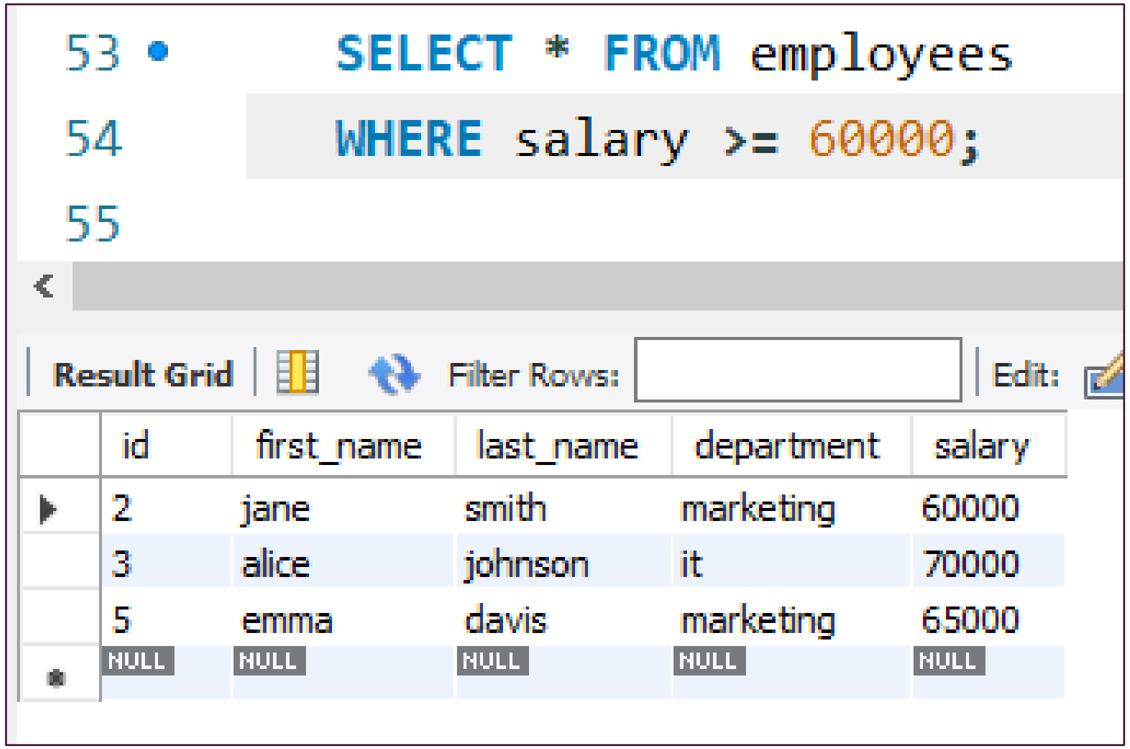
```
SELECT * FROM employees
WHERE salary < 60000;
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
4	bob	brown	sales	55000
NULL	NULL	NULL	NULL	NULL

COMPARISON OPERATOR – GREATER THAN OR EQUAL TO (>=)



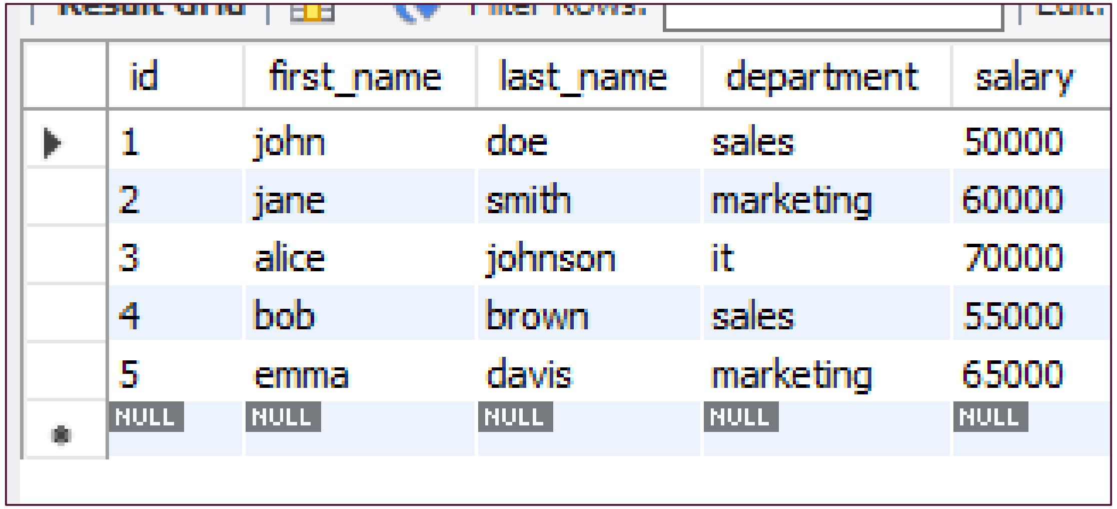
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



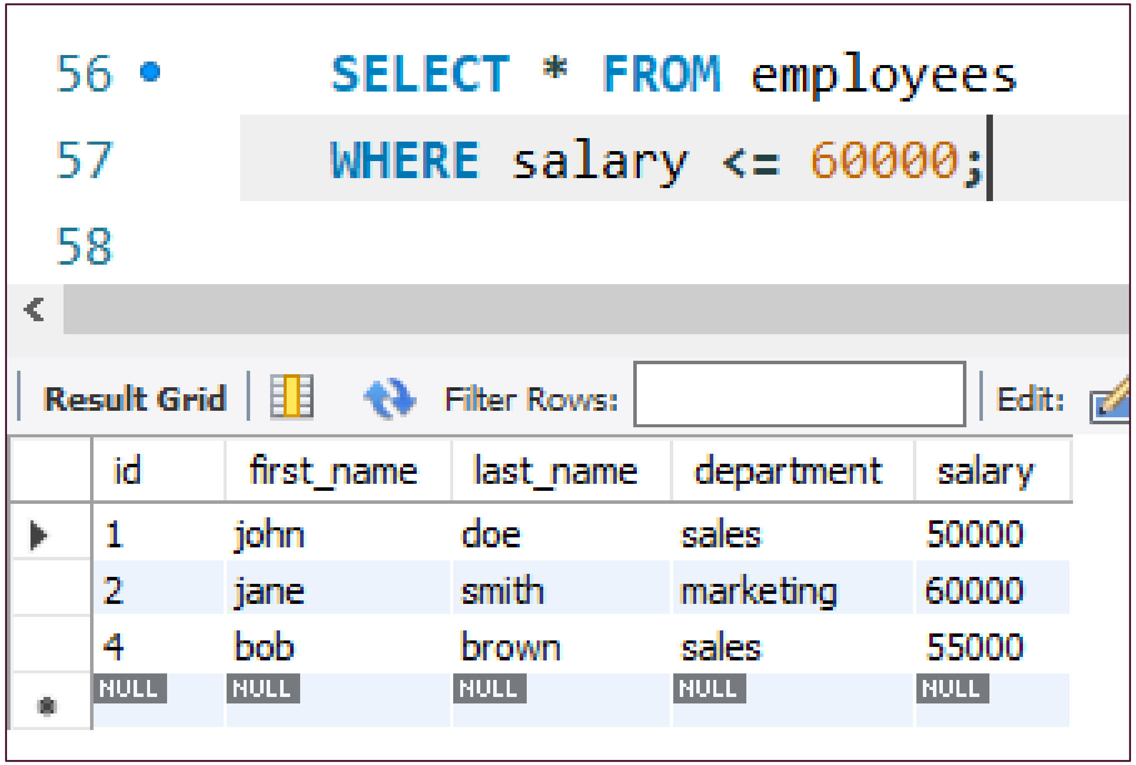
```
SELECT * FROM employees
WHERE salary >= 60000;
```

id	first_name	last_name	department	salary
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

COMPARISON OPERATOR – LESS THAN OR EQUAL TO (<=)



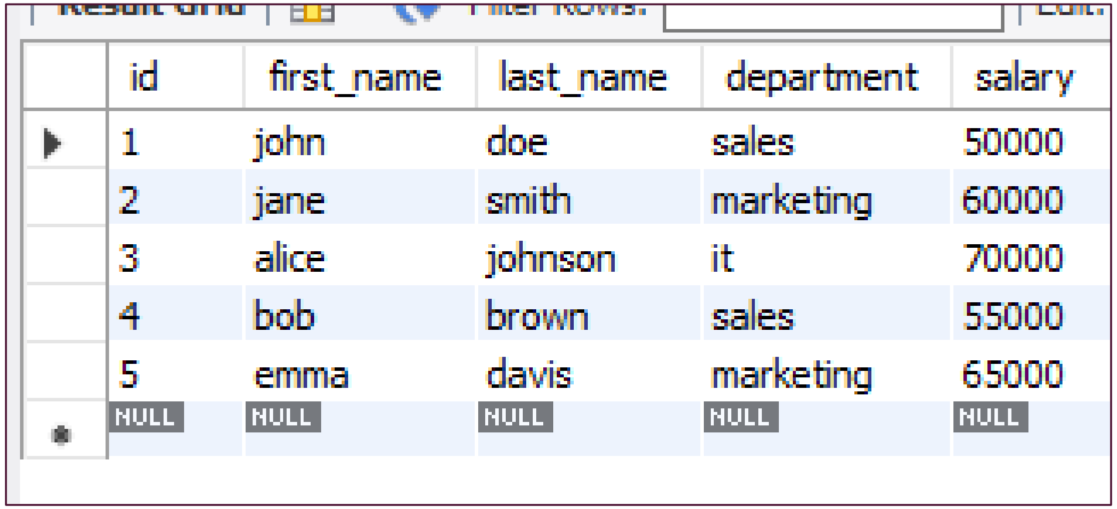
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



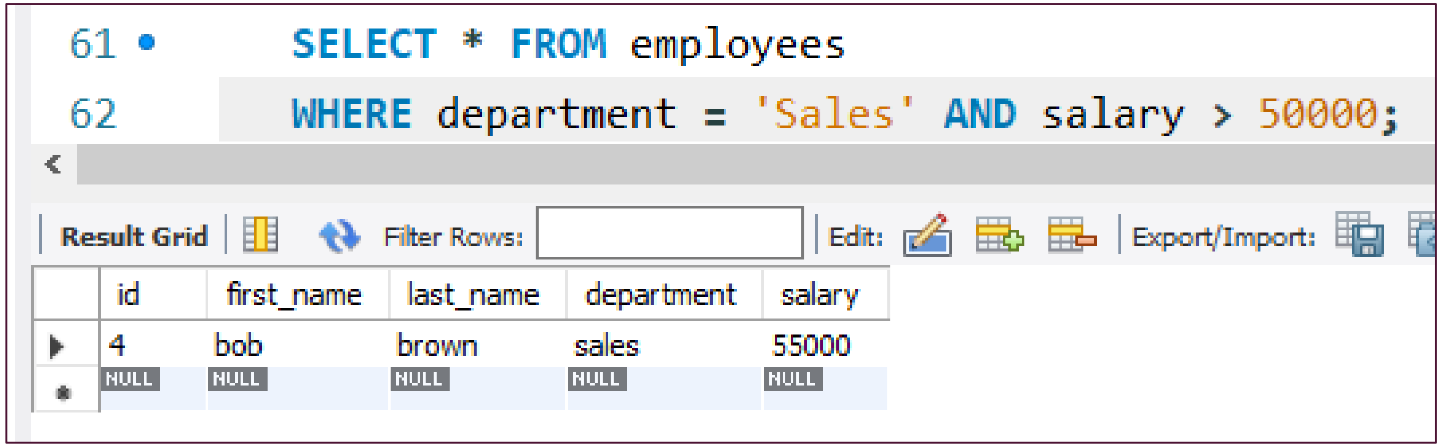
```
SELECT * FROM employees
WHERE salary <= 60000;
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
4	bob	brown	sales	55000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – AND



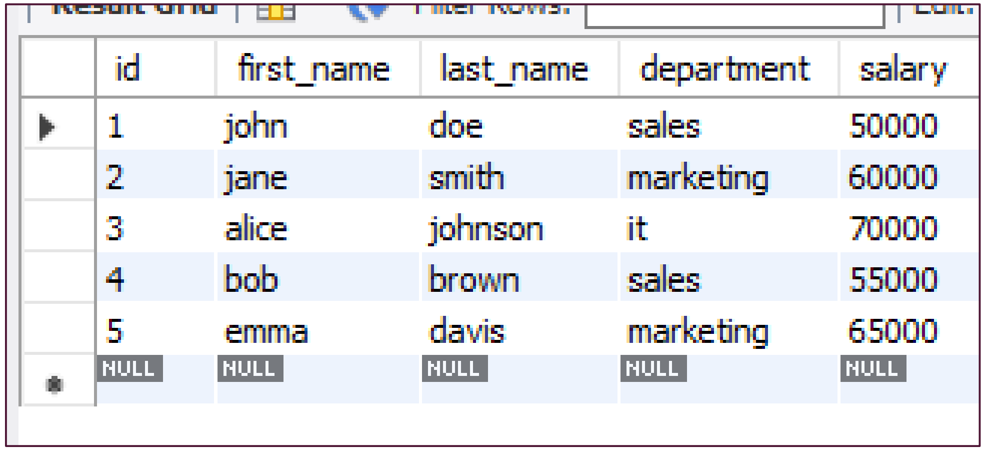
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



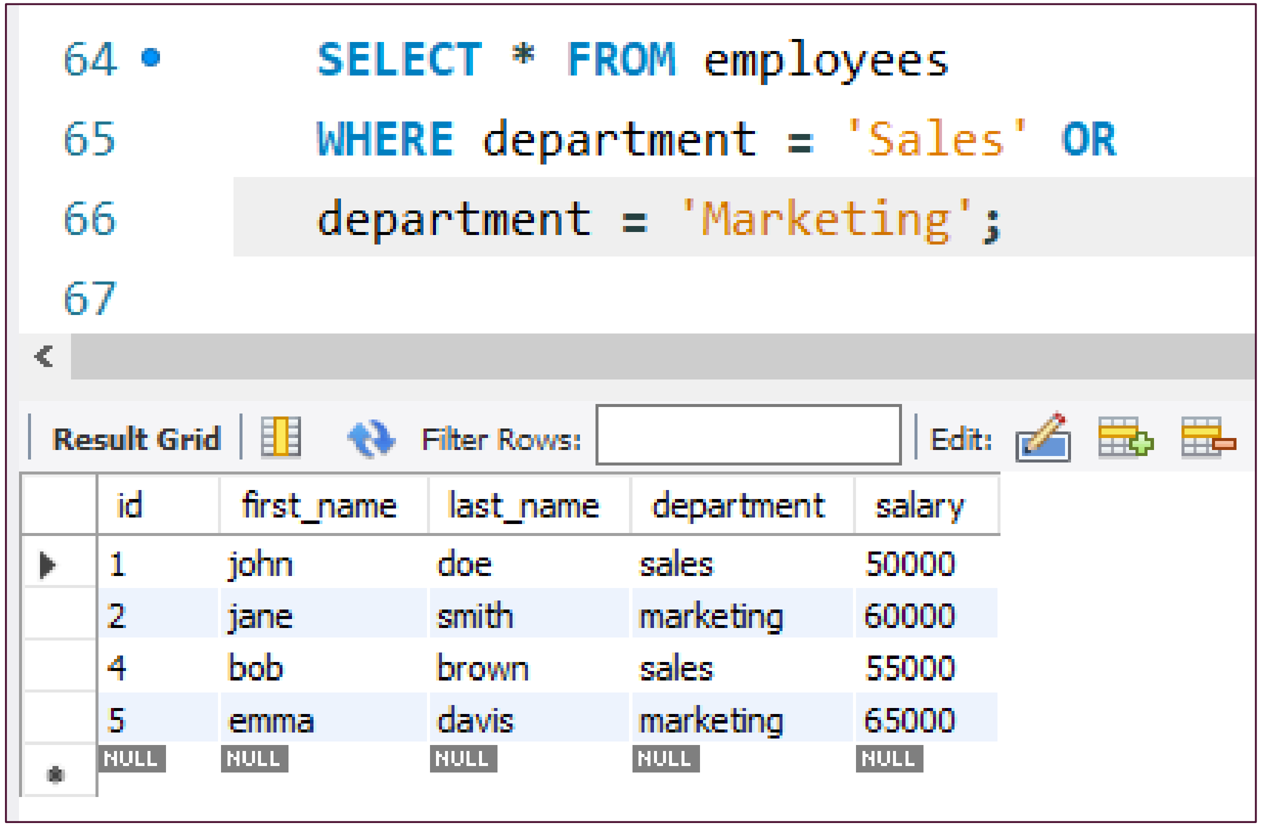
```sql
SELECT * FROM employees
WHERE department = 'Sales' AND salary > 50000;
```

id	first_name	last_name	department	salary
4	bob	brown	sales	55000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – OR



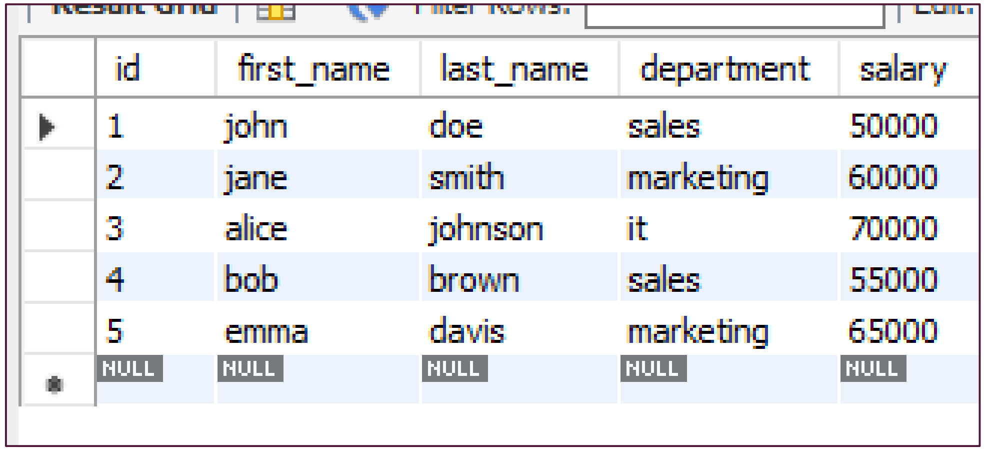
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



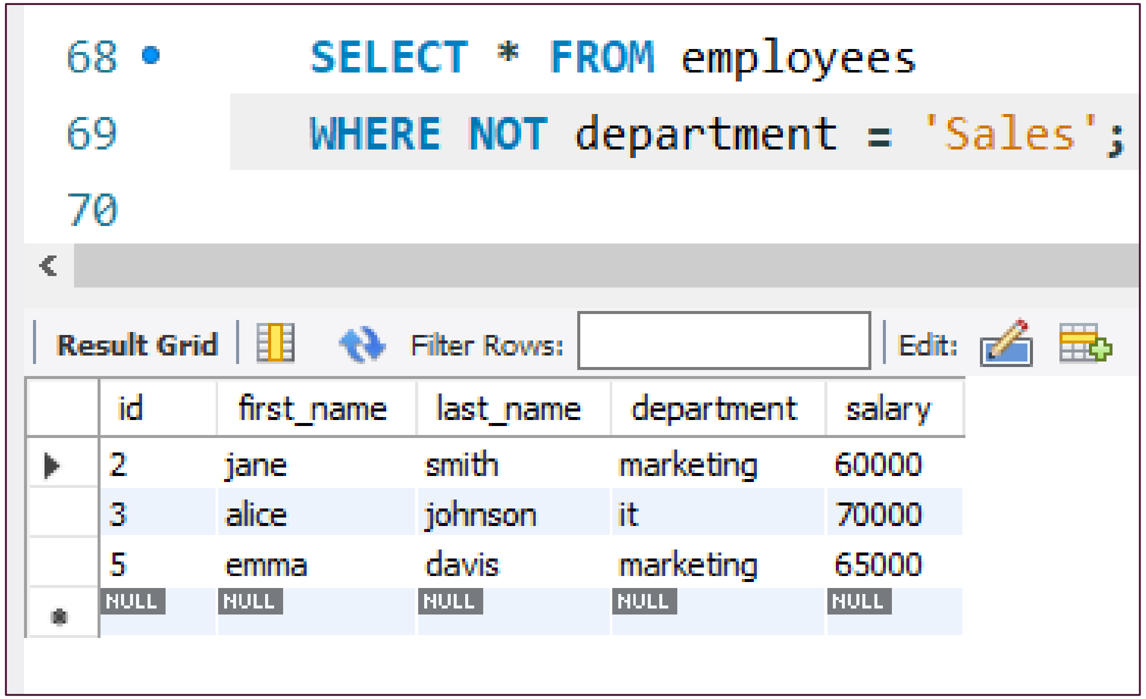
```sql
SELECT * FROM employees
WHERE department = 'Sales' OR
department = 'Marketing';
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – NOT



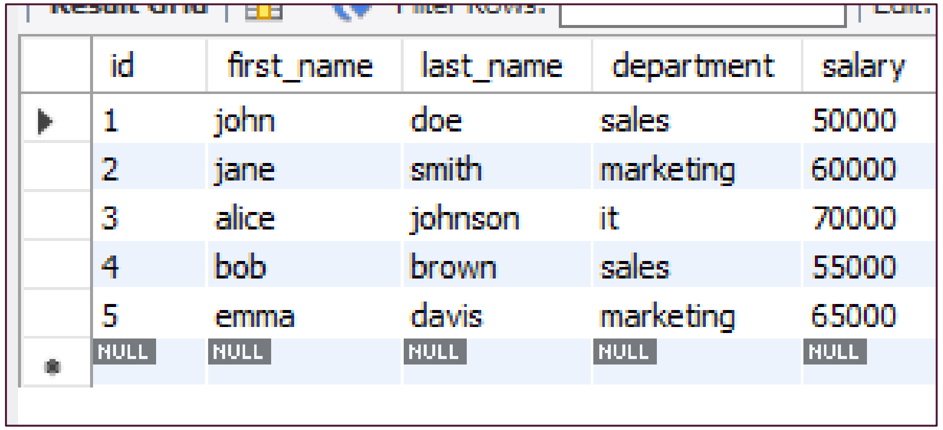
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



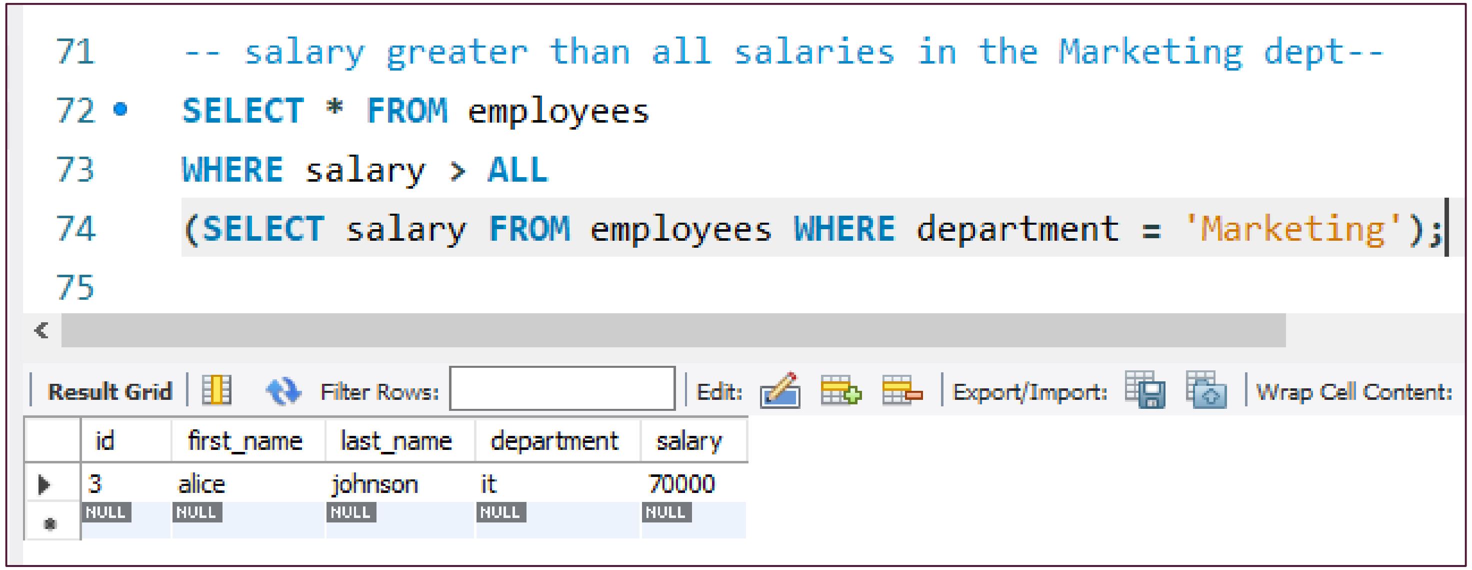
```sql
SELECT * FROM employees
WHERE NOT department = 'Sales';
```

id	first_name	last_name	department	salary
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – ALL



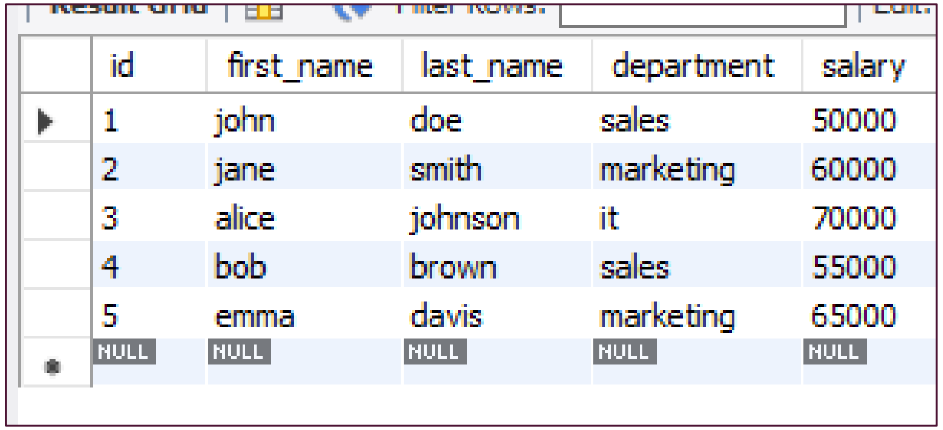
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



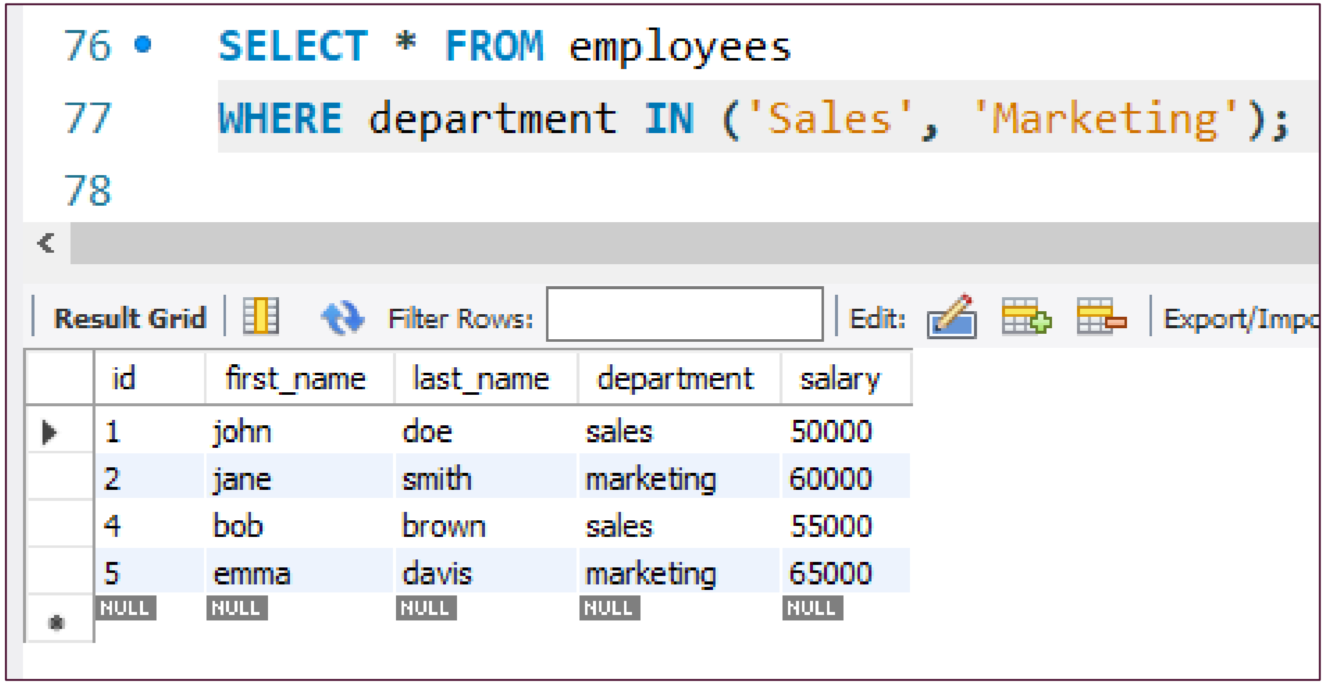
```
-- salary greater than all salaries in the Marketing dept--
SELECT * FROM employees
WHERE salary > ALL
(SELECT salary FROM employees WHERE department = 'Marketing');
```

id	first_name	last_name	department	salary
3	alice	johnson	it	70000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – IN



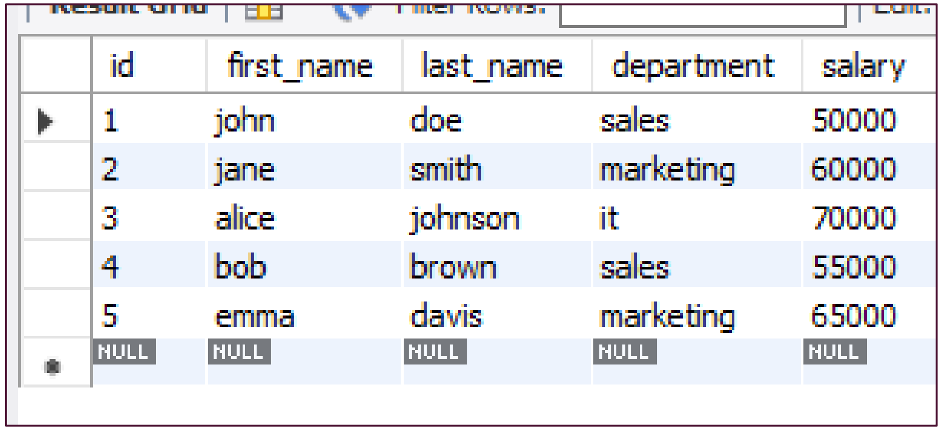
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



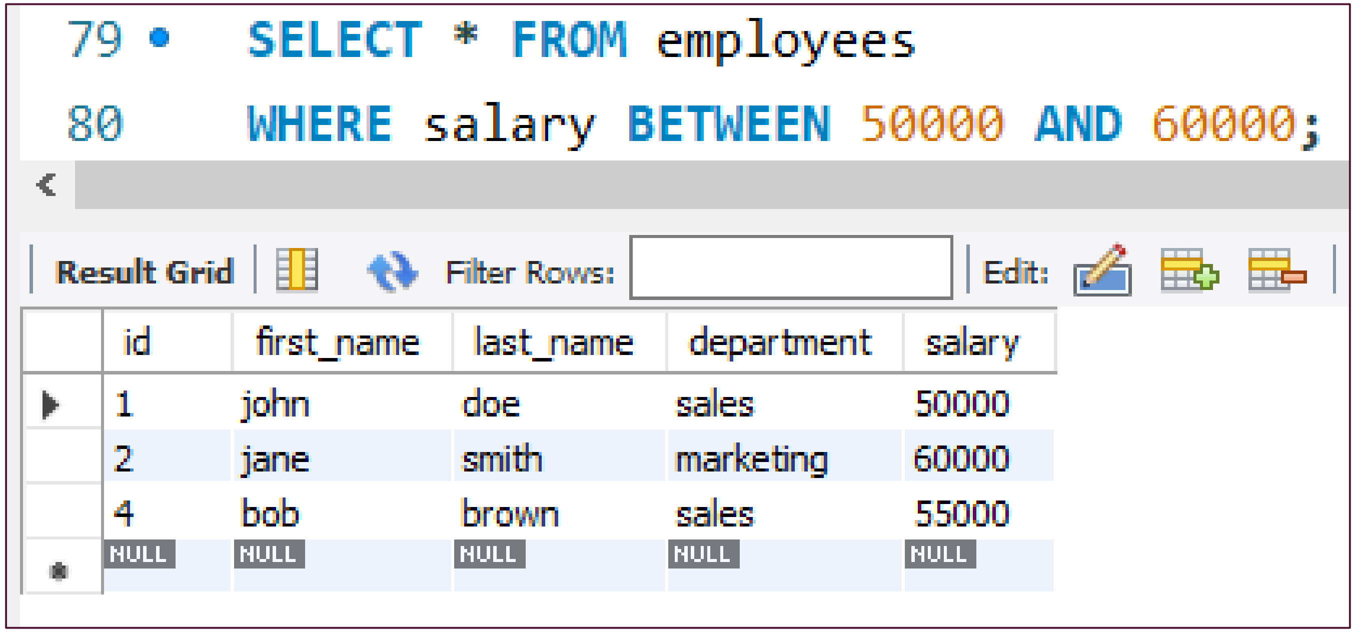
```
SELECT * FROM employees
WHERE department IN ('Sales', 'Marketing');
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – BETWEEN



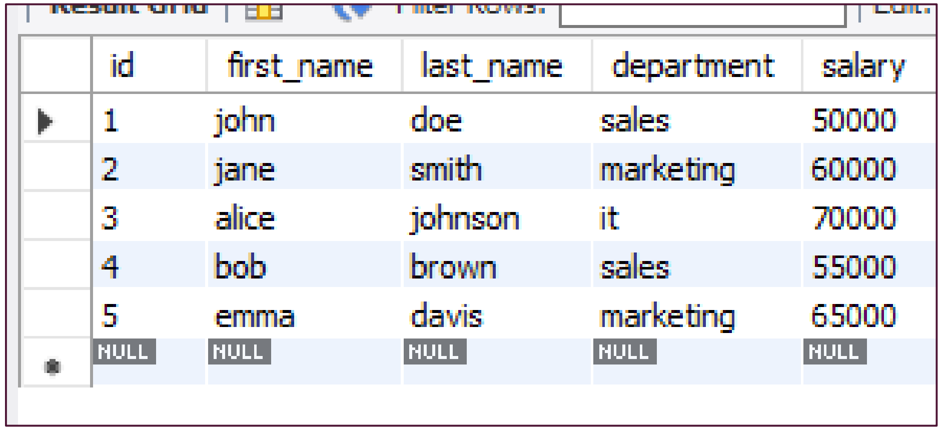
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



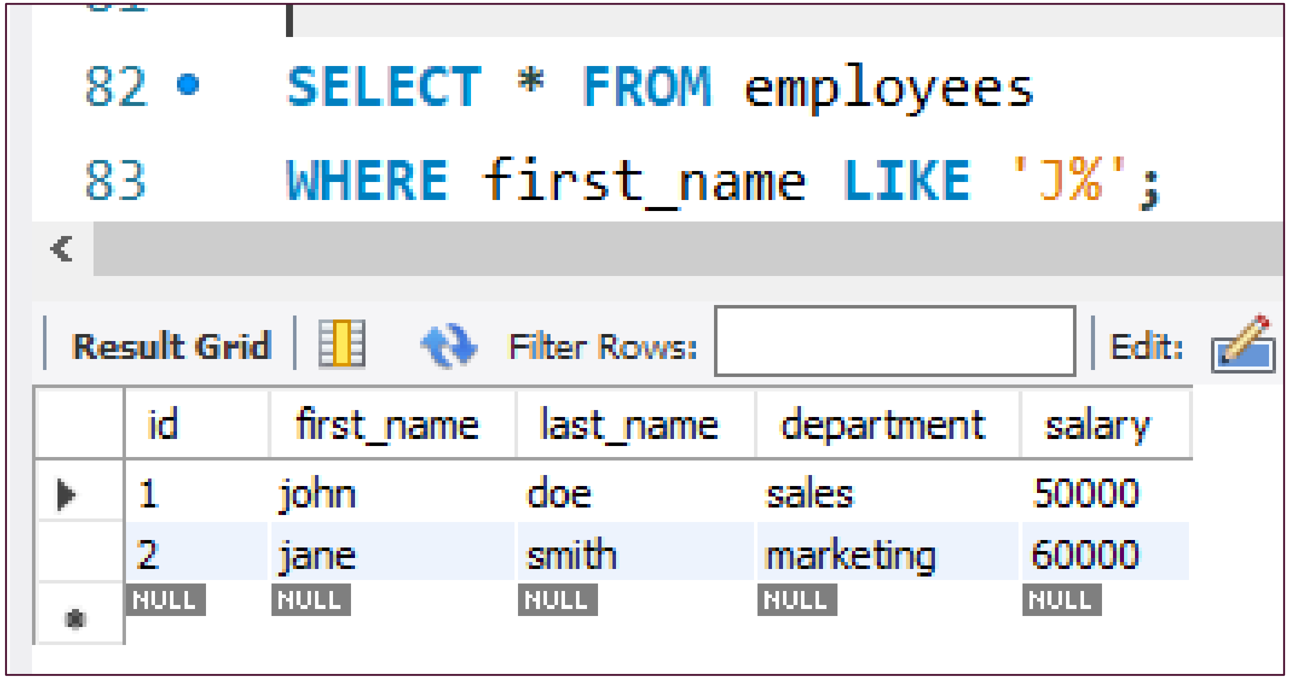
```sql
SELECT * FROM employees
WHERE salary BETWEEN 50000 AND 60000;
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
4	bob	brown	sales	55000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – LIKE



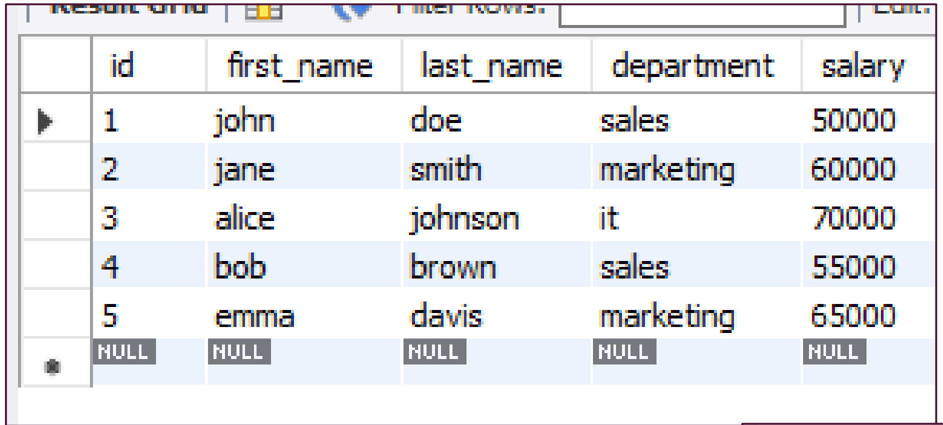
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



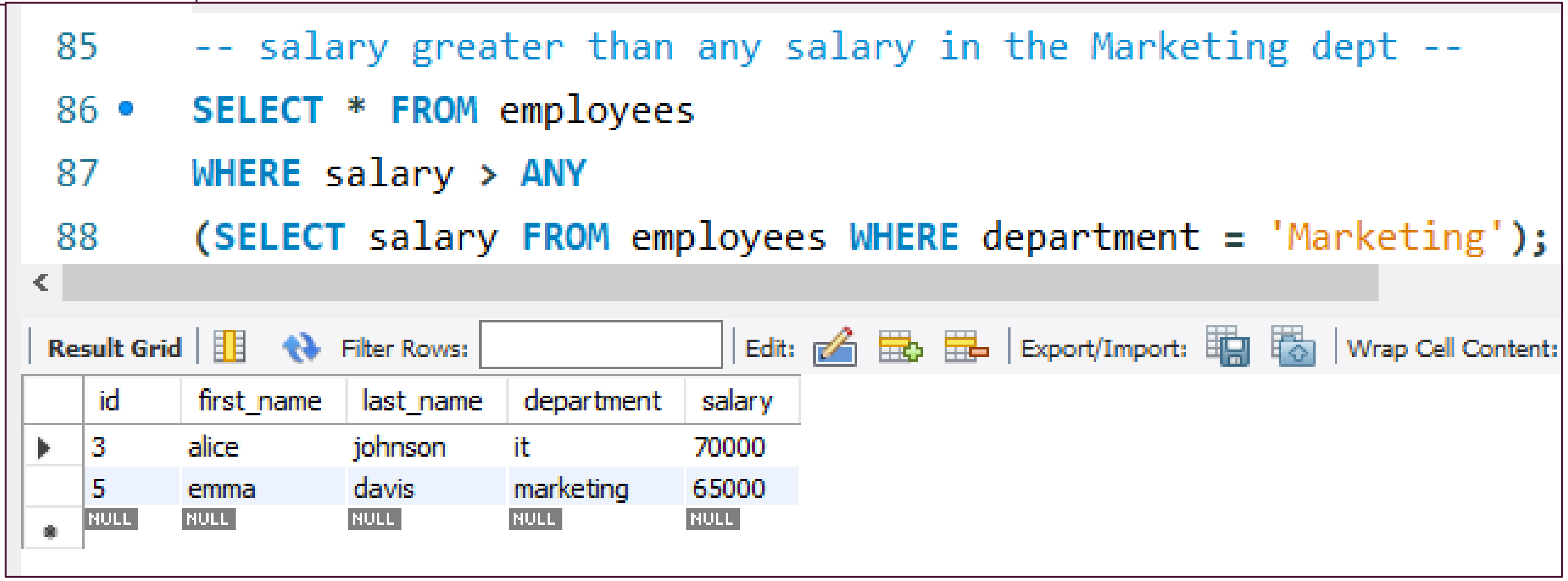
```
SELECT * FROM employees
WHERE first_name LIKE 'J%';
```

id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
NULL	NULL	NULL	NULL	NULL

LOGICAL OPERATOR – ANY



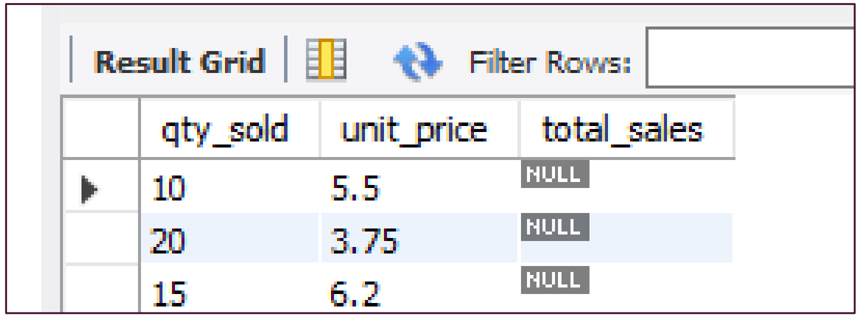
id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL



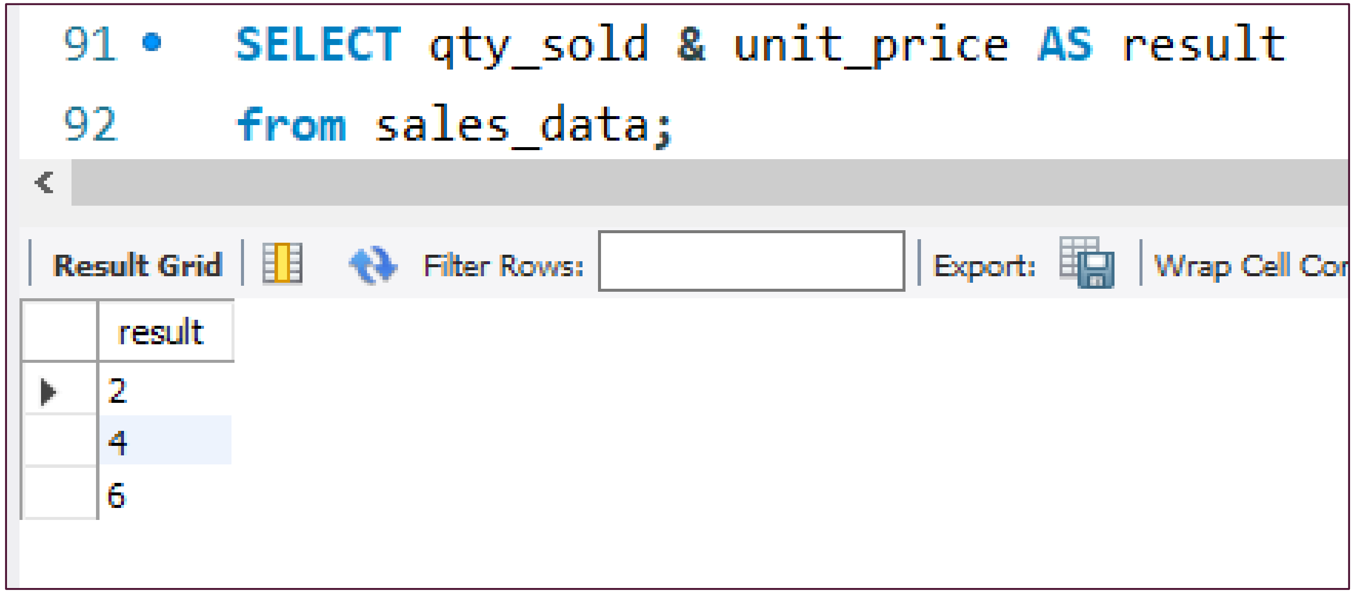
```
-- salary greater than any salary in the Marketing dept --
SELECT * FROM employees
WHERE salary > ANY
(SELECT salary FROM employees WHERE department = 'Marketing');
```

id	first_name	last_name	department	salary
3	alice	johnson	it	70000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

BITWISE OPERATOR – AND(&)



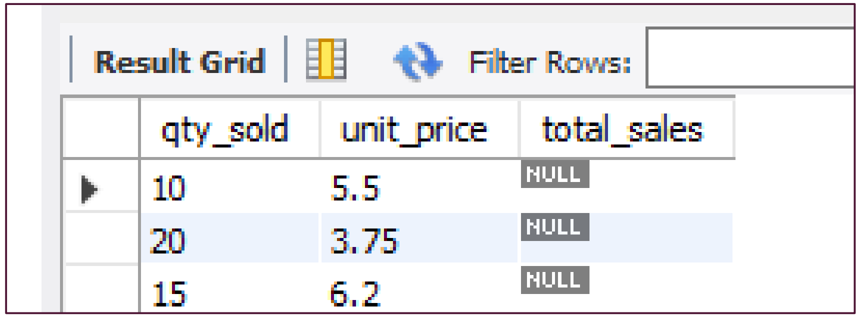
	qty_sold	unit_price	total_sales
▶	10	5.5	NULL
	20	3.75	NULL
	15	6.2	NULL



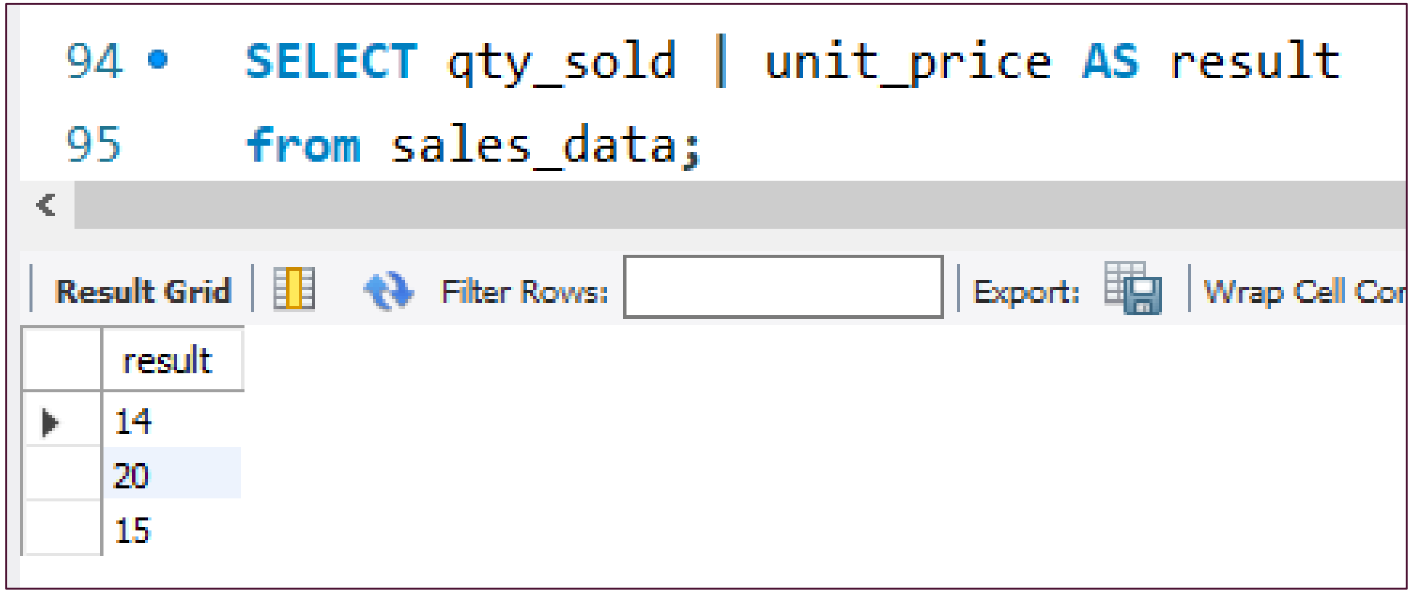
```sql
SELECT qty_sold & unit_price AS result
from sales_data;
```

	result
▶	2
	4
	6

BITWISE OPERATOR – OR(|)



	qty_sold	unit_price	total_sales
▶	10	5.5	NULL
	20	3.75	NULL
	15	6.2	NULL



```
SELECT qty_sold | unit_price AS result
from sales_data;
```

	result
▶	14
	20
	15

AGGREGATE FUNCTIONS IN SQL

SQL Series Part 6

-Mayuri Dandekar

WHAT IS FUNCTIONS?

Functions in SQL are the database objects that contains a **set of SQL statements** to perform a specific task. A function **accepts input parameters, perform actions, and then return the result**.

Types of Function:

1. **System Defined Function** : these are built-in functions

   Example: rand(), round(), upper(), lower(), count(), sum(), avg(), max(), etc

2. **User-Defined Function** : Once you define a function, you can call it in the same way as the built-in functions

AGGREGATE FUNCTIONS

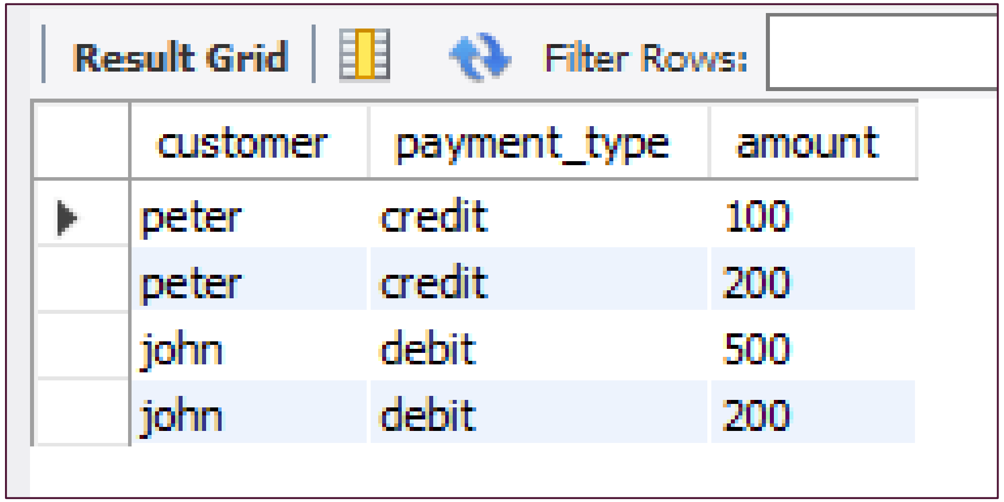
Aggregate function **performs a calculation** on multiple values and returns a single value.

Aggregate functions are often **used with GROUP BY & SELECT statement**

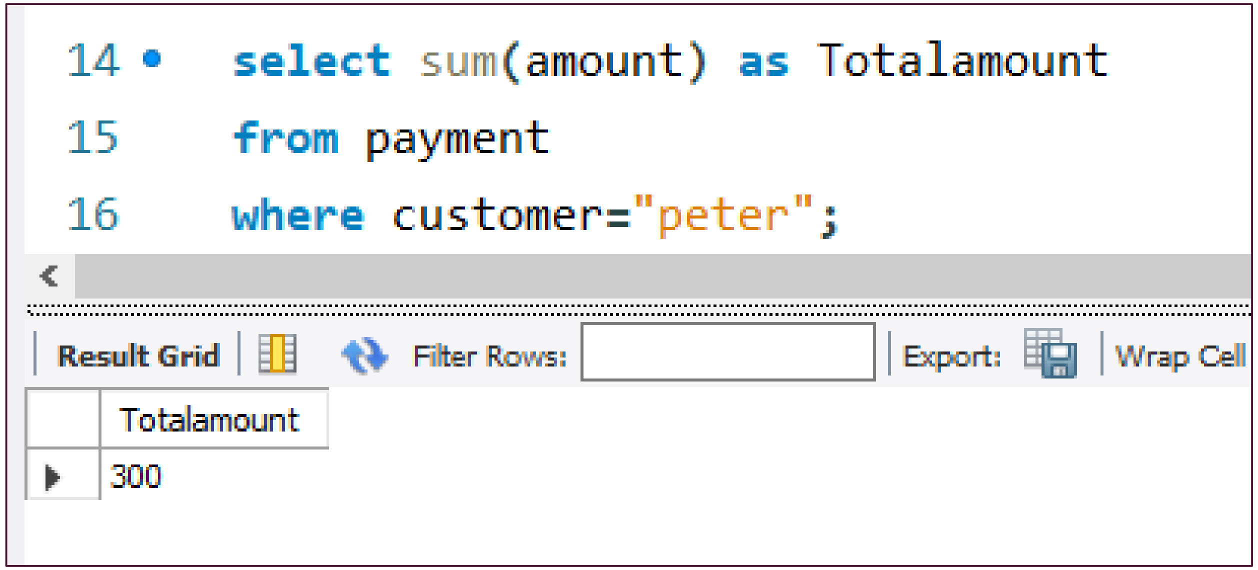
- COUNT() returns number of values
- SUM() returns sum of all values
- AVG() returns average value
- MAX() returns maximum value
- MIN() returns minimum value
- ROUND() Rounds a number to a specified number of decimal places

AGGREGATE FUNCTION -- SUM

Sum function sum the value of all the rows in the group. If the group by clause is omitted then it sums all the rows.



customer	payment_type	amount
peter	credit	100
peter	credit	200
john	debit	500
john	debit	200

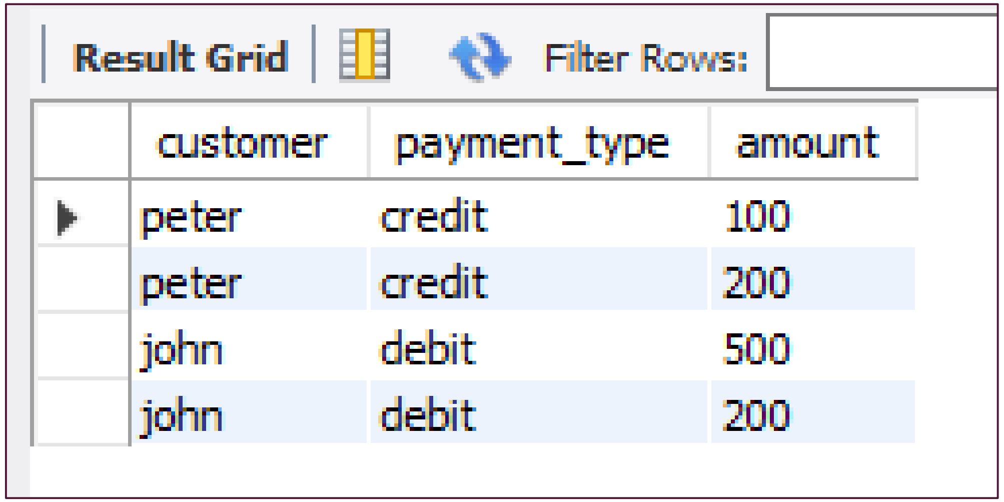


```
select sum(amount) as Totalamount
from payment
where customer="peter";
```

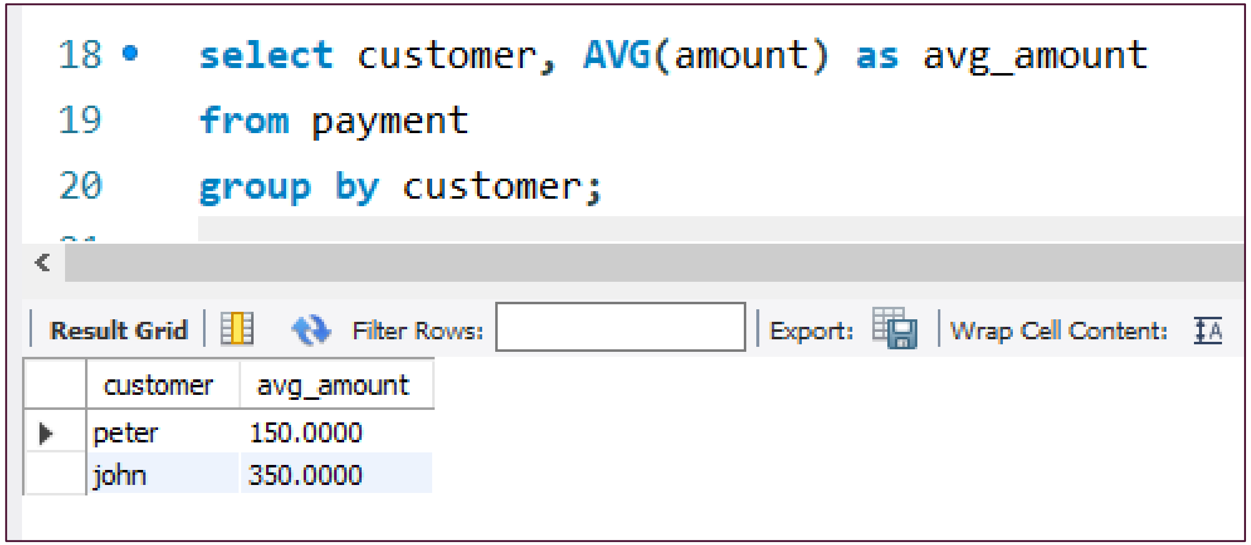
Totalamount
300

AGGREGATE FUNCTION -- AVG

The aggregate function AVG() returns the **average of a given expression**, usually numeric values in a column.



customer	payment_type	amount
peter	credit	100
peter	credit	200
john	debit	500
john	debit	200

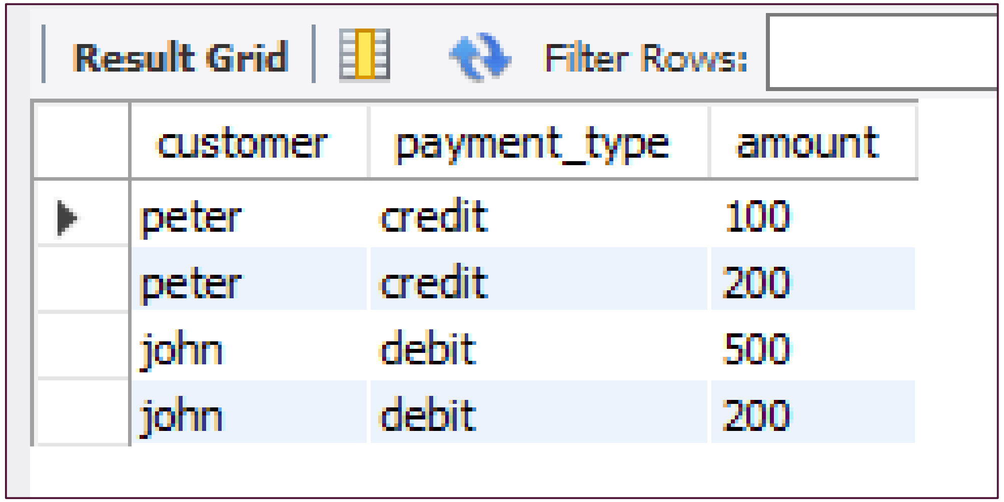


```
select customer, AVG(amount) as avg_amount
from payment
group by customer;
```

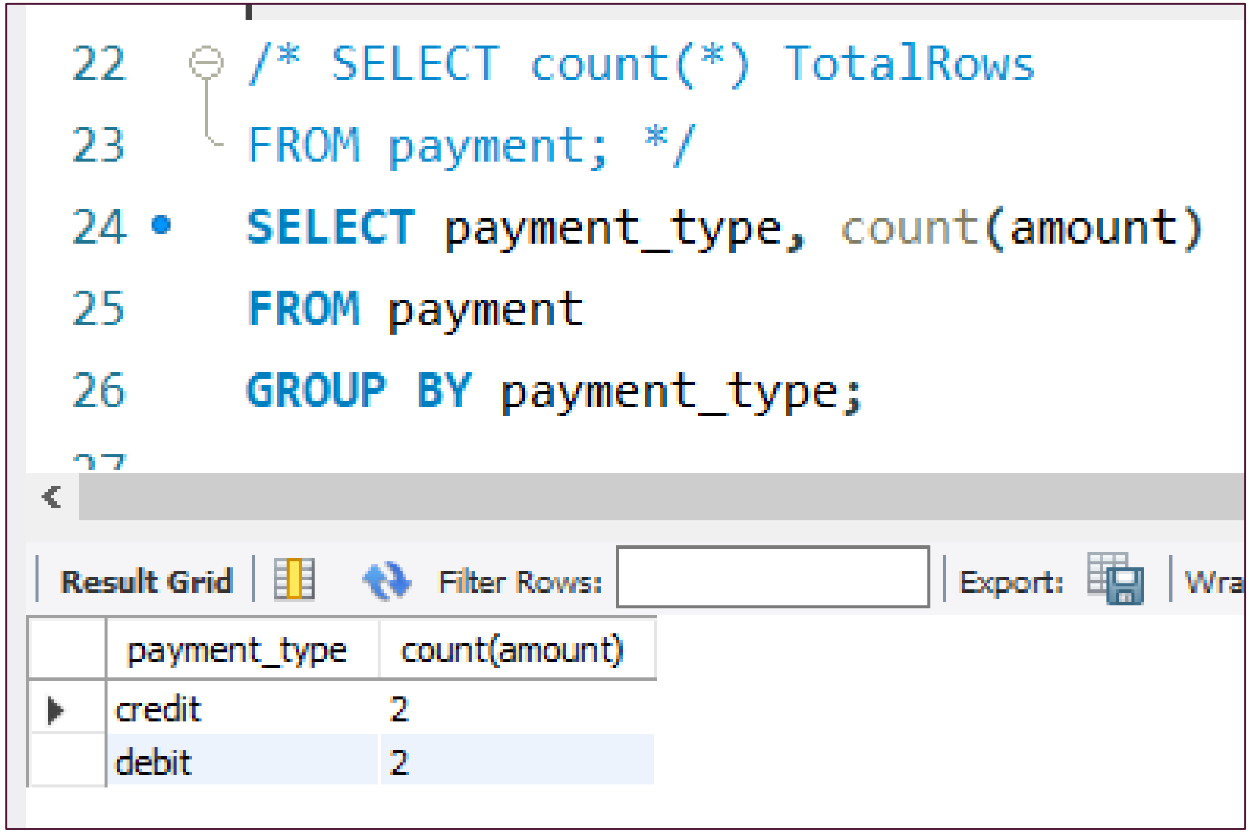
customer	avg_amount
peter	150.0000
john	350.0000

AGGREGATE FUNCTION -- COUNT

You can count the number of rows or count as per the given expression



	customer	payment_type	amount
▶	peter	credit	100
	peter	credit	200
	john	debit	500
	john	debit	200

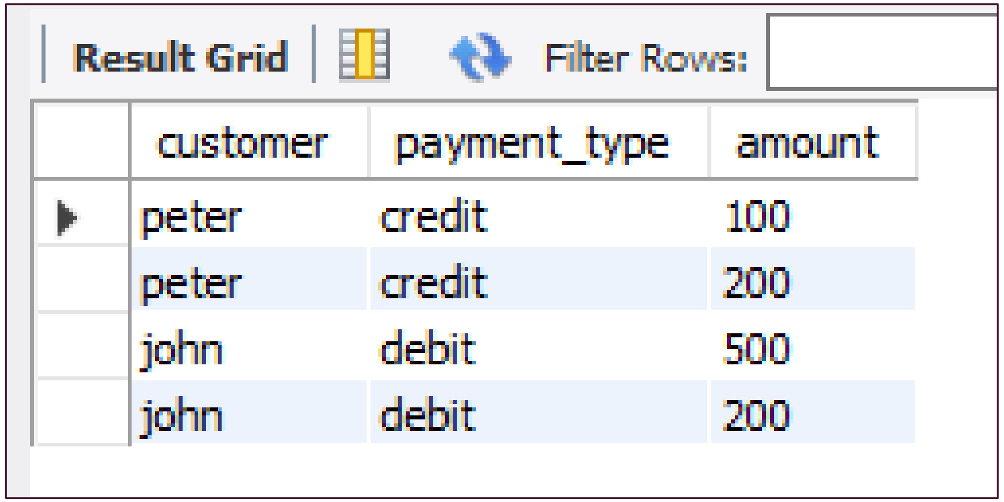


```
/* SELECT count(*) TotalRows
FROM payment; */
SELECT payment_type, count(amount)
FROM payment
GROUP BY payment_type;
```

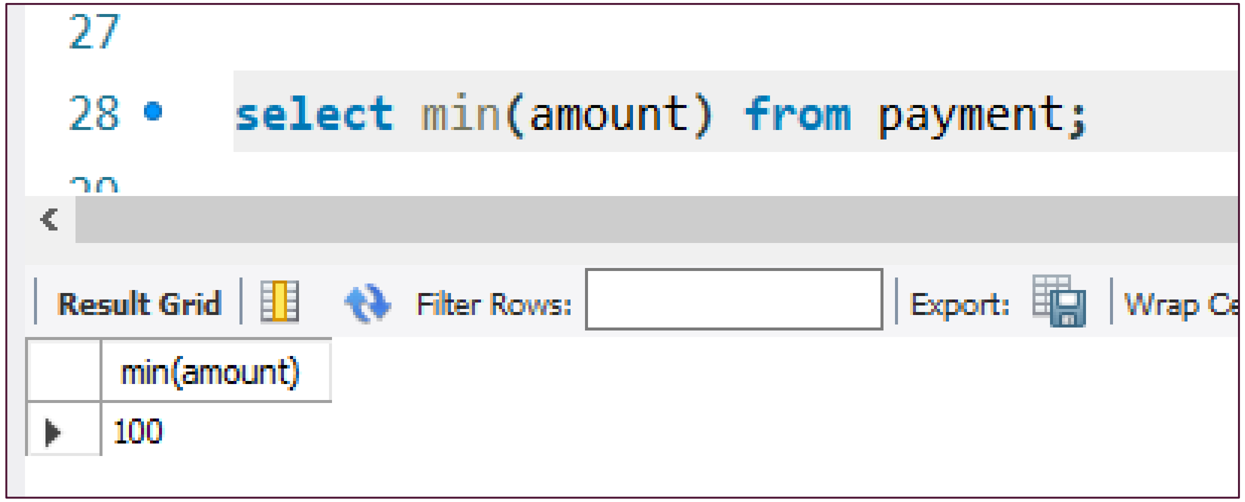
	payment_type	count(amount)
▶	credit	2
	debit	2

AGGREGATE FUNCTION -- MIN

Find the **smallest value** of column



	customer	payment_type	amount
▶	peter	credit	100
	peter	credit	200
	john	debit	500
	john	debit	200

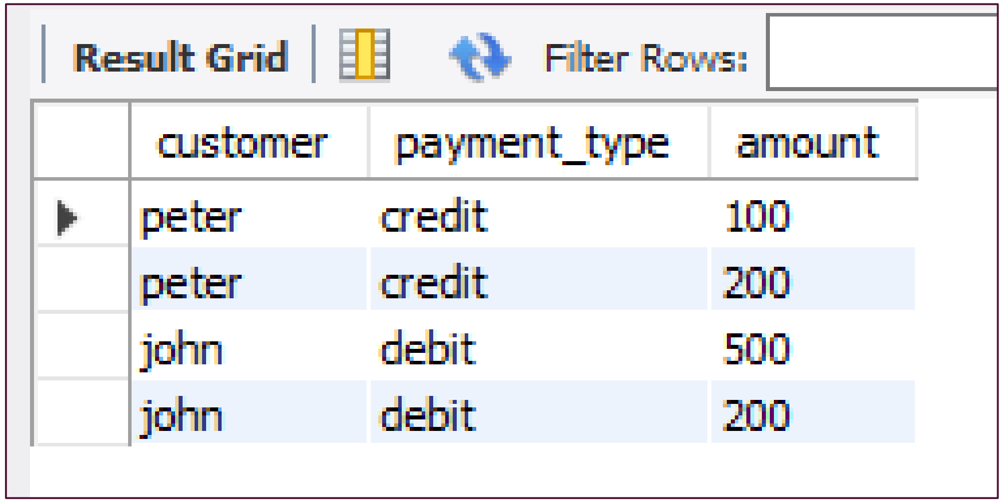


```
select min(amount) from payment;
```

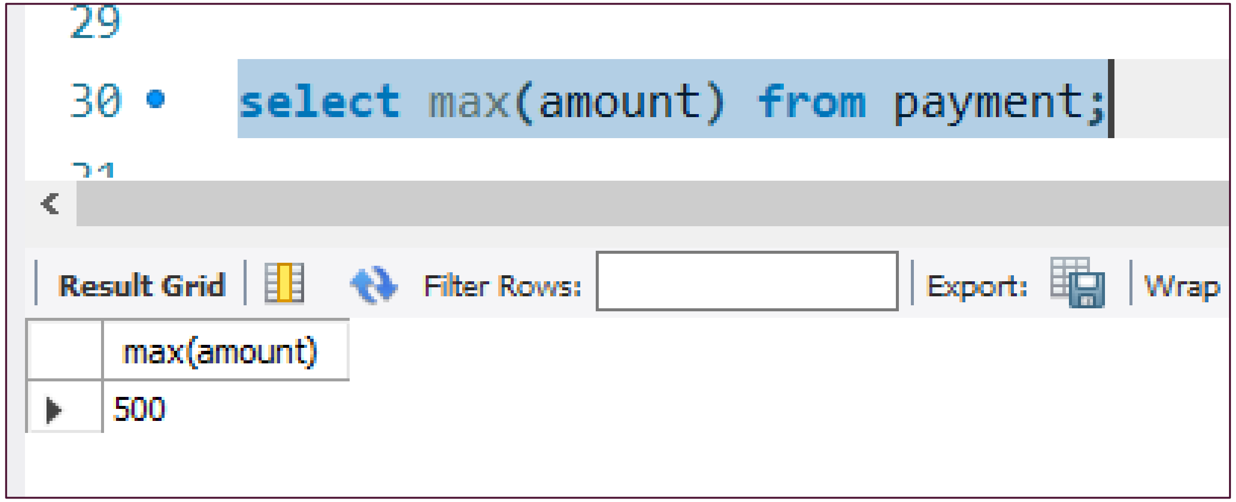
	min(amount)
▶	100

AGGREGATE FUNCTION -- MAX

Find the **largest value** of column



customer	payment_type	amount
peter	credit	100
peter	credit	200
john	debit	500
john	debit	200

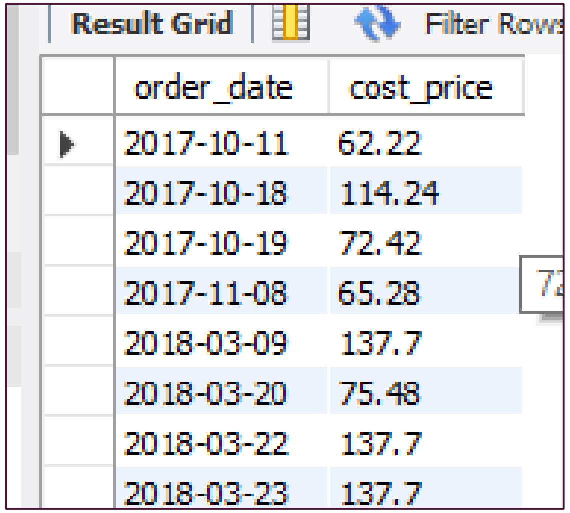


```sql
select max(amount) from payment;
```

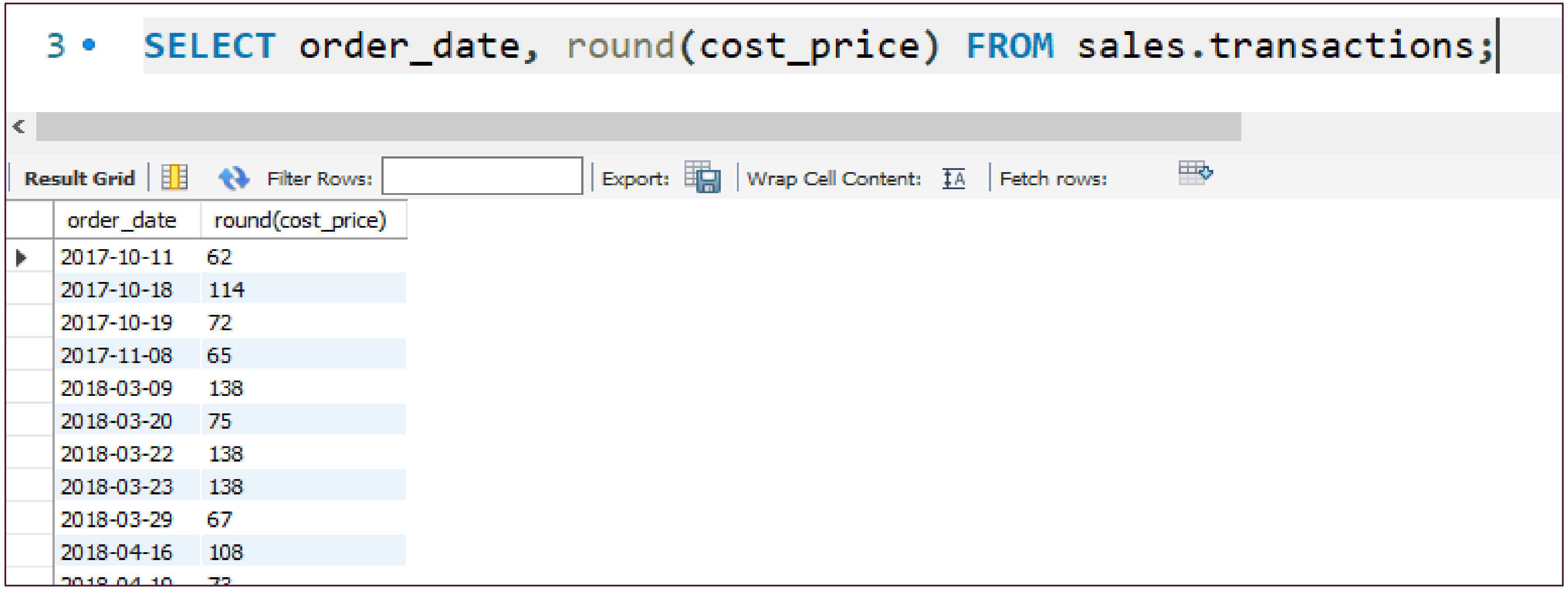
max(amount)
500

AGGREGATE FUNCTION -- ROUND

Rounds a number to specific number of a decimal place.



order_date	cost_price
2017-10-11	62.22
2017-10-18	114.24
2017-10-19	72.42
2017-11-08	65.28
2018-03-09	137.7
2018-03-20	75.48
2018-03-22	137.7
2018-03-23	137.7



```
SELECT order_date, round(cost_price) FROM sales.transactions;
```

order_date	round(cost_price)
2017-10-11	62
2017-10-18	114
2017-10-19	72
2017-11-08	65
2018-03-09	138
2018-03-20	75
2018-03-22	138
2018-03-23	138
2018-03-29	67
2018-04-16	108

STRING FUNCTIONS IN SQL

SQL Series Part 7

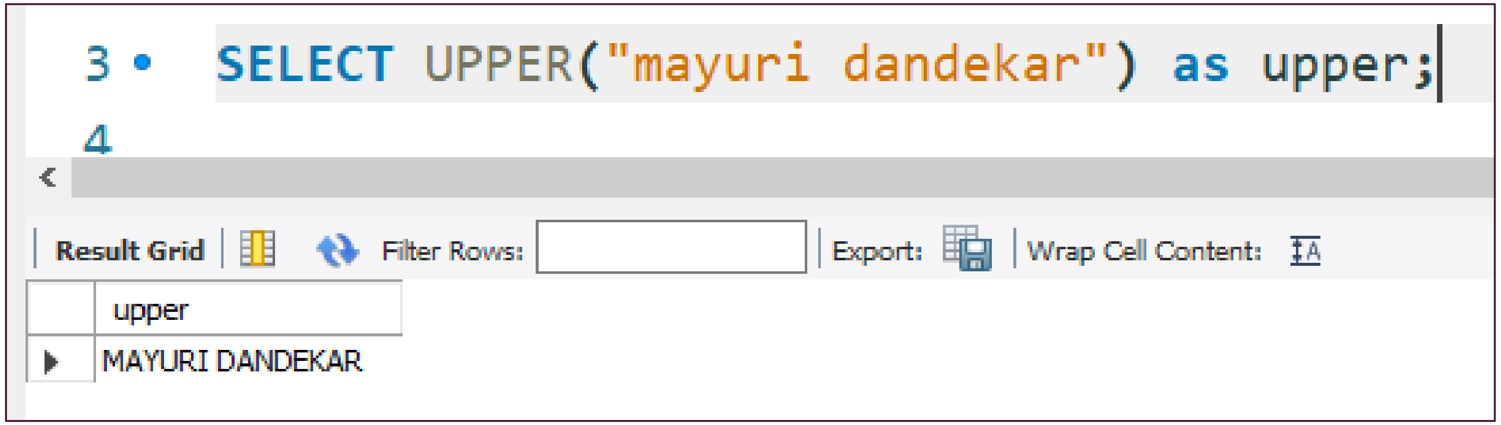
-Mayuri Dandekar

STRING FUNCTIONS

String functions are used to perform an **operation on input string** and return an output string.

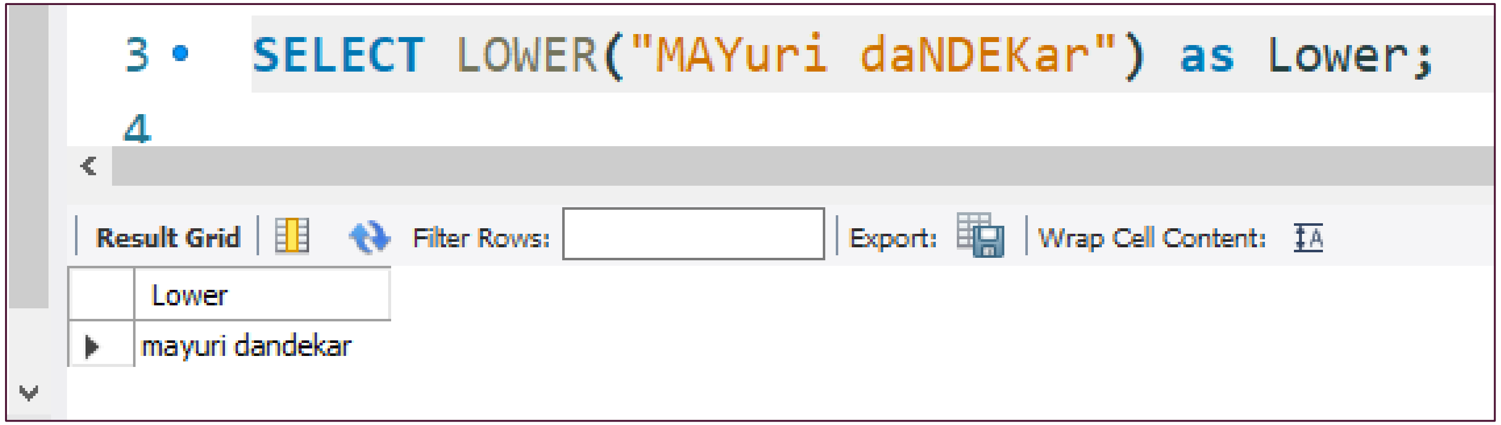
- **UPPER**() converts the value of a field to uppercase
- **LOWER**() converts the value of a field to lowercase
- **LENGTH**() returns the length of the value in a text field
- **SUBSTRING**() extracts a substring from a string
- **NOW**() returns the current system date and time
- **FORMAT**() used to set the format of a field
- **CONCAT**() adds two or more strings together
- **REPLACE**() Replaces all occurrences of a substring within a string, with a new substring
- **TRIM**() removes leading and trailing spaces (or other specified characters) from a string

STRING FUNCTIONS – UPPER() AND LOWER()



```
SELECT UPPER("mayuri dandekar") as upper;
```

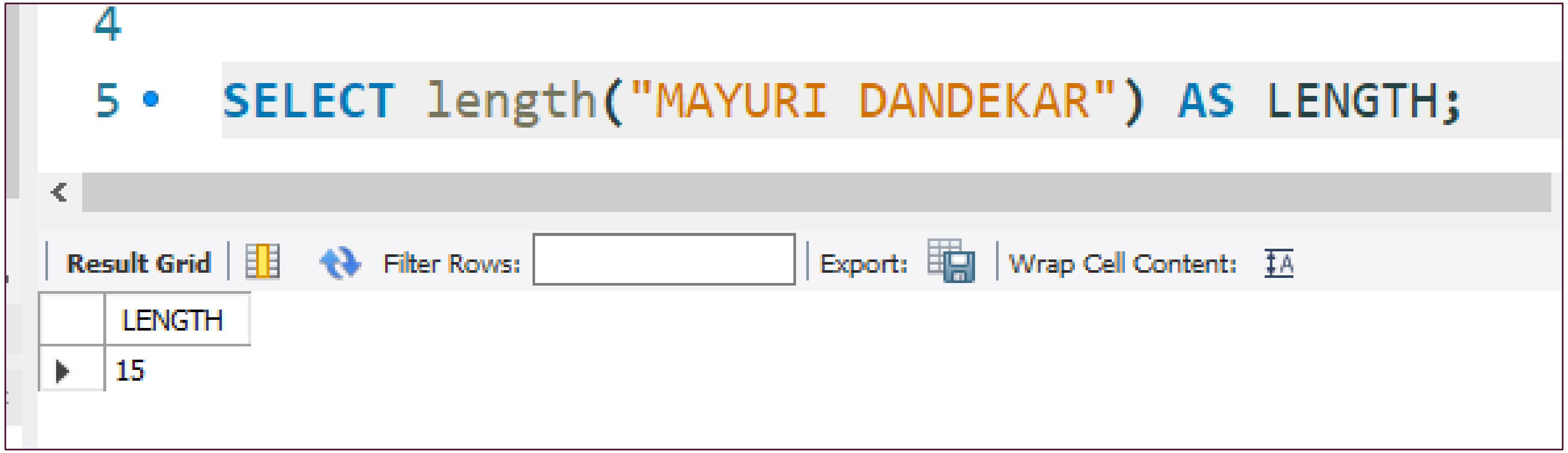
```
upper
MAYURI DANDEKAR
```



```
SELECT LOWER("MAYuri daNDEKar") as Lower;
```

```
Lower
mayuri dandekar
```

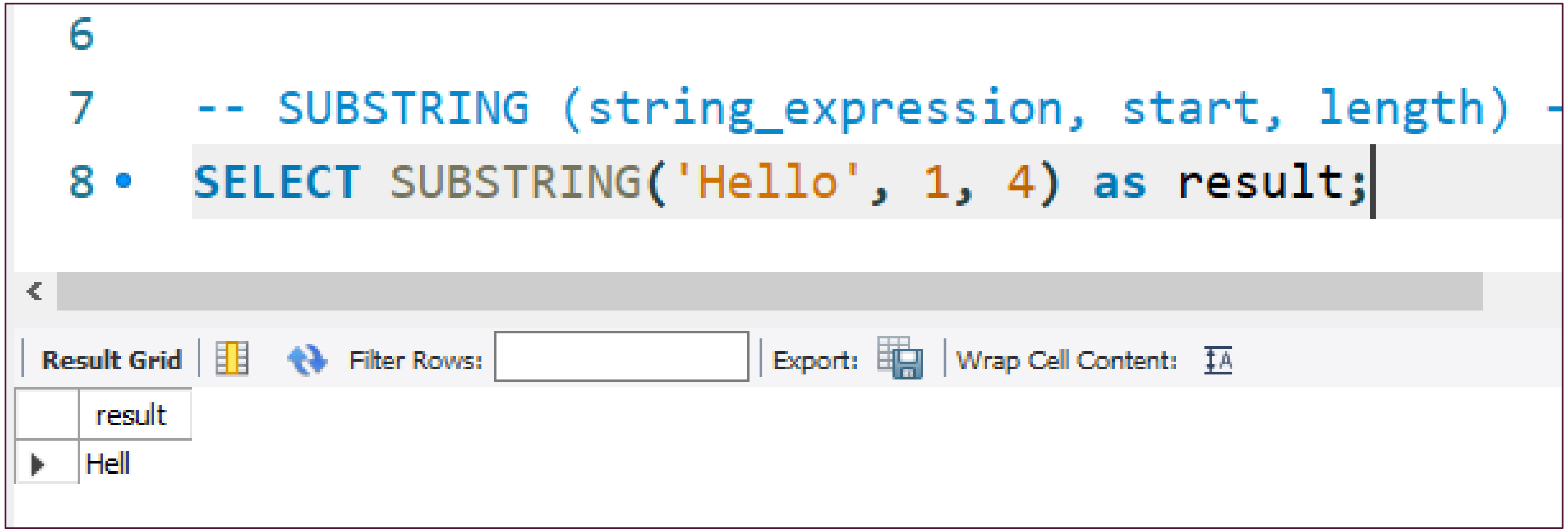
STRING FUNCTIONS – LENGTH()



```
SELECT length("MAYURI DANDEKAR") AS LENGTH;
```

```
LENGTH
15
```

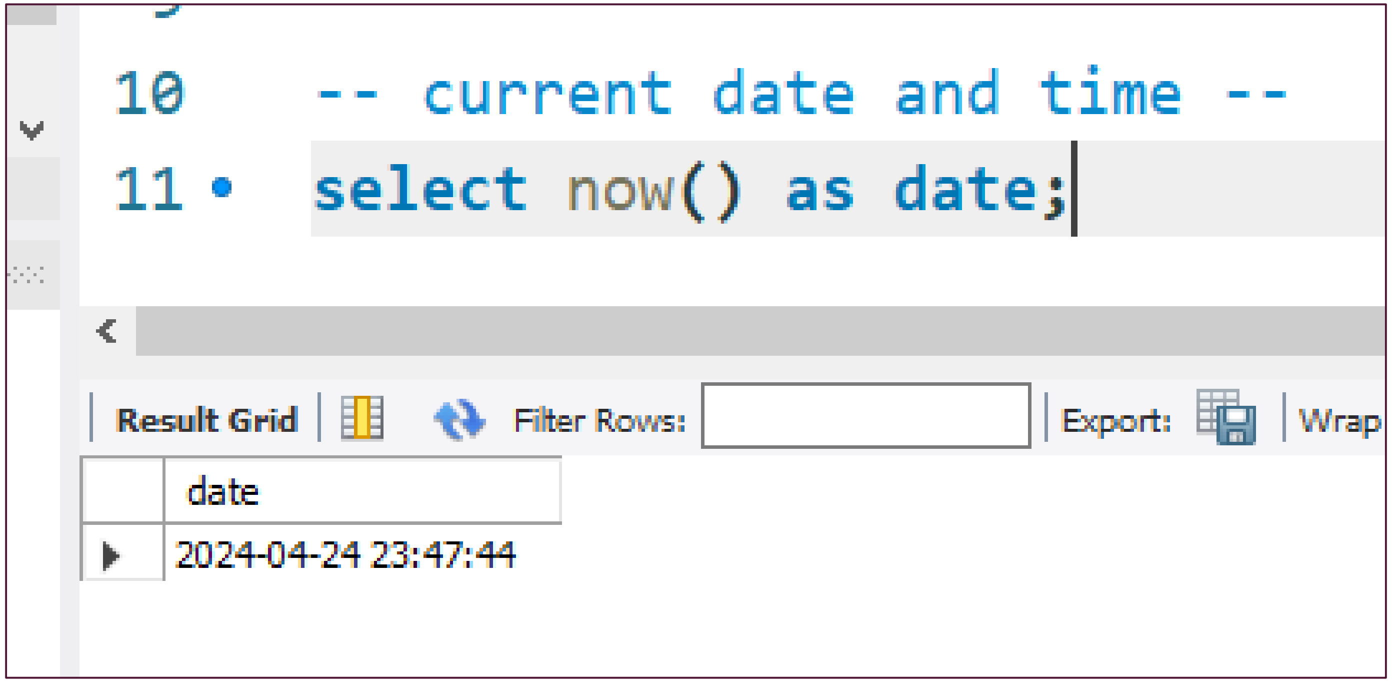
STRING FUNCTIONS – SUBSTRING()



```sql
-- SUBSTRING (string_expression, start, length) -
SELECT SUBSTRING('Hello', 1, 4) as result;
```

```
result
Hell
```

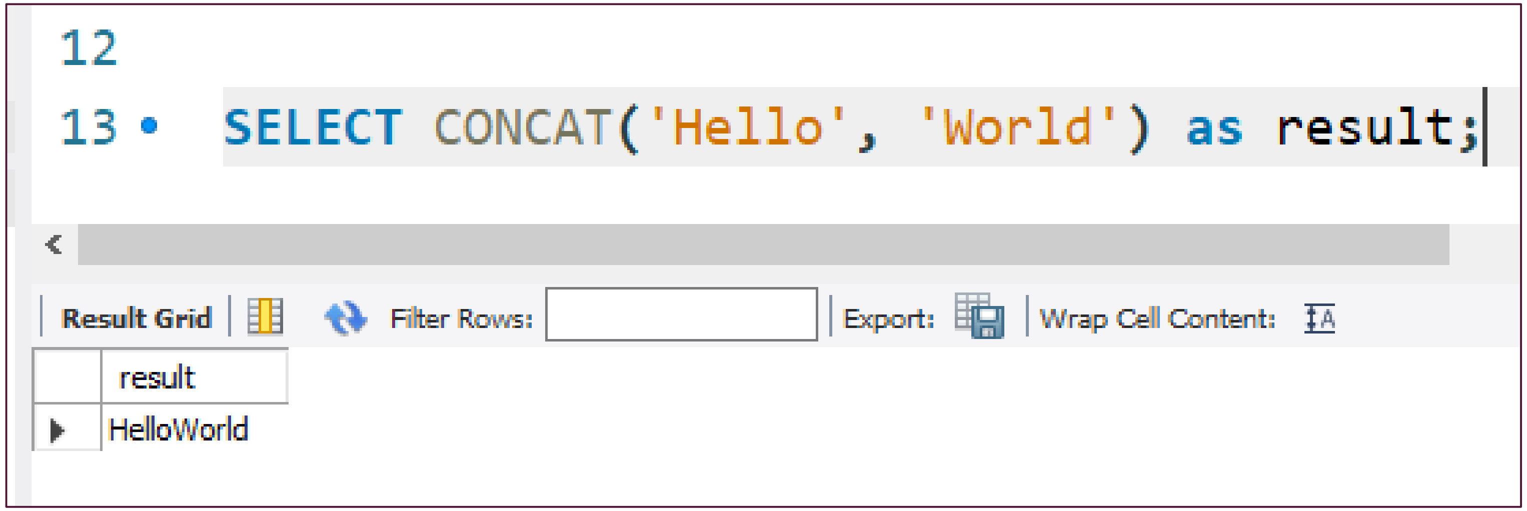
STRING FUNCTIONS – NOW()



```
-- current date and time --
select now() as date;
```

```
date
2024-04-24 23:47:44
```

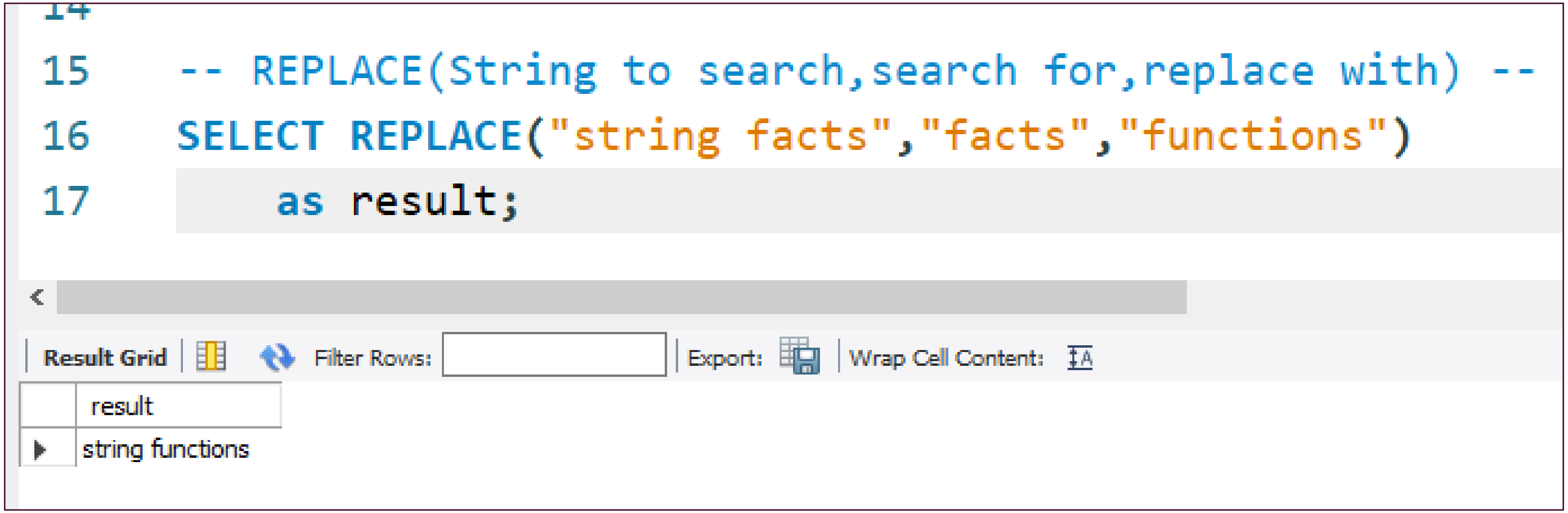
STRING FUNCTIONS – CONCAT()



```sql
SELECT CONCAT('Hello', 'World') as result;
```

```
result
HelloWorld
```

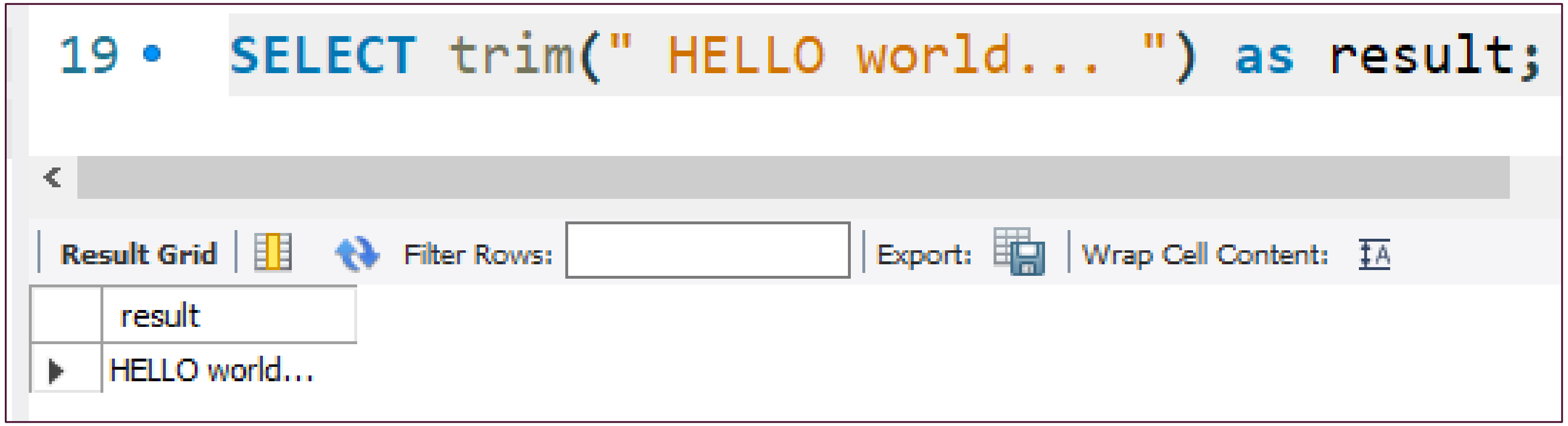
STRING FUNCTIONS – REPLACE()



```
-- REPLACE(String to search,search for,replace with) --
SELECT REPLACE("string facts","facts","functions")
    as result;
```

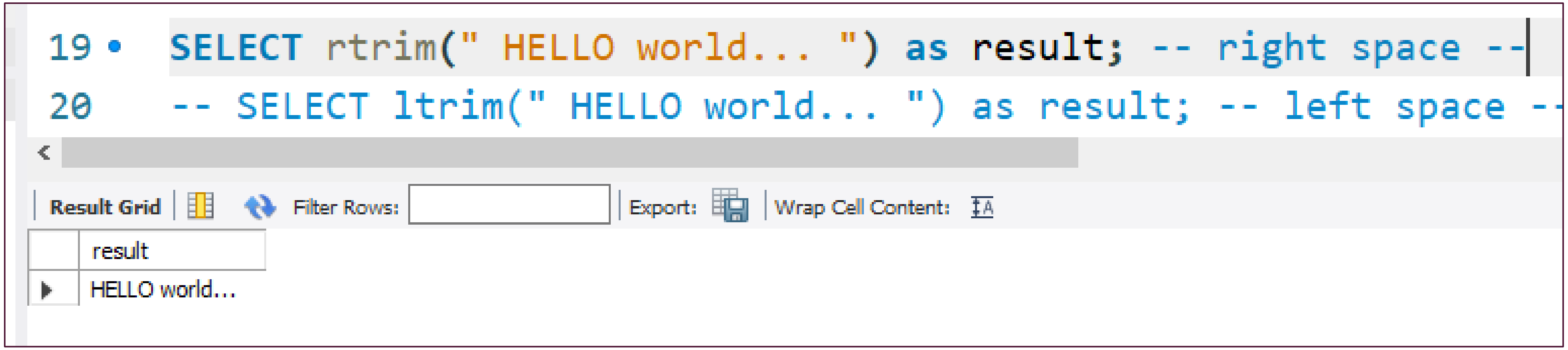
```
result
string functions
```

STRING FUNCTIONS – TRIM()



```sql
SELECT trim(" HELLO world... ") as result;
```

```
result
HELLO world...
```



```sql
SELECT rtrim(" HELLO world... ") as result; -- right space --
-- SELECT ltrim(" HELLO world... ") as result; -- left space --
```

```
result
HELLO world...
```

TIMESTAMP & EXTRACT FUNCTIONS IN SQL

SQL Series Part 8

-Mayuri Dandekar

TIMESTAMP

In SQL, we use date and time data types to **store calendar information**.

TIME contains only time, format HH:MI:SS

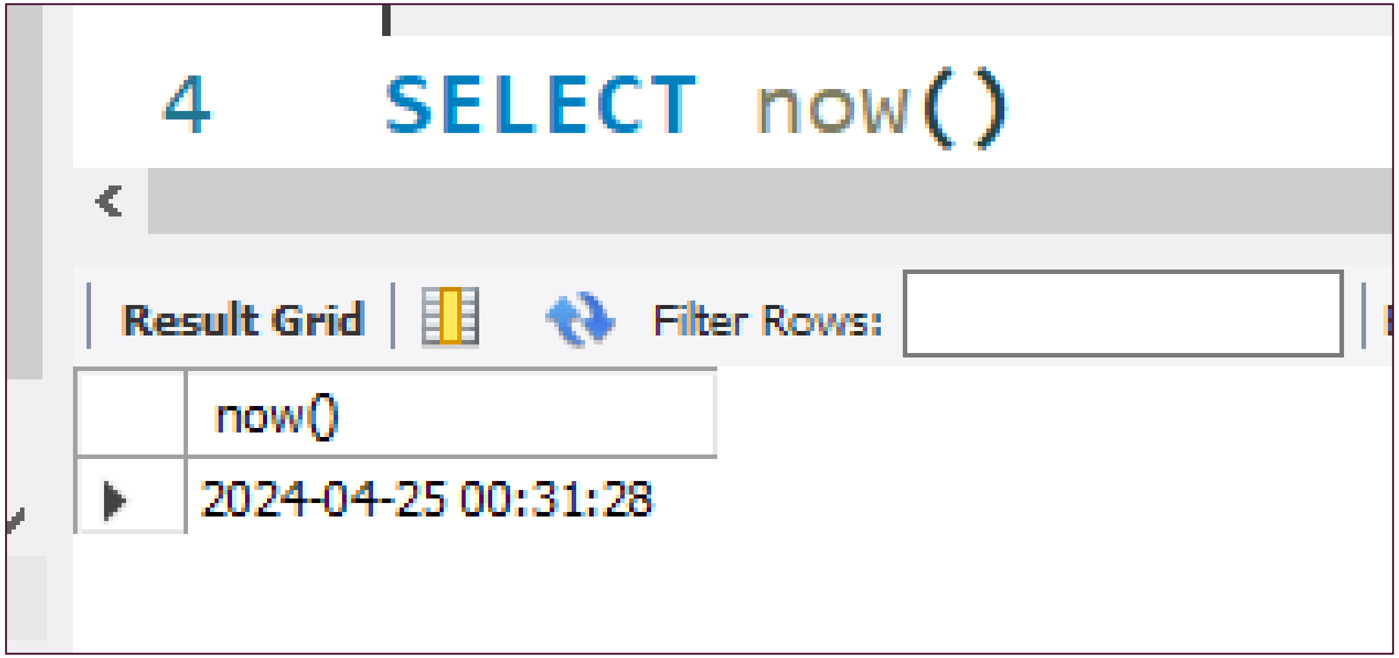
DATE contains on date, format YYYY-MM-DD

YEAR contains on year, format YYYY or YY

TIMESTAMP contains date and time, format YYYY-MM-DD HH:MI:SS

TIMESTAMPTZ contains date, time and time zone

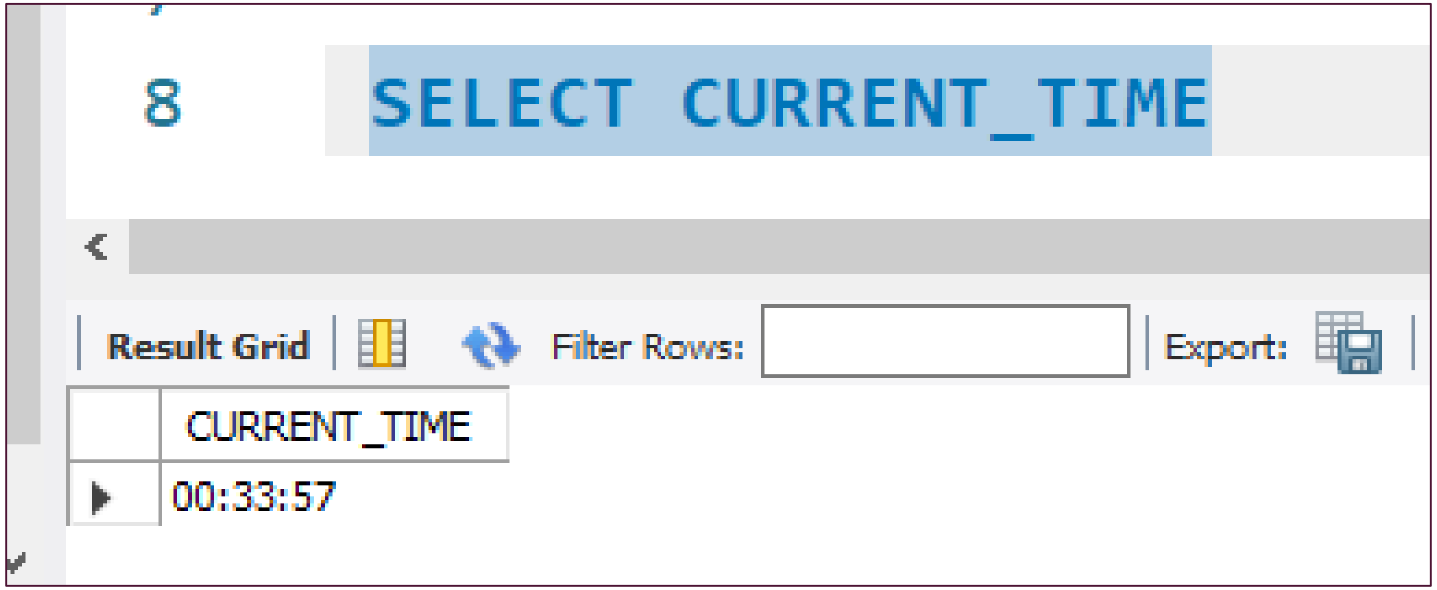
TIMESTAMP FUNCTIONS – NOW()



```
4    SELECT now()
```

```
now()
2024-04-25 00:31:28
```

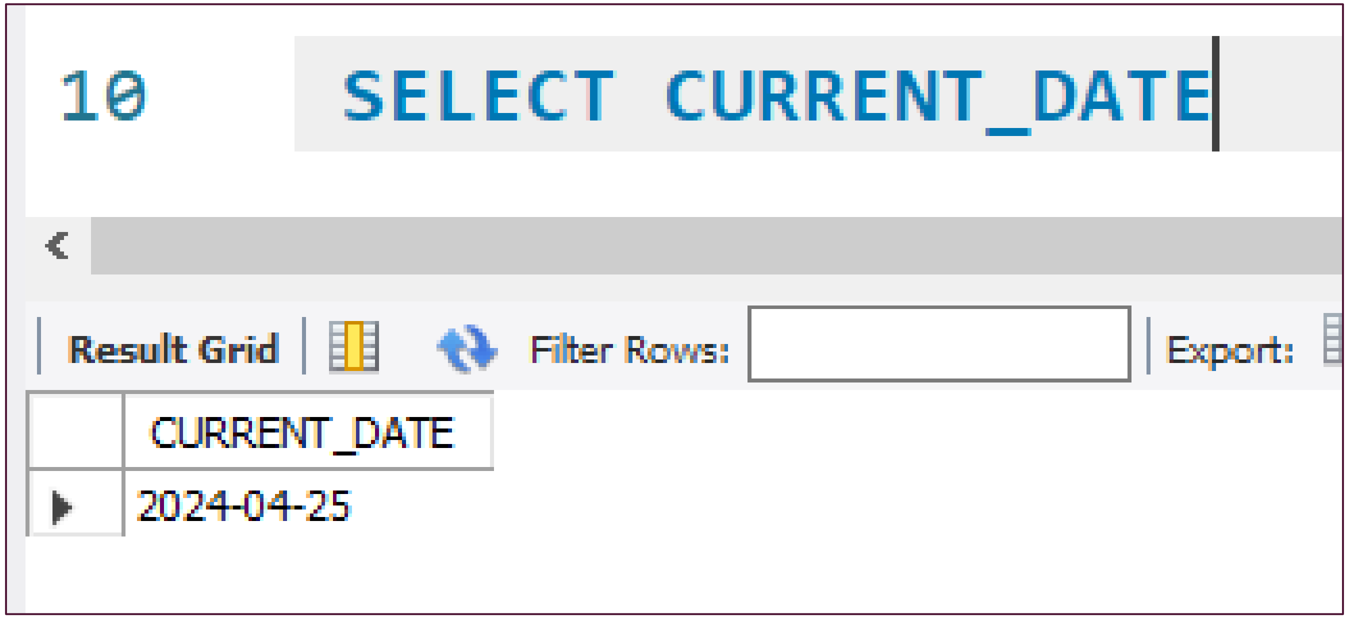
TIMESTAMP FUNCTIONS – CURRENT_TIME



```sql
SELECT CURRENT_TIME
```

```
CURRENT_TIME
00:33:57
```

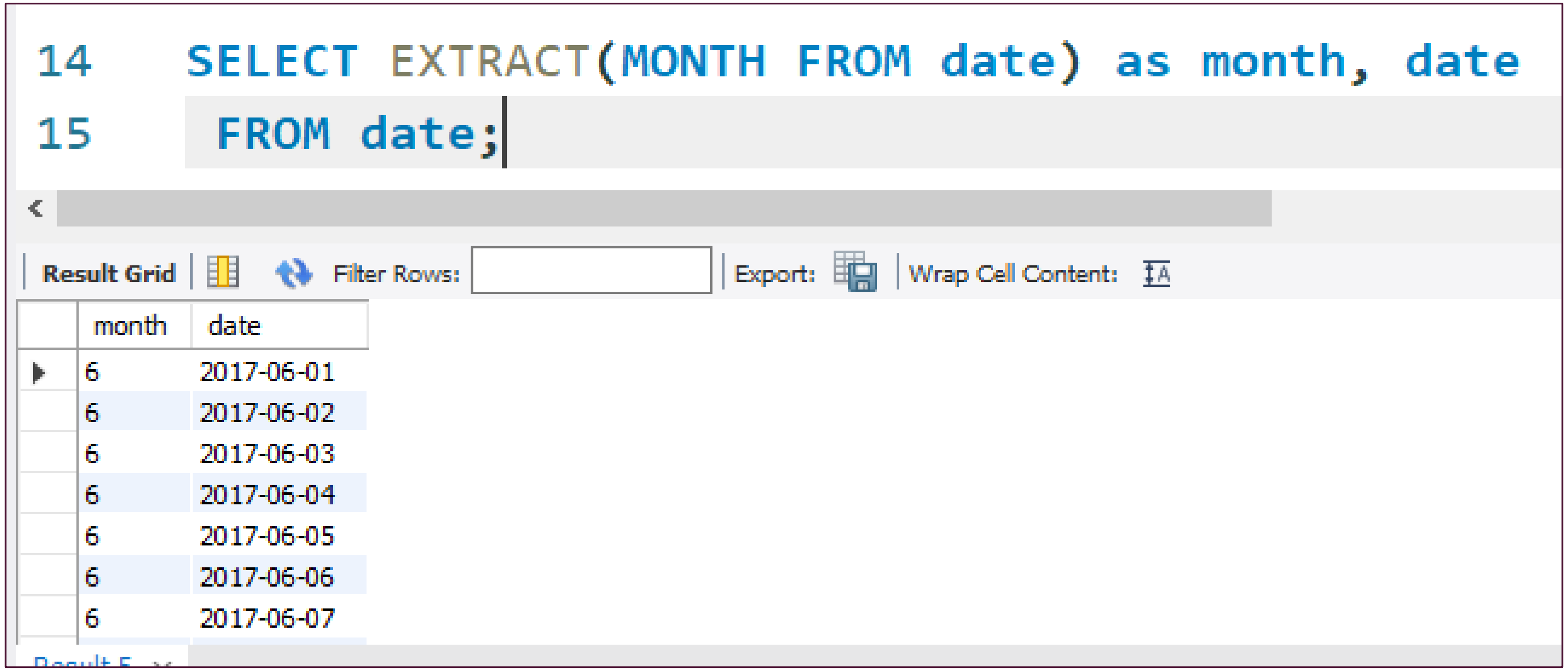
TIMESTAMP FUNCTIONS – CURRENT_DATE



```
SELECT CURRENT_DATE
```

```
CURRENT_DATE
2024-04-25
```

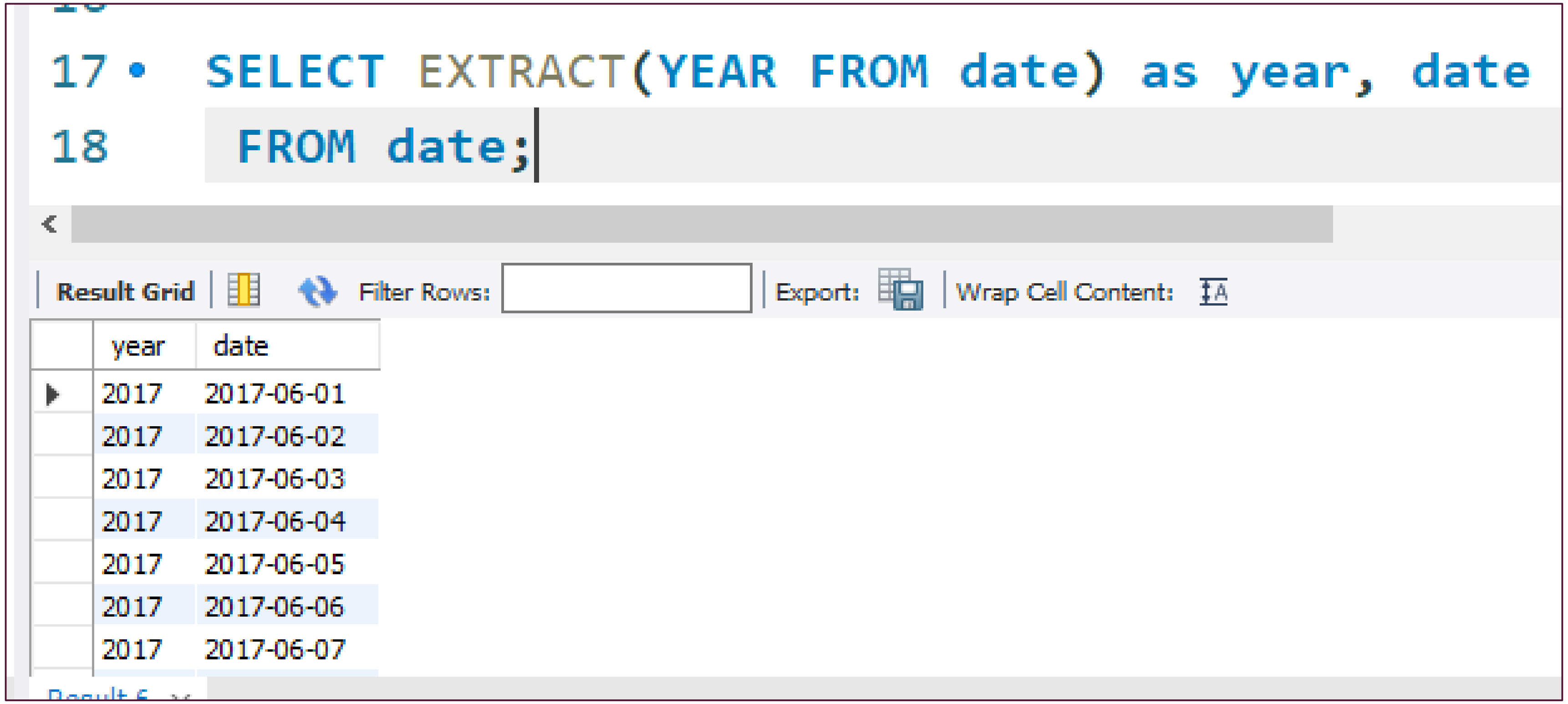
EXTRACT FUNCTIONS – MONTH



```
SELECT EXTRACT(MONTH FROM date) as month, date
 FROM date;
```

month	date
6	2017-06-01
6	2017-06-02
6	2017-06-03
6	2017-06-04
6	2017-06-05
6	2017-06-06
6	2017-06-07

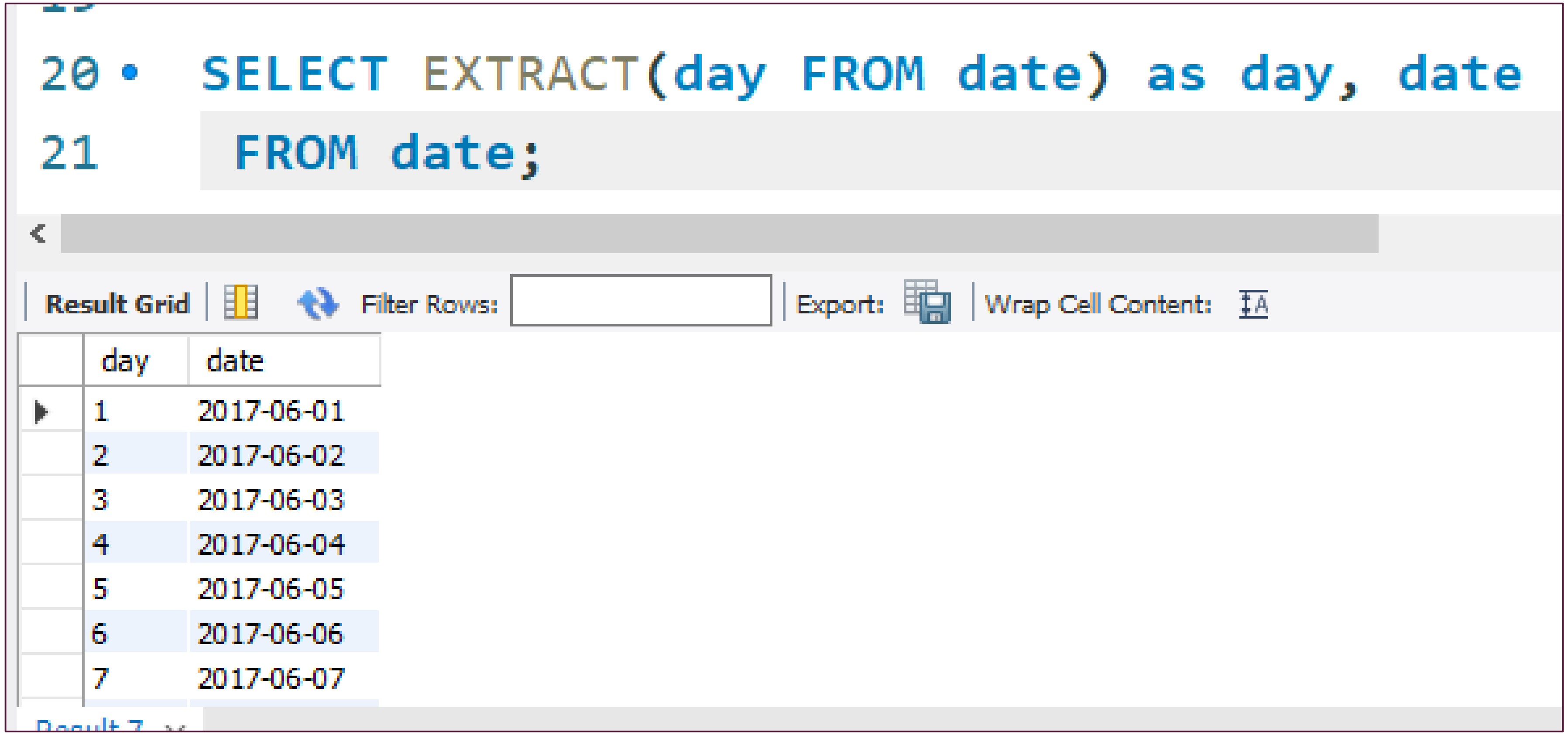
EXTRACT FUNCTIONS – YEAR



```sql
SELECT EXTRACT(YEAR FROM date) as year, date
 FROM date;
```

year	date
2017	2017-06-01
2017	2017-06-02
2017	2017-06-03
2017	2017-06-04
2017	2017-06-05
2017	2017-06-06
2017	2017-06-07

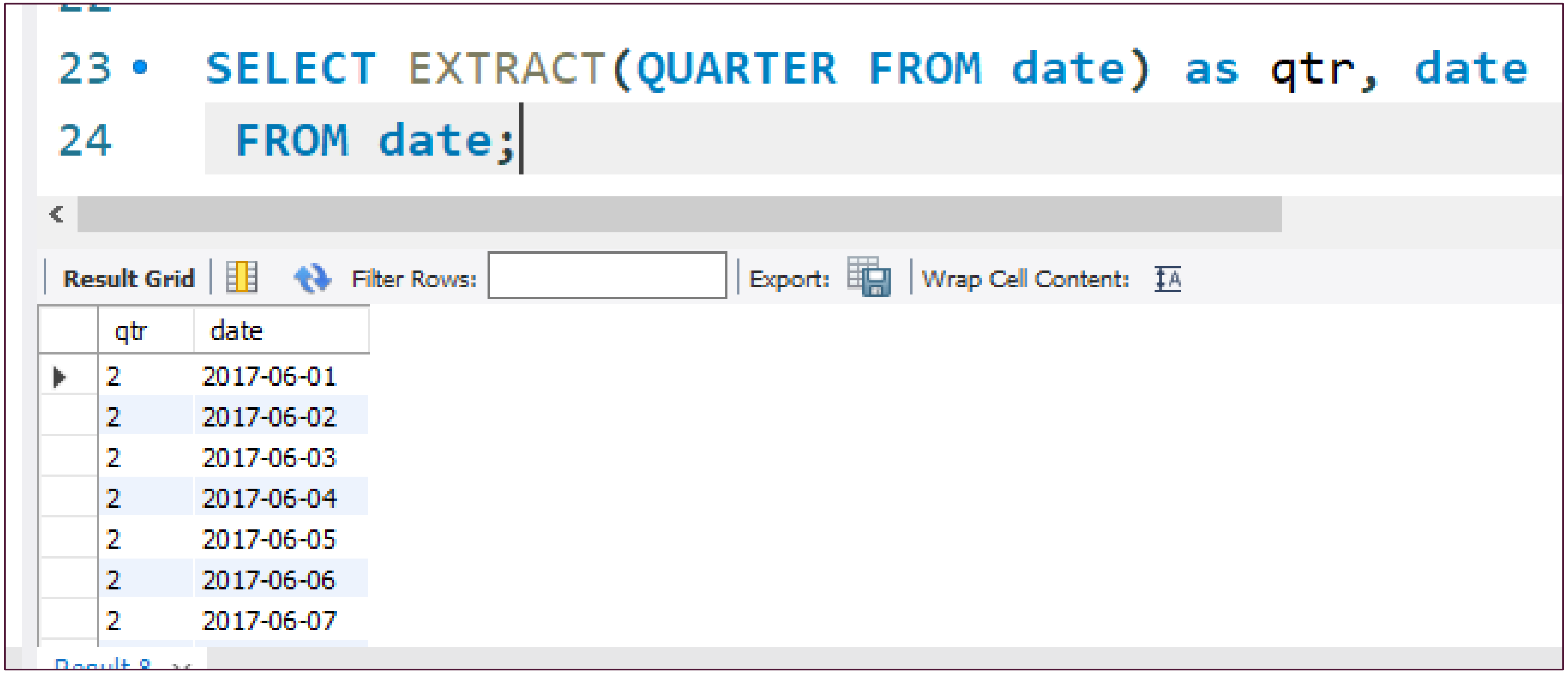
EXTRACT FUNCTIONS – DAY



```sql
SELECT EXTRACT(day FROM date) as day, date
 FROM date;
```

day	date
1	2017-06-01
2	2017-06-02
3	2017-06-03
4	2017-06-04
5	2017-06-05
6	2017-06-06
7	2017-06-07

EXTRACT FUNCTIONS – QUARTER



```
SELECT EXTRACT(QUARTER FROM date) as qtr, date
 FROM date;
```

qtr	date
2	2017-06-01
2	2017-06-02
2	2017-06-03
2	2017-06-04
2	2017-06-05
2	2017-06-06
2	2017-06-07

JOINS IN SQL

SQL Series Part 9

-Mayuri Dandekar

JOINS

JOIN is a method of **combining information** from two tables.

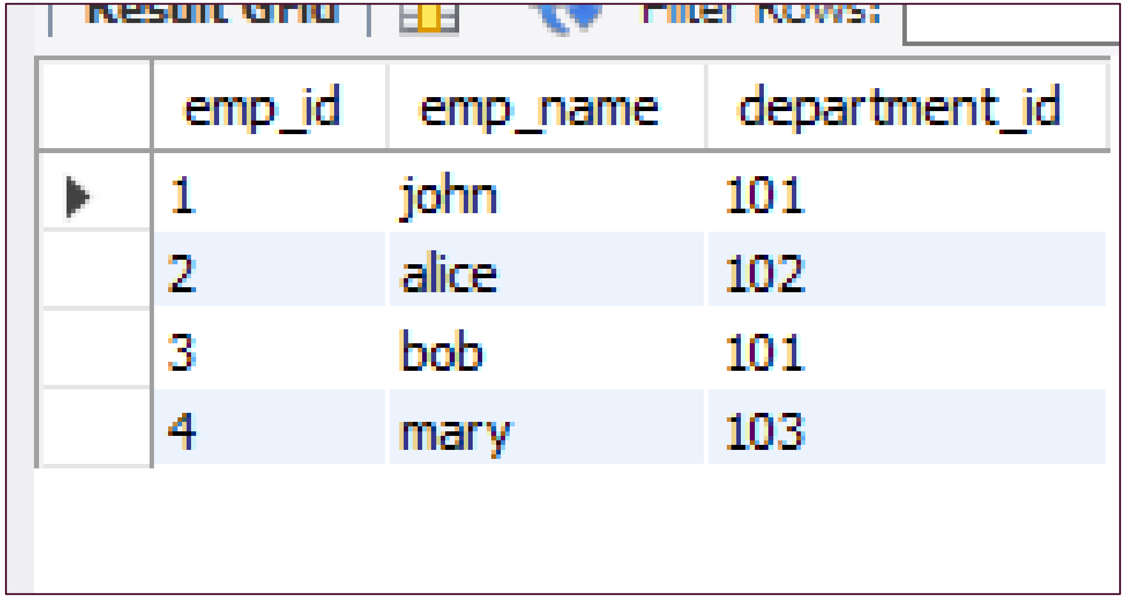
INNER JOIN -- Returns records that have matching values in both tables

LEFT JOIN -- Returns all records from the left table, and the matched records from the right table

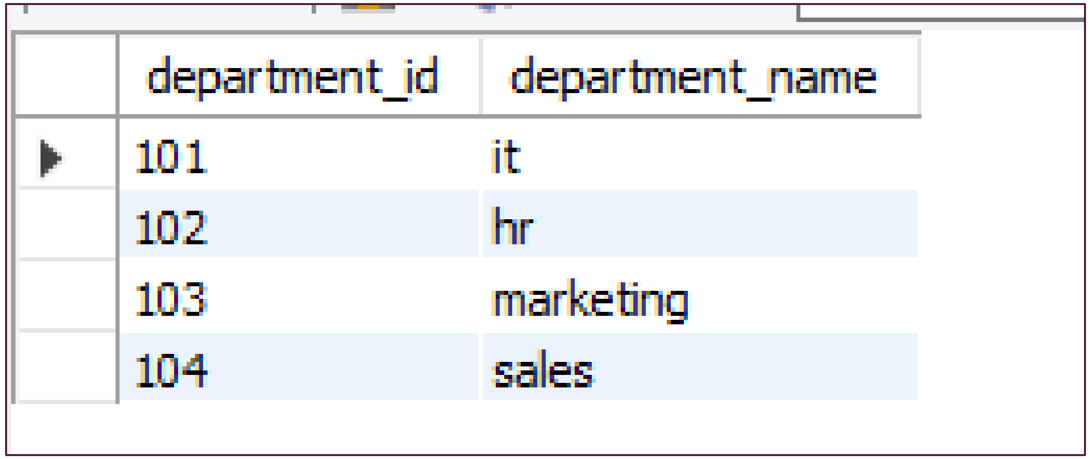
RIGHT JOIN -- Returns all records from the right table, and the matched records from the left table

FULL JOIN -- Returns all records when there is a match in either left or right table

Sample dataset used



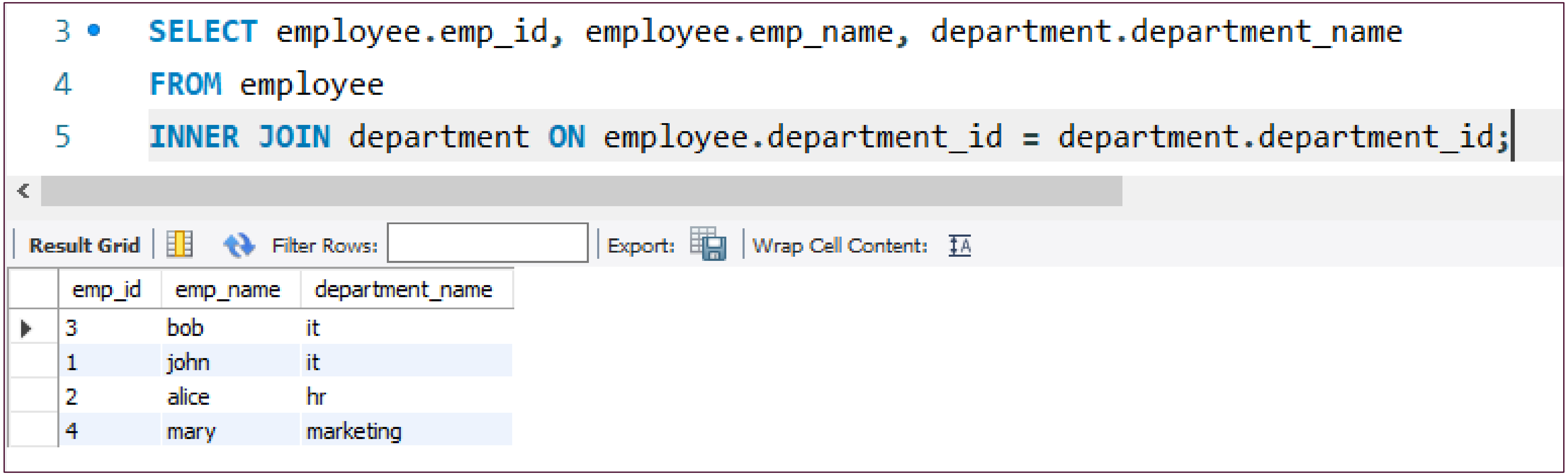
	emp_id	emp_name	department_id
▶	1	john	101
	2	alice	102
	3	bob	101
	4	mary	103



	department_id	department_name
▶	101	it
	102	hr
	103	marketing
	104	sales

INNER JOIN

An INNER JOIN retrieves rows from both tables where **there is a match in the specified columns**.

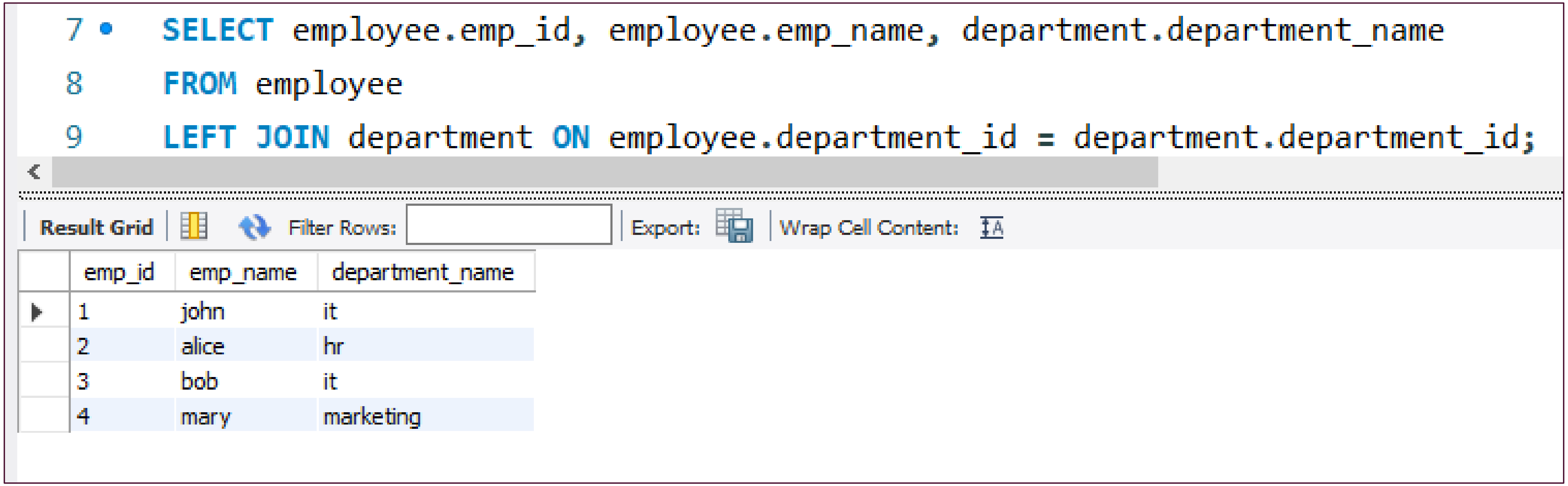


```sql
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
INNER JOIN department ON employee.department_id = department.department_id;
```

emp_id	emp_name	department_name
3	bob	it
1	john	it
2	alice	hr
4	mary	marketing

LEFT JOIN

A LEFT JOIN returns all rows from the left table and matching rows from the right table, with NULL values where there is no match in the right table.

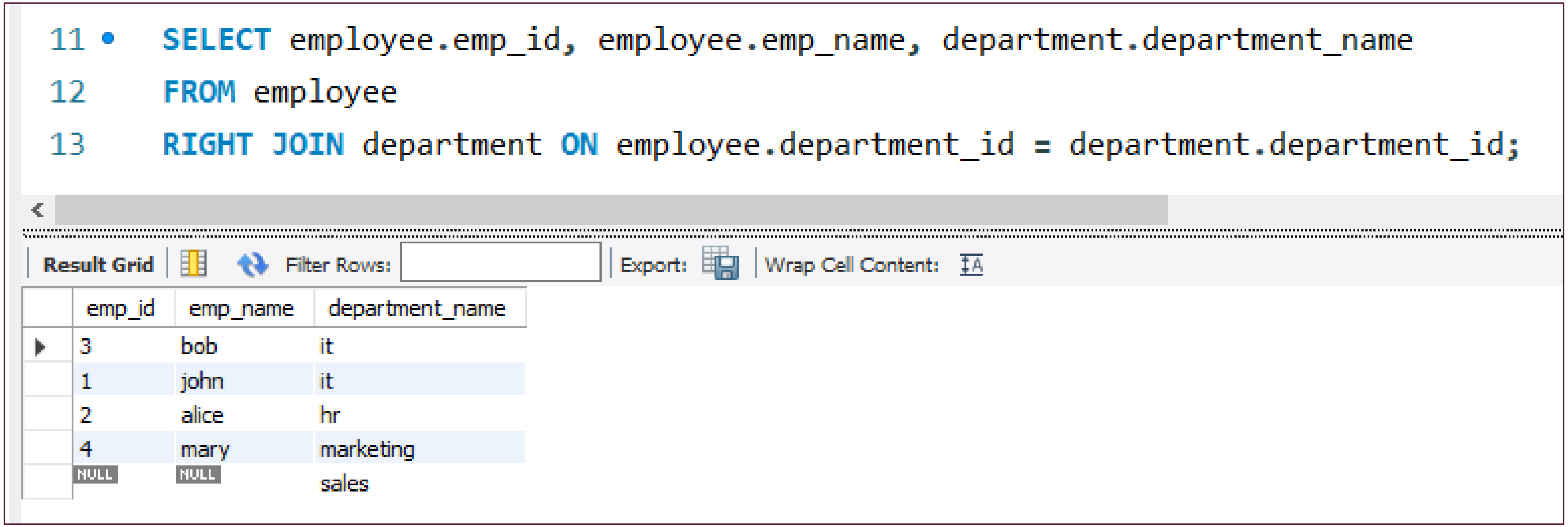


```
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
LEFT JOIN department ON employee.department_id = department.department_id;
```

emp_id	emp_name	department_name
1	john	it
2	alice	hr
3	bob	it
4	mary	marketing

RIGHT JOIN

A RIGHT JOIN returns all rows from the right table and matching rows from the left table, with NULL values where there is no match in the left table

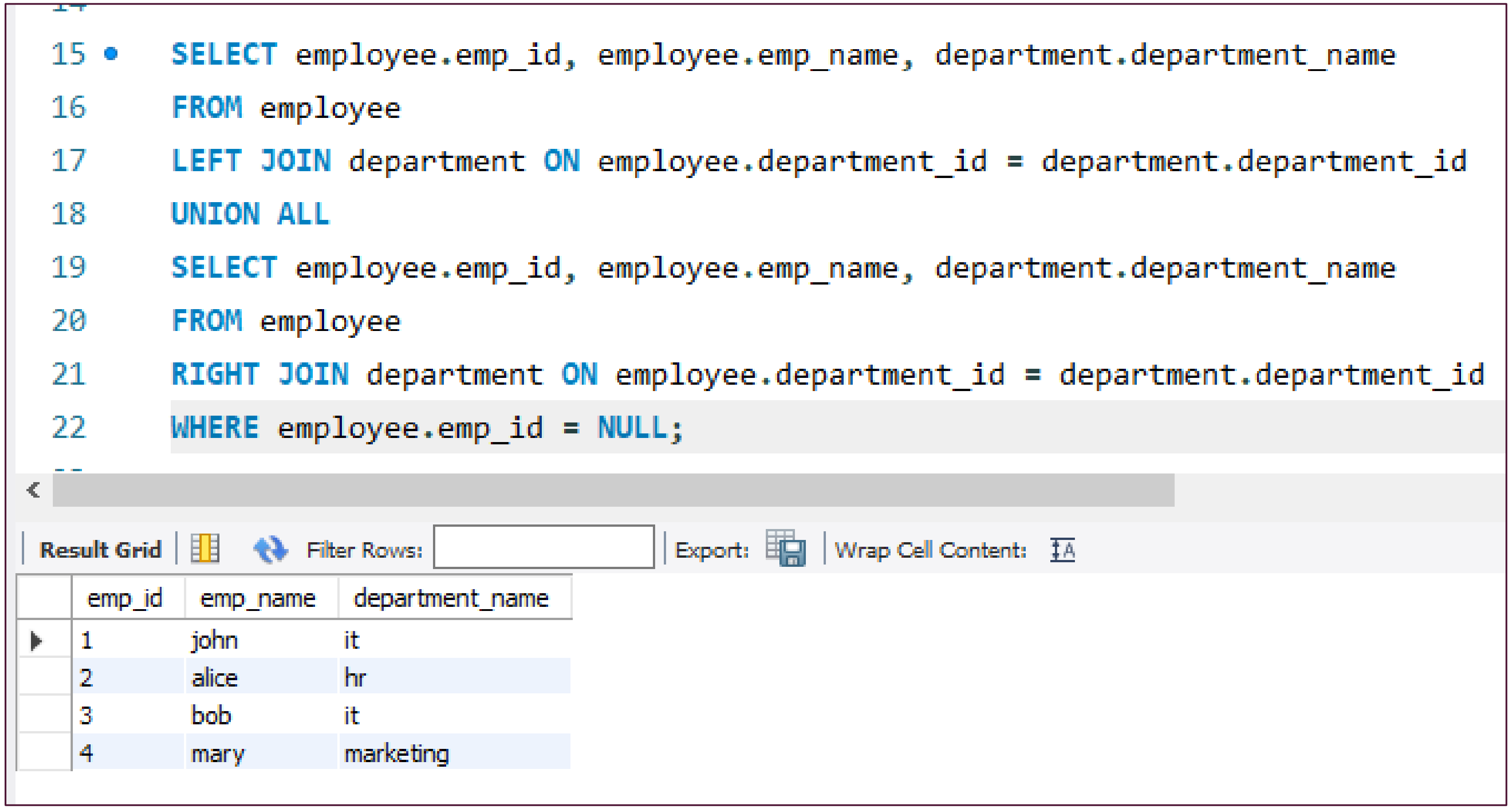


```sql
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
RIGHT JOIN department ON employee.department_id = department.department_id;
```

emp_id	emp_name	department_name
3	bob	it
1	john	it
2	alice	hr
4	mary	marketing
NULL	NULL	sales

FULL JOIN

A FULL JOIN returns all rows from both tables and NULL values where there is no match.



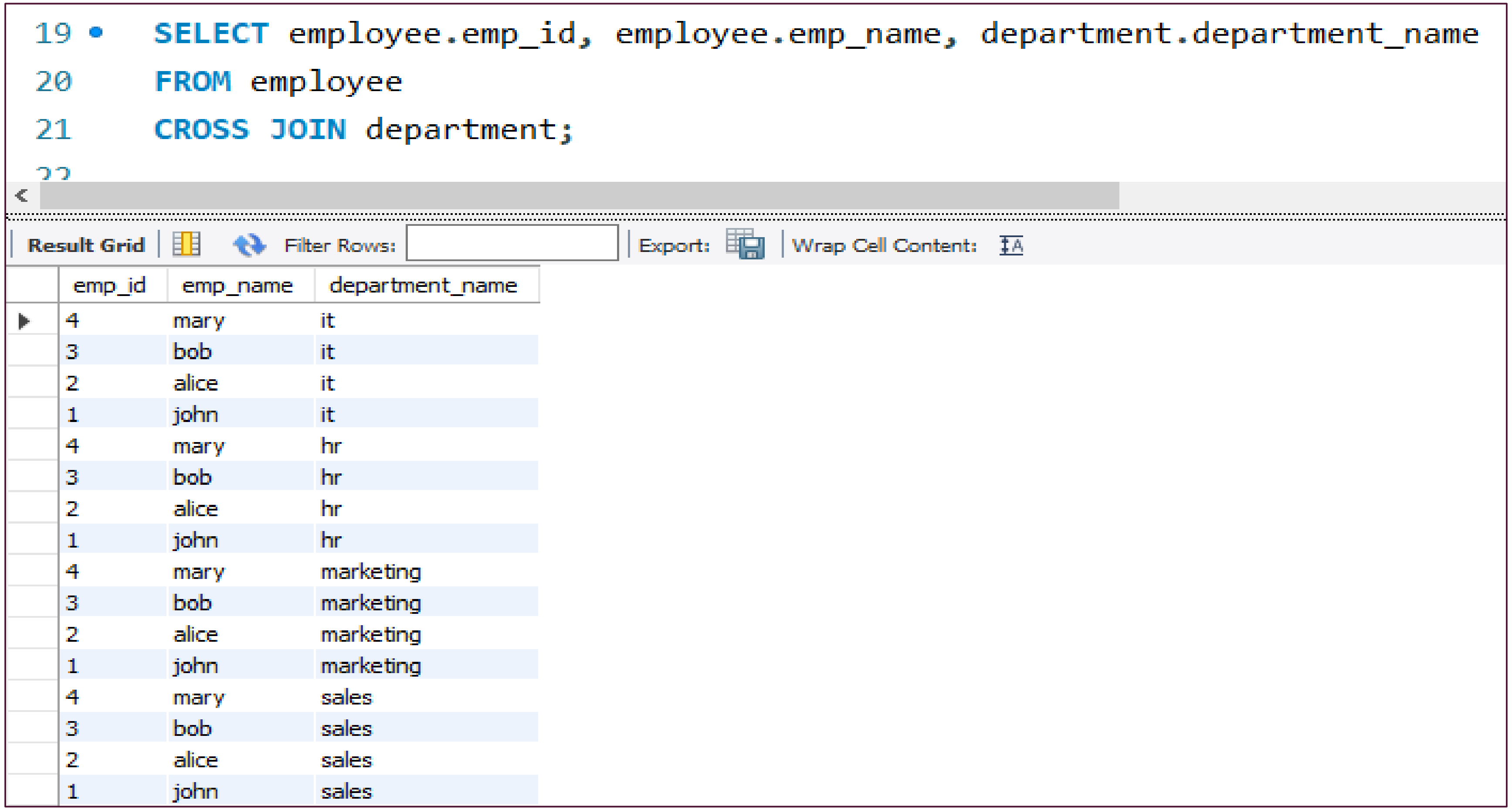
```sql
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
LEFT JOIN department ON employee.department_id = department.department_id
UNION ALL
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
RIGHT JOIN department ON employee.department_id = department.department_id
WHERE employee.emp_id = NULL;
```

emp_id	emp_name	department_name
1	john	it
2	alice	hr
3	bob	it
4	mary	marketing

Note—
My database does not support full join directly, so I tried it by combining LEFT JOIN, RIGHT JOIN & UNION ALL

CROSS JOIN

A CROSS JOIN returns the Cartesian product of the two tables, meaning it combines every row from the first table with every row from the second table.

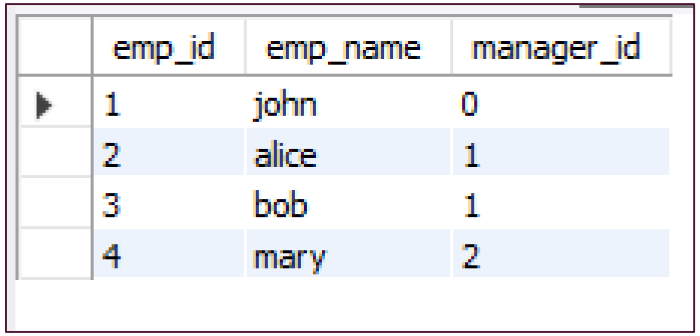


```sql
SELECT employee.emp_id, employee.emp_name, department.department_name
FROM employee
CROSS JOIN department;
```

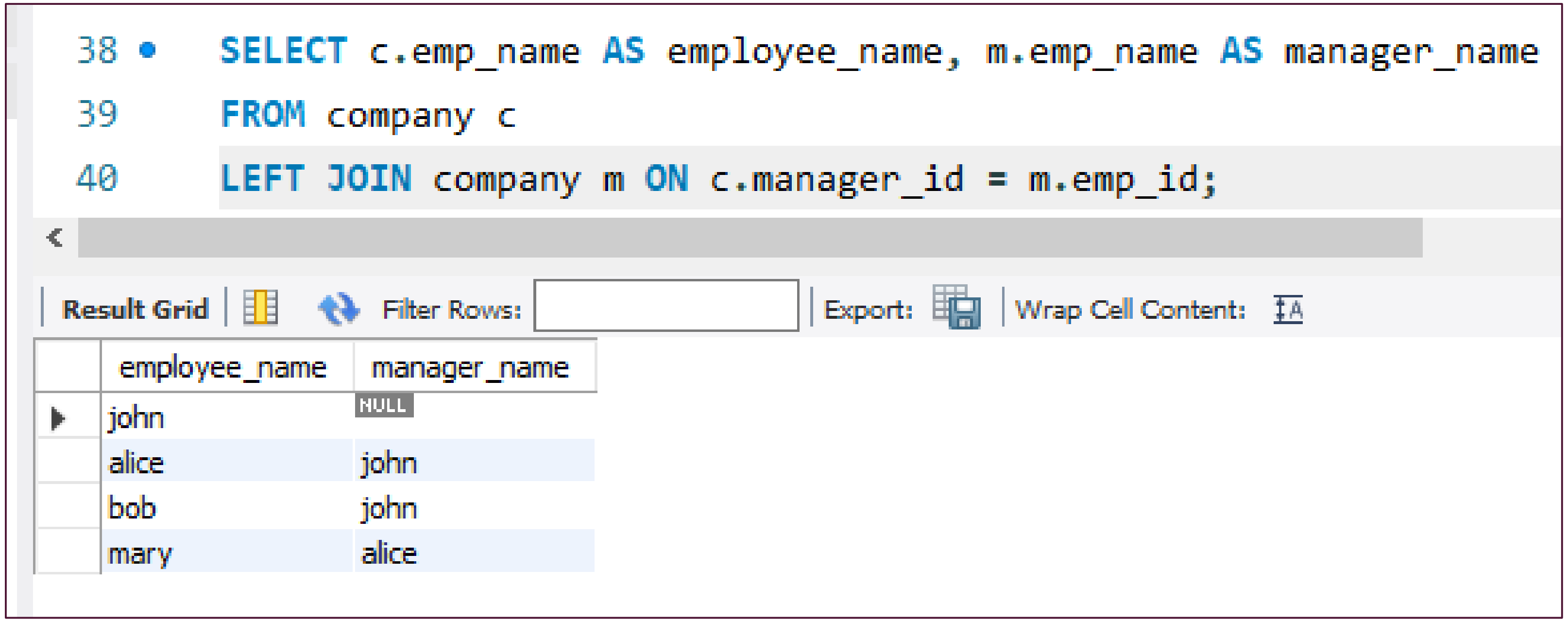
emp_id	emp_name	department_name
4	mary	it
3	bob	it
2	alice	it
1	john	it
4	mary	hr
3	bob	hr
2	alice	hr
1	john	hr
4	mary	marketing
3	bob	marketing
2	alice	marketing
1	john	marketing
4	mary	sales
3	bob	sales
2	alice	sales
1	john	sales

SELF JOIN

A self join is a special type of join where a table is joined with itself. This is useful when you have hierarchical data or need to compare rows within the same table.



emp_id	emp_name	manager_id
1	john	0
2	alice	1
3	bob	1
4	mary	2

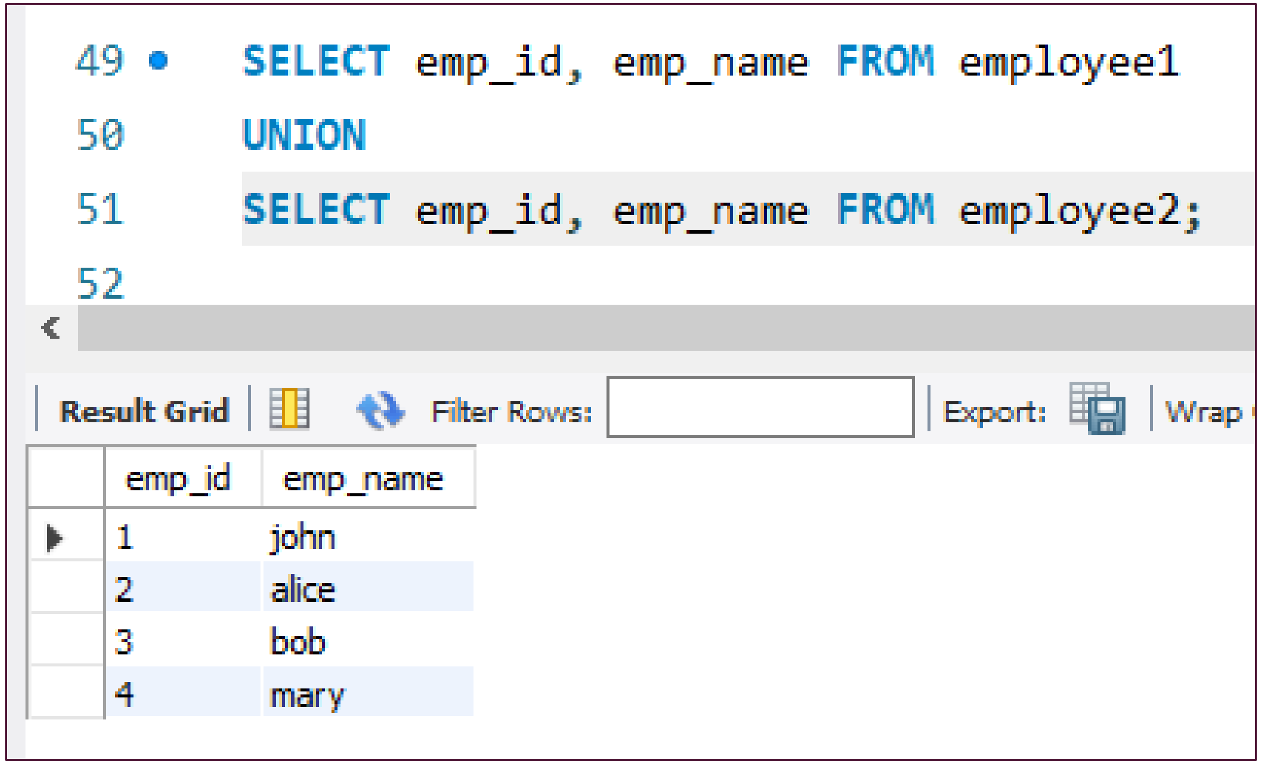


```
SELECT c.emp_name AS employee_name, m.emp_name AS manager_name
FROM company c
LEFT JOIN company m ON c.manager_id = m.emp_id;
```

employee_name	manager_name
john	NULL
alice	john
bob	john
mary	alice

UNION

UNION is used to combine the results of two or more SELECT statements into a single result set. It removes duplicate rows by default

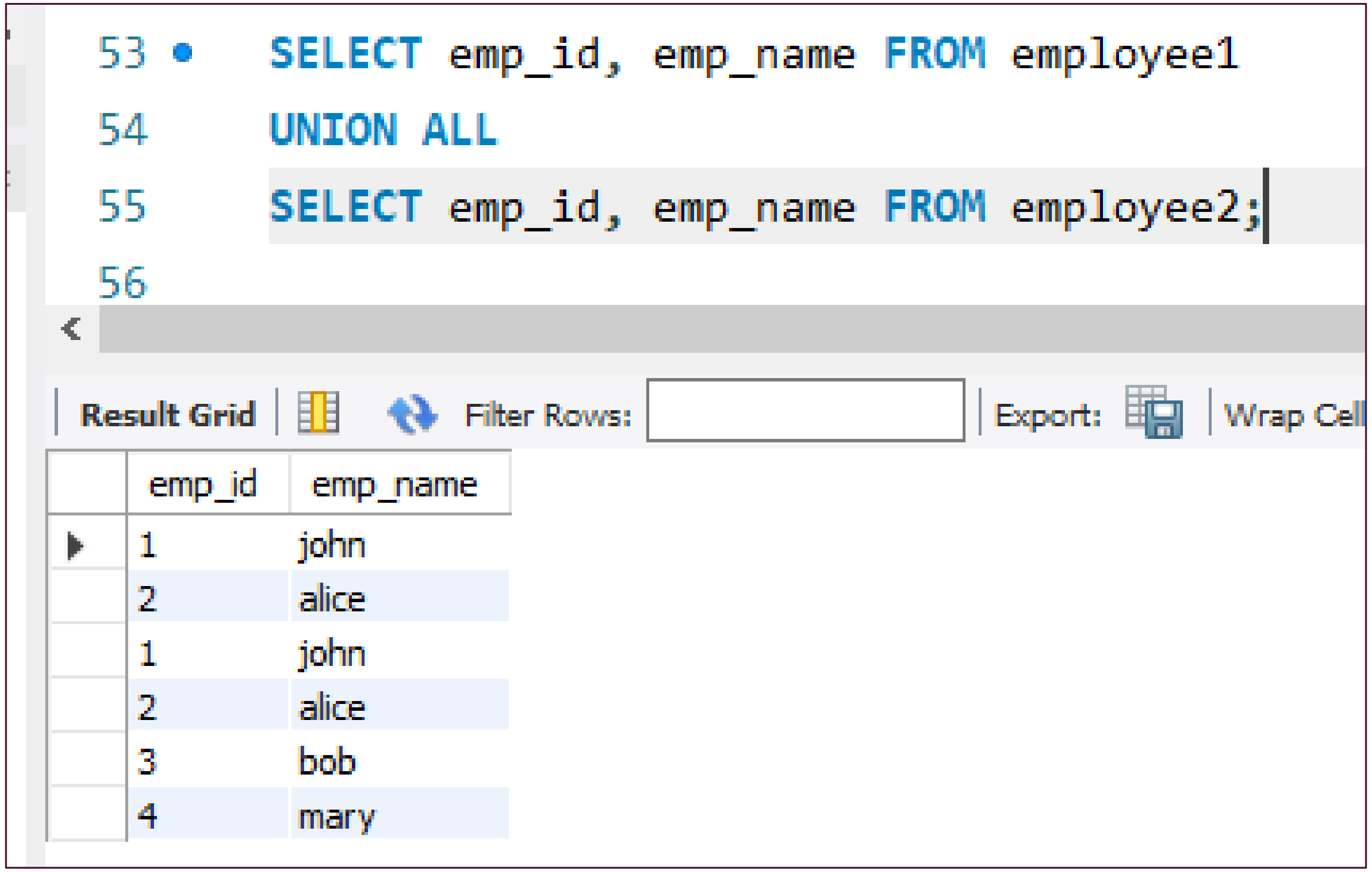


```
SELECT emp_id, emp_name FROM employee1
UNION
SELECT emp_id, emp_name FROM employee2;
```

emp_id	emp_name
1	john
2	alice
3	bob
4	mary

UNION ALL

UNION ALL is similar to UNION, but it retains duplicate rows from the combined result sets.



```
SELECT emp_id, emp_name FROM employee1
UNION ALL
SELECT emp_id, emp_name FROM employee2;
```

emp_id	emp_name
1	john
2	alice
1	john
2	alice
3	bob
4	mary

SUBQUERY IN SQL

SQL Series Part 10

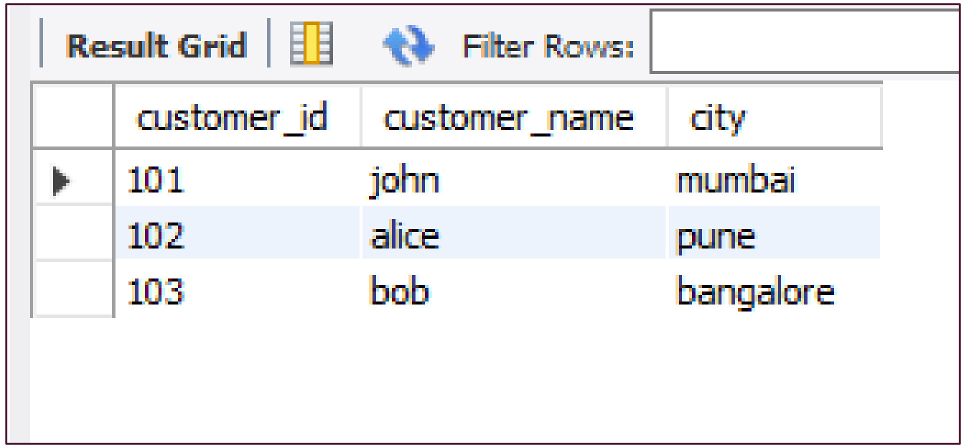
-Mayuri Dandekar

SUBQUERY

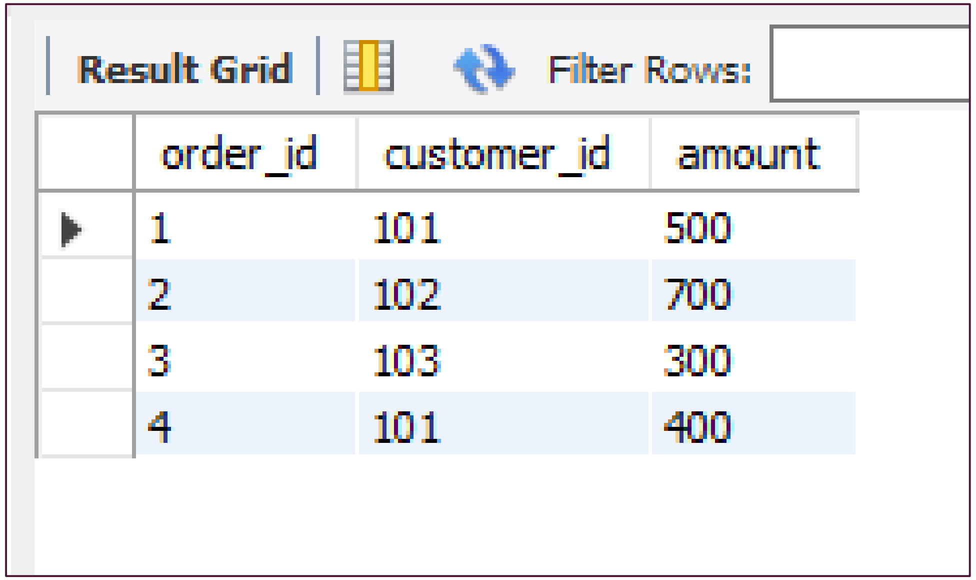
A subquery, also known as a **nested query** or **inner query**, is a query nested **within another SQL query**. Subqueries are enclosed within parentheses and can be used in various parts of a SQL statement, such as SELECT, INSERT, UPDATE, or DELETE statements.

Subqueries can be used to retrieve data based on conditions, perform calculations, filter results, or even modify data.

Example dataset

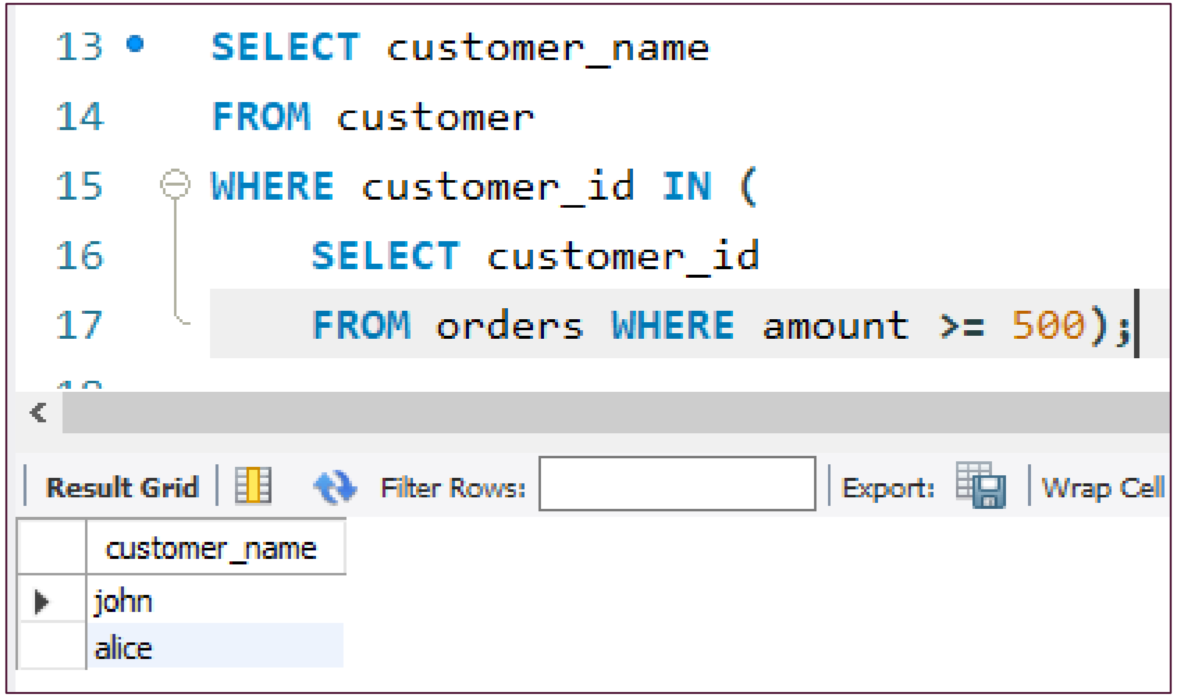


customer_id	customer_name	city
101	john	mumbai
102	alice	pune
103	bob	bangalore



order_id	customer_id	amount
1	101	500
2	102	700
3	103	300
4	101	400

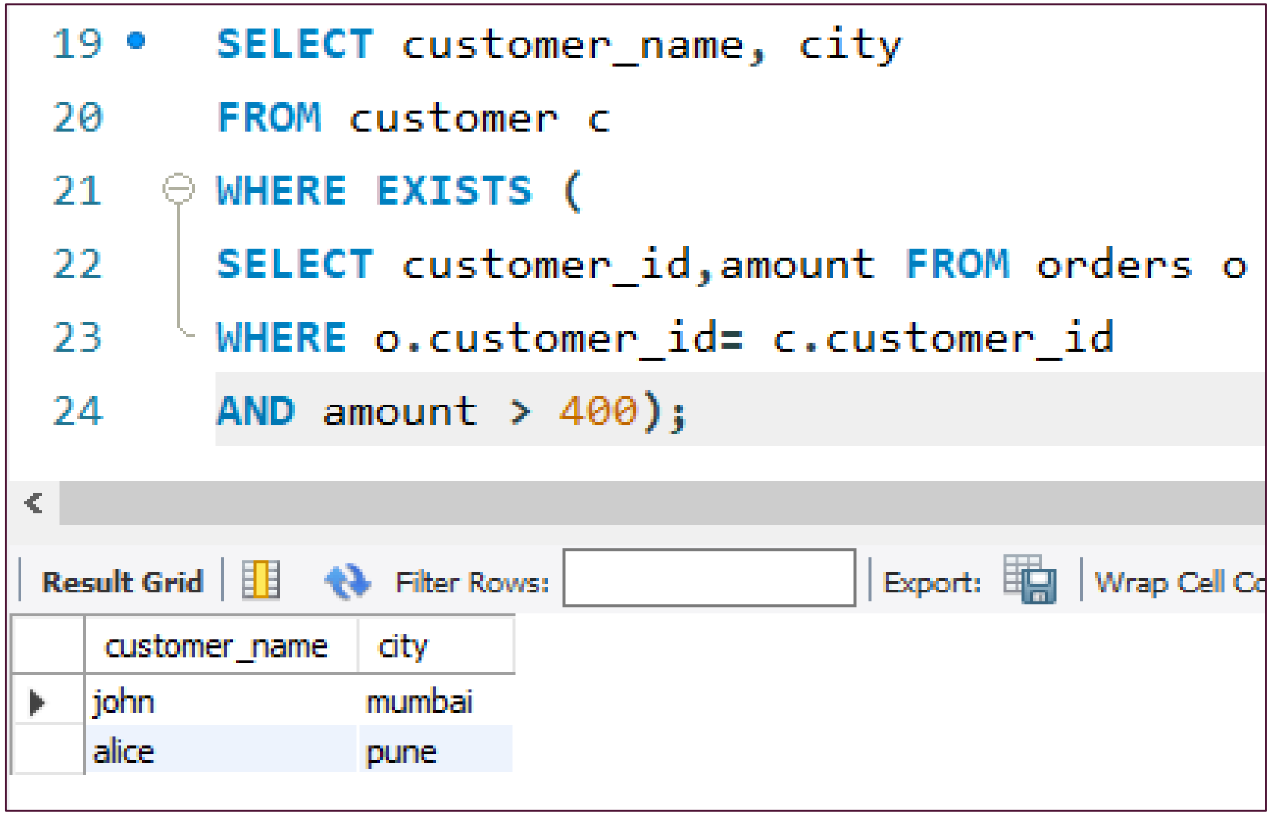
Retrieve the names of customers who have placed orders with a total amount greater than 500



```sql
SELECT customer_name
FROM customer
WHERE customer_id IN (
    SELECT customer_id
    FROM orders WHERE amount >= 500);
```

customer_name
john
alice

Details of customers, whose order amount is more than 400 with same ID’s.



```sql
SELECT customer_name, city
FROM customer c
WHERE EXISTS (
SELECT customer_id,amount FROM orders o
WHERE o.customer_id= c.customer_id
AND amount > 400);
```

customer_name	city
john	mumbai
alice	pune

WINDOW FUNCTIONS IN SQL

SQL Series Part 11

-Mayuri Dandekar

WINDOW FUNCTIONS

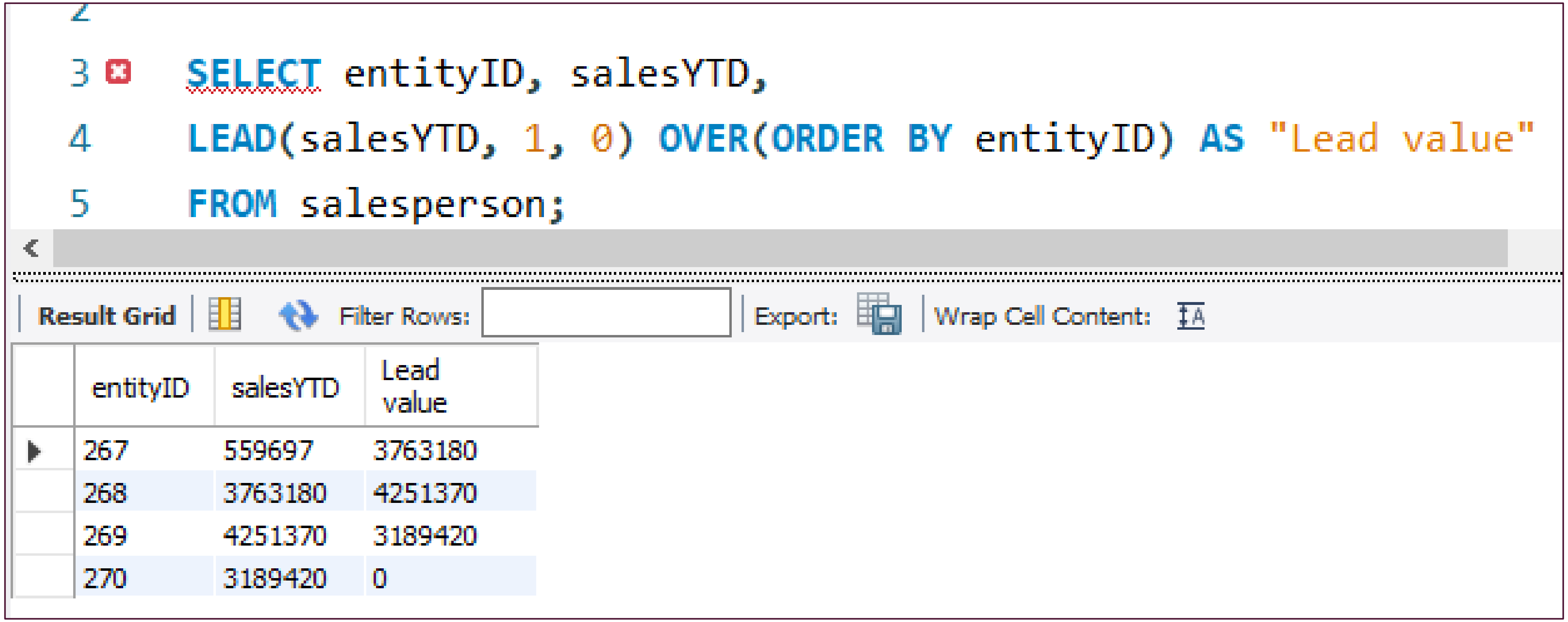
Window functions applies **aggregate, ranking and analytic functions** over a particular window (set of rows).

And **OVER clause** is used with window functions to define that window.

SYNTAX—

```
SELECT column_name(s),
fun( ) OVER ( [ <PARTITION BY Clause> ]
              [ <ORDER BY Clause> ]
              [ <ROW or RANGE Clause> ] )
FROM table_name;
```

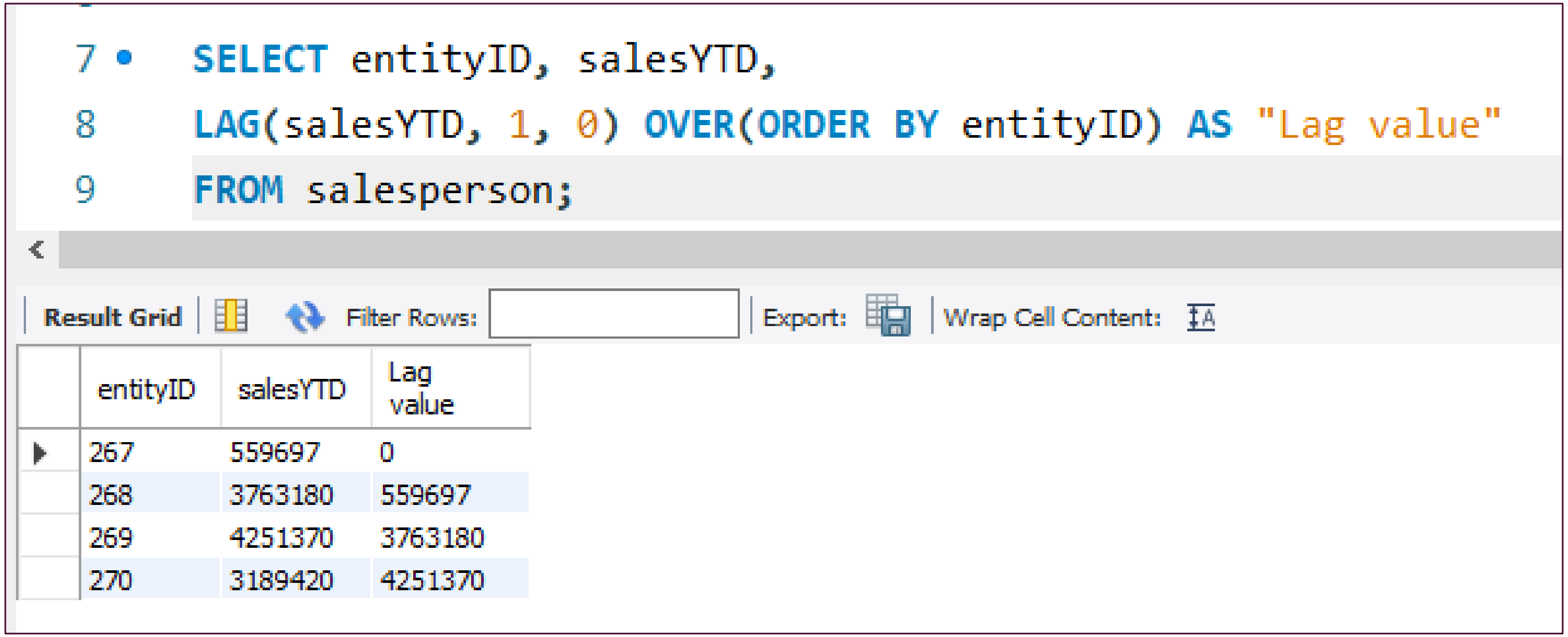
LEAD



```
SELECT entityID, salesYTD,
LEAD(salesYTD, 1, 0) OVER(ORDER BY entityID) AS "Lead value"
FROM salesperson;
```

entityID	salesYTD	Lead value
267	559697	3763180
268	3763180	4251370
269	4251370	3189420
270	3189420	0

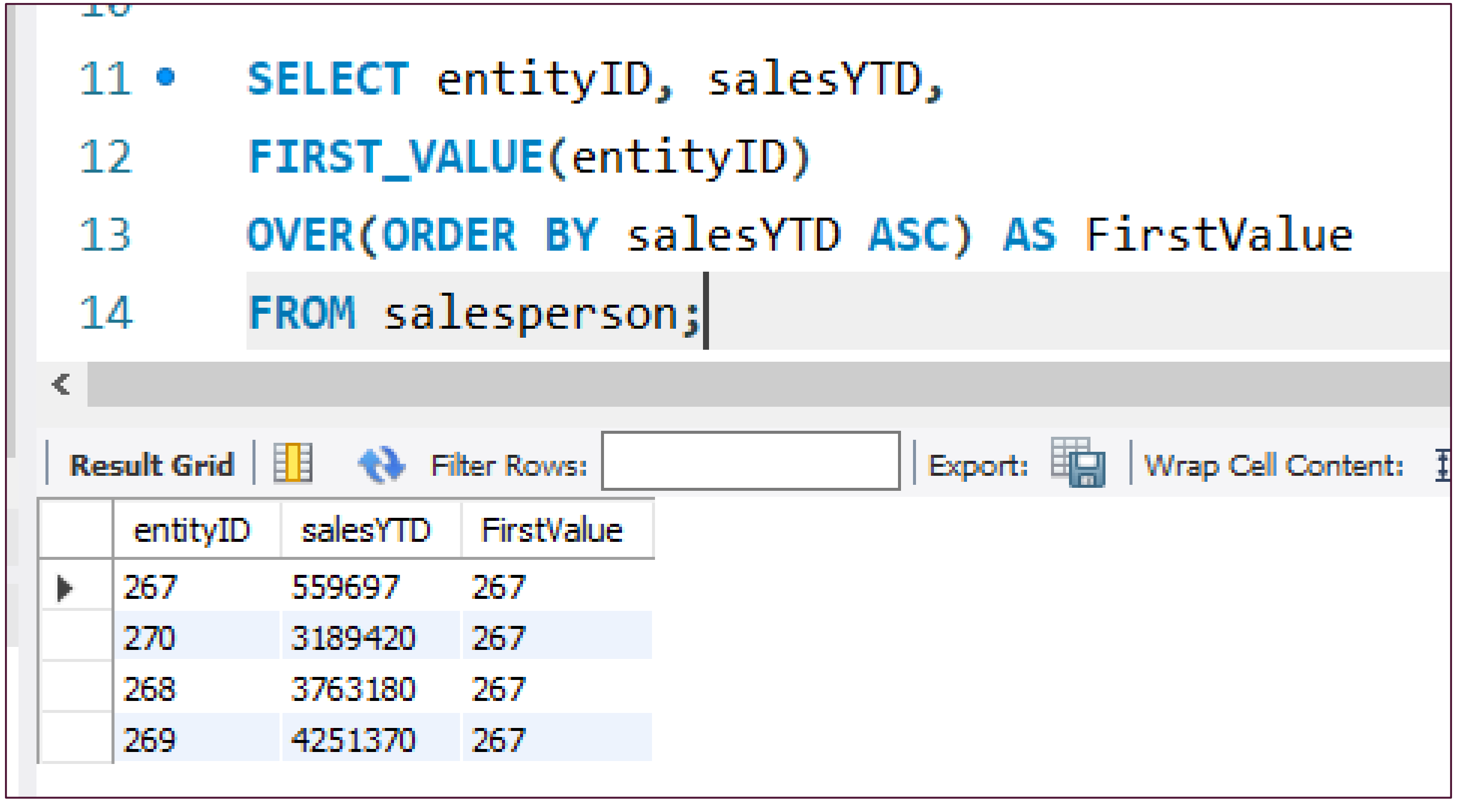
LAG



```
SELECT entityID, salesYTD,
LAG(salesYTD, 1, 0) OVER(ORDER BY entityID) AS "Lag value"
FROM salesperson;
```

entityID	salesYTD	Lag value
267	559697	0
268	3763180	559697
269	4251370	3763180
270	3189420	4251370

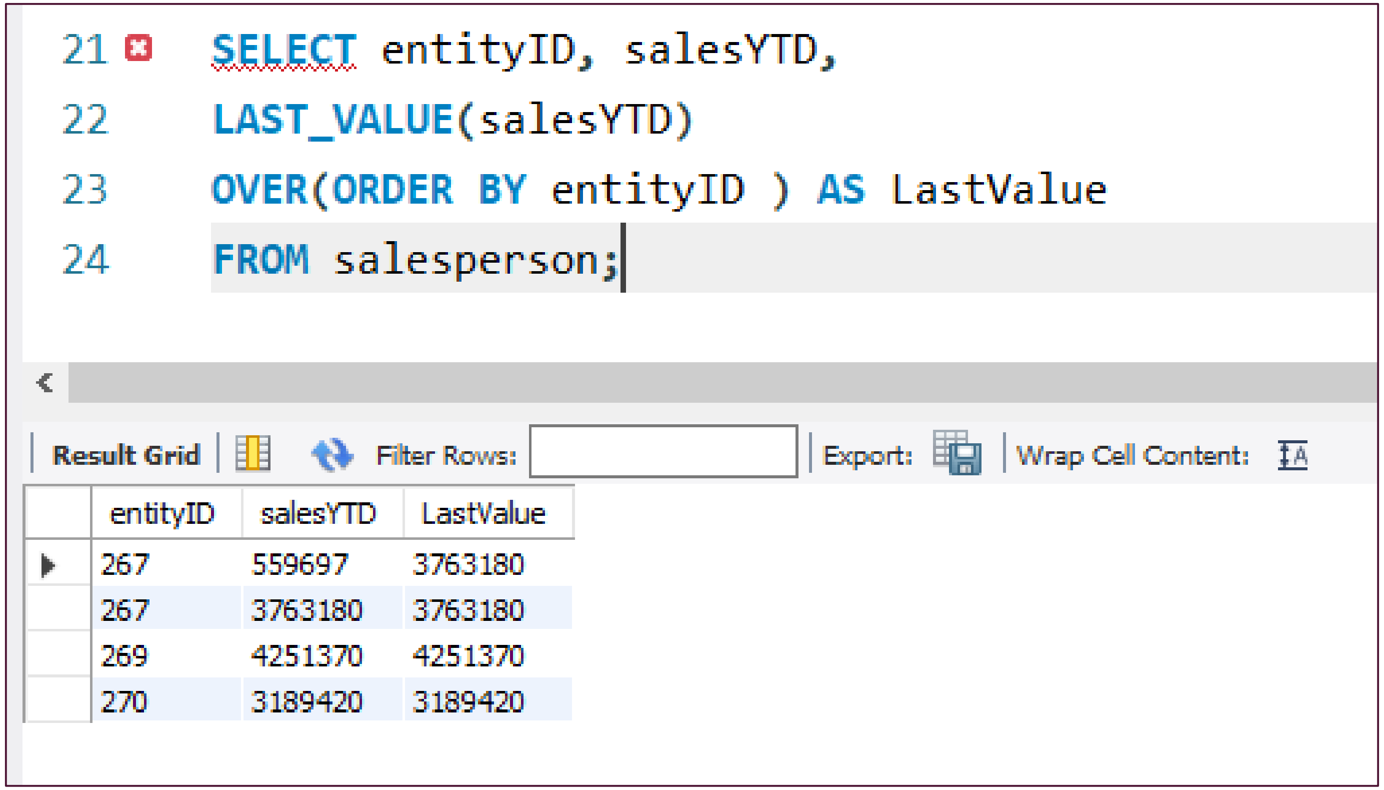
FIRST_VALUE



```
SELECT entityID, salesYTD,
FIRST_VALUE(entityID)
OVER(ORDER BY salesYTD ASC) AS FirstValue
FROM salesperson;
```

entityID	salesYTD	FirstValue
267	559697	267
270	3189420	267
268	3763180	267
269	4251370	267

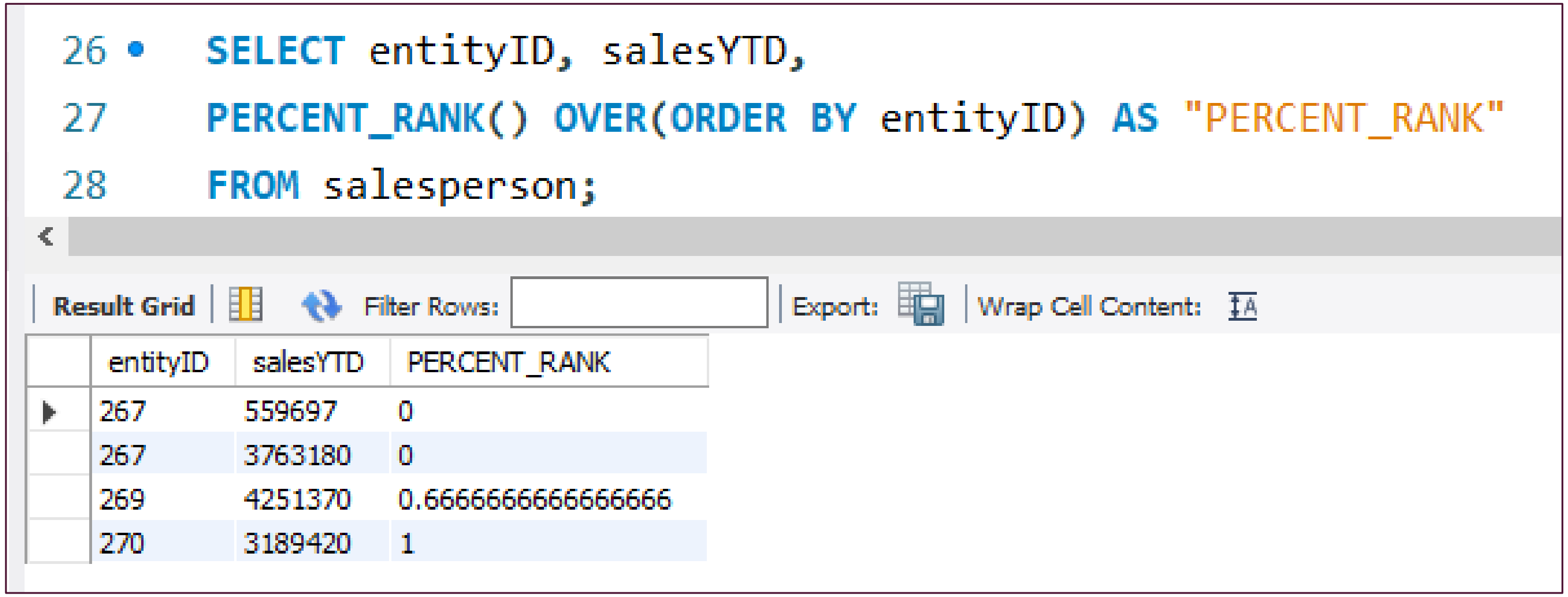
LAST_VALUE



```sql
SELECT entityID, salesYTD,
LAST_VALUE(salesYTD)
OVER(ORDER BY entityID ) AS LastValue
FROM salesperson;
```

entityID	salesYTD	LastValue
267	559697	3763180
267	3763180	3763180
269	4251370	4251370
270	3189420	3189420

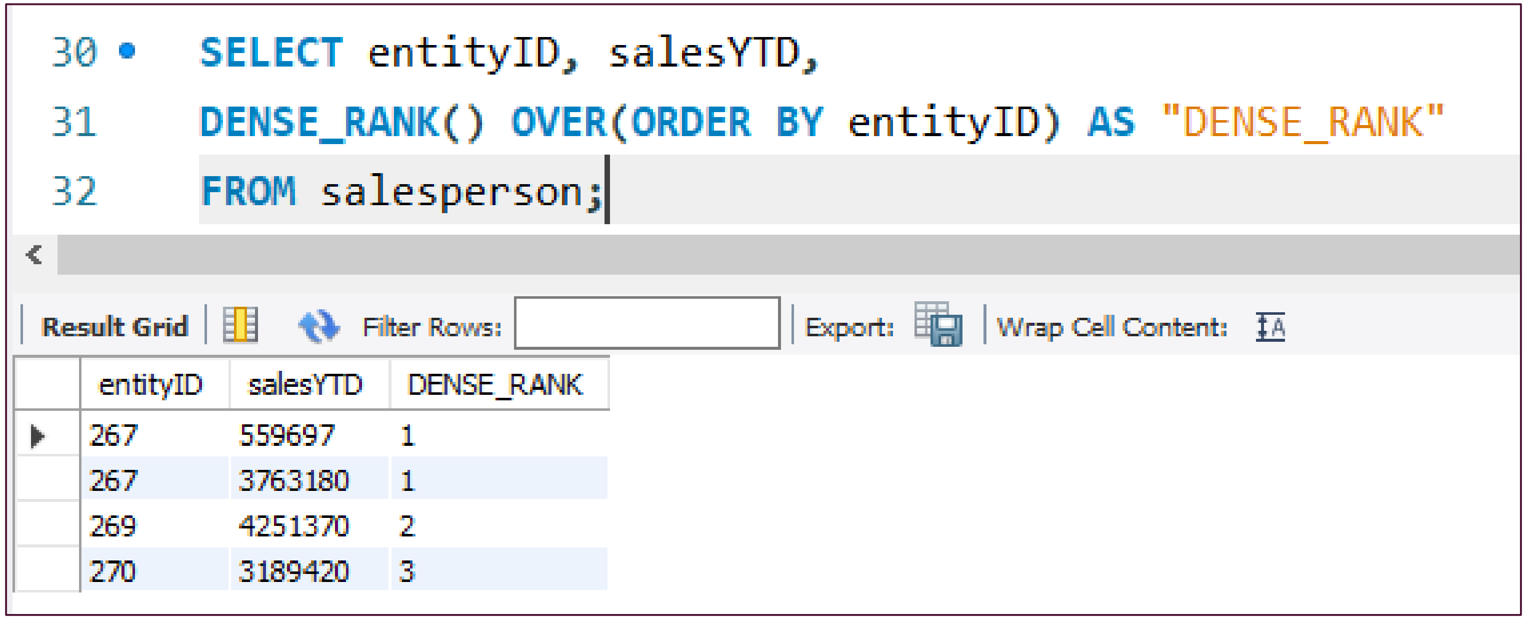
PERCENT_RANK



```
SELECT entityID, salesYTD,
PERCENT_RANK() OVER(ORDER BY entityID) AS "PERCENT_RANK"
FROM salesperson;
```

entityID	salesYTD	PERCENT_RANK
267	559697	0
267	3763180	0
269	4251370	0.6666666666666666
270	3189420	1

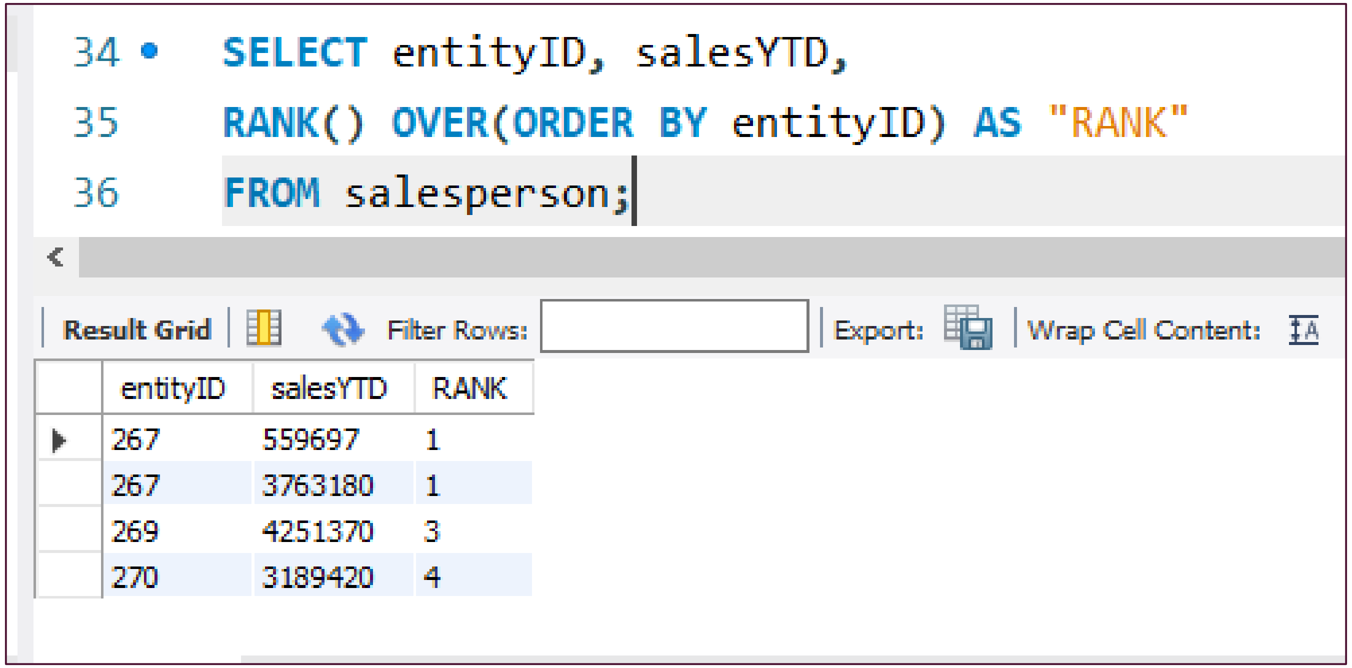
DENSE_RANK



```sql
SELECT entityID, salesYTD,
DENSE_RANK() OVER(ORDER BY entityID) AS "DENSE_RANK"
FROM salesperson;
```

entityID	salesYTD	DENSE_RANK
267	559697	1
267	3763180	1
269	4251370	2
270	3189420	3

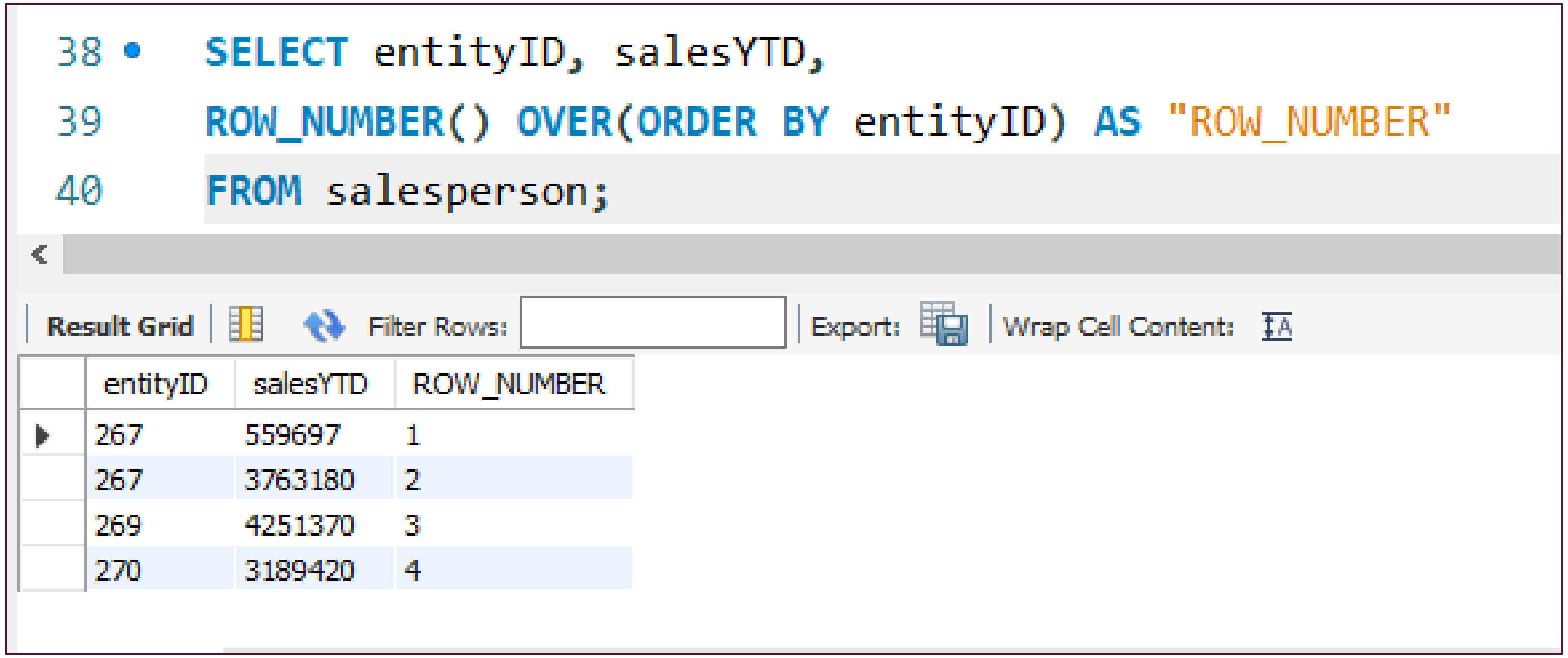
RANK



```sql
SELECT entityID, salesYTD,
RANK() OVER(ORDER BY entityID) AS "RANK"
FROM salesperson;
```

entityID	salesYTD	RANK
267	559697	1
267	3763180	1
269	4251370	3
270	3189420	4

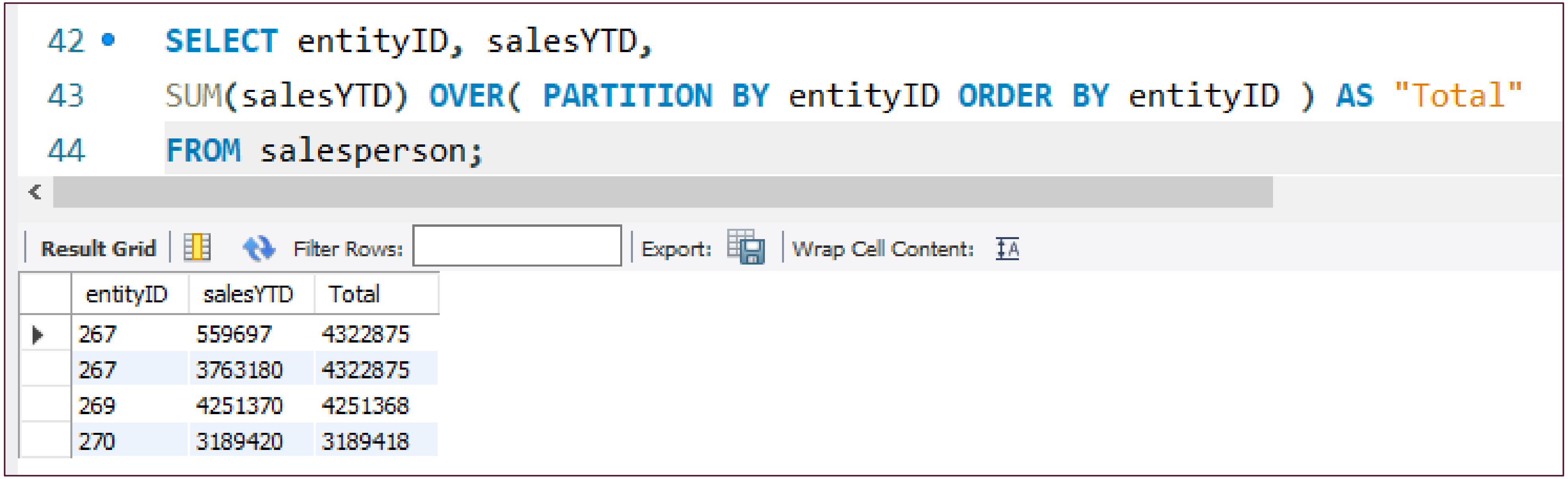
ROW_NUMBER



```
SELECT entityID, salesYTD,
ROW_NUMBER() OVER(ORDER BY entityID) AS "ROW_NUMBER"
FROM salesperson;
```

entityID	salesYTD	ROW_NUMBER
267	559697	1
267	3763180	2
269	4251370	3
270	3189420	4

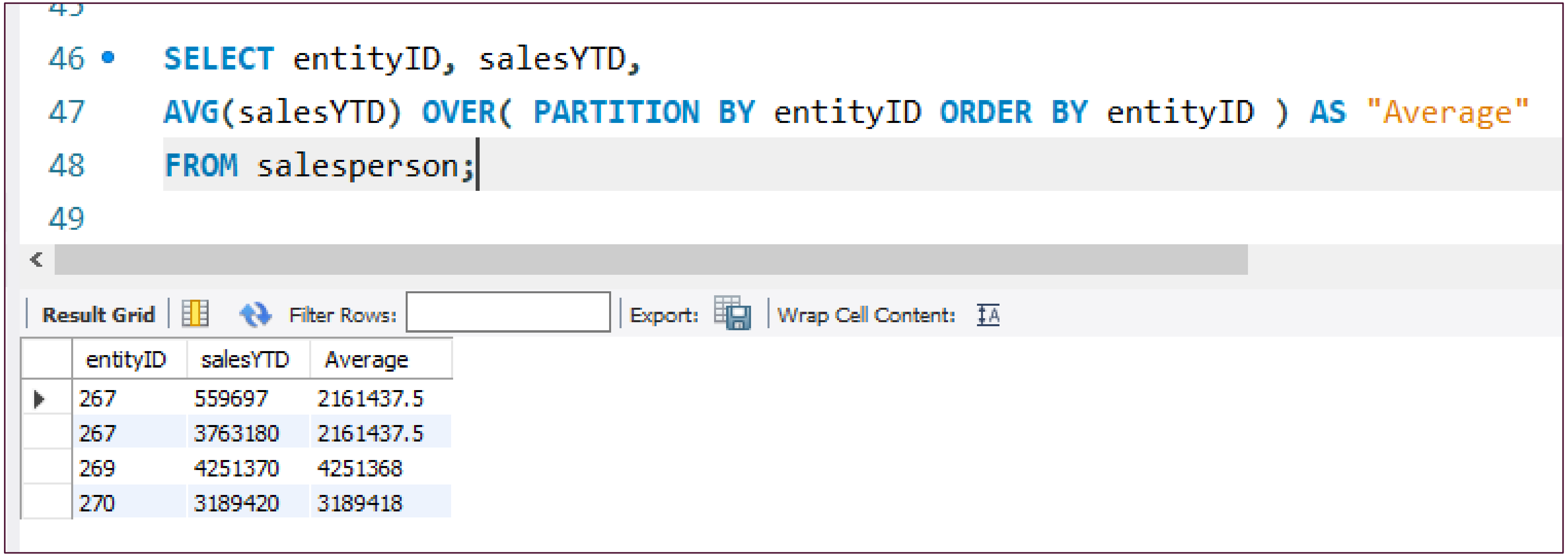
SUM



```sql
SELECT entityID, salesYTD,
SUM(salesYTD) OVER( PARTITION BY entityID ORDER BY entityID ) AS "Total"
FROM salesperson;
```

entityID	salesYTD	Total
267	559697	4322875
267	3763180	4322875
269	4251370	4251368
270	3189420	3189418

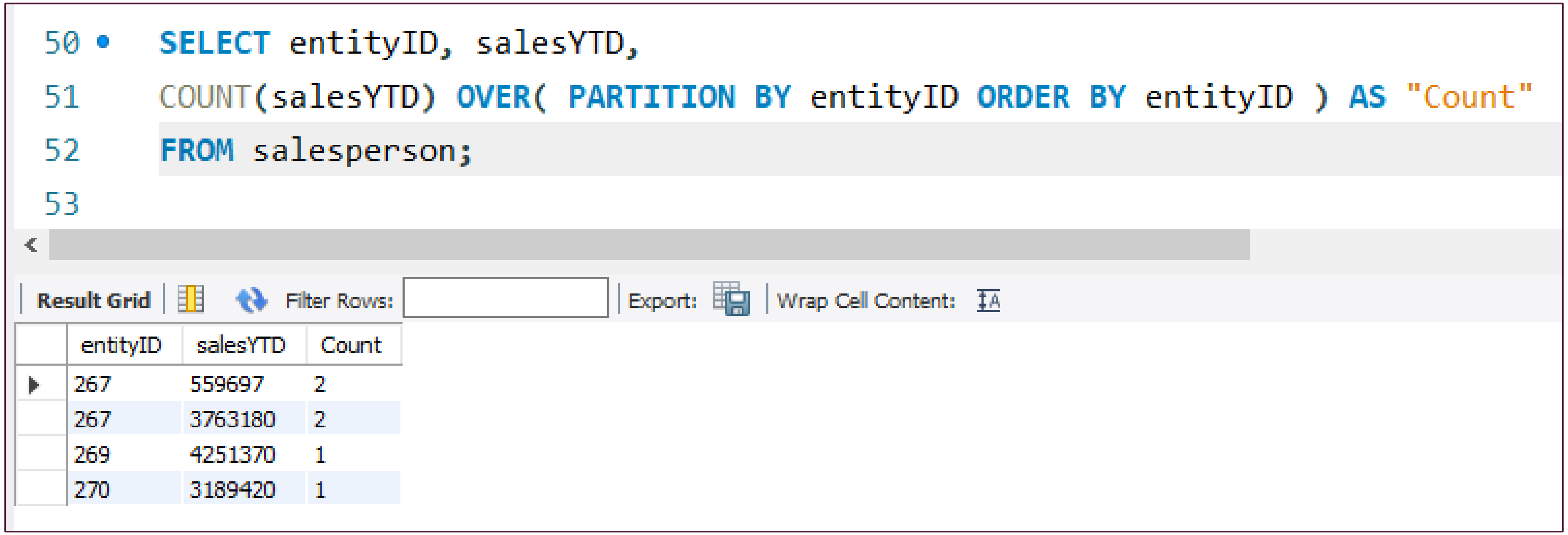
AVERAGE



```sql
SELECT entityID, salesYTD,
AVG(salesYTD) OVER( PARTITION BY entityID ORDER BY entityID ) AS "Average"
FROM salesperson;
```

entityID	salesYTD	Average
267	559697	2161437.5
267	3763180	2161437.5
269	4251370	4251368
270	3189420	3189418

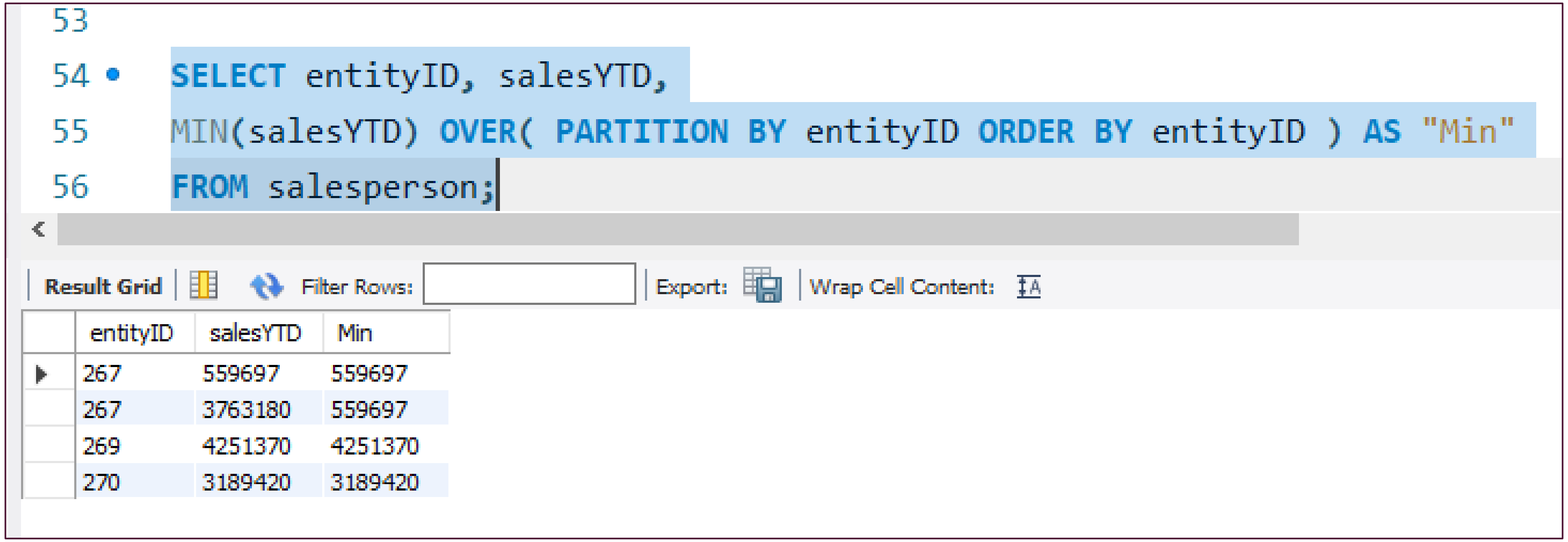
COUNT



```
SELECT entityID, salesYTD,
COUNT(salesYTD) OVER( PARTITION BY entityID ORDER BY entityID ) AS "Count"
FROM salesperson;
```

entityID	salesYTD	Count
267	559697	2
267	3763180	2
269	4251370	1
270	3189420	1

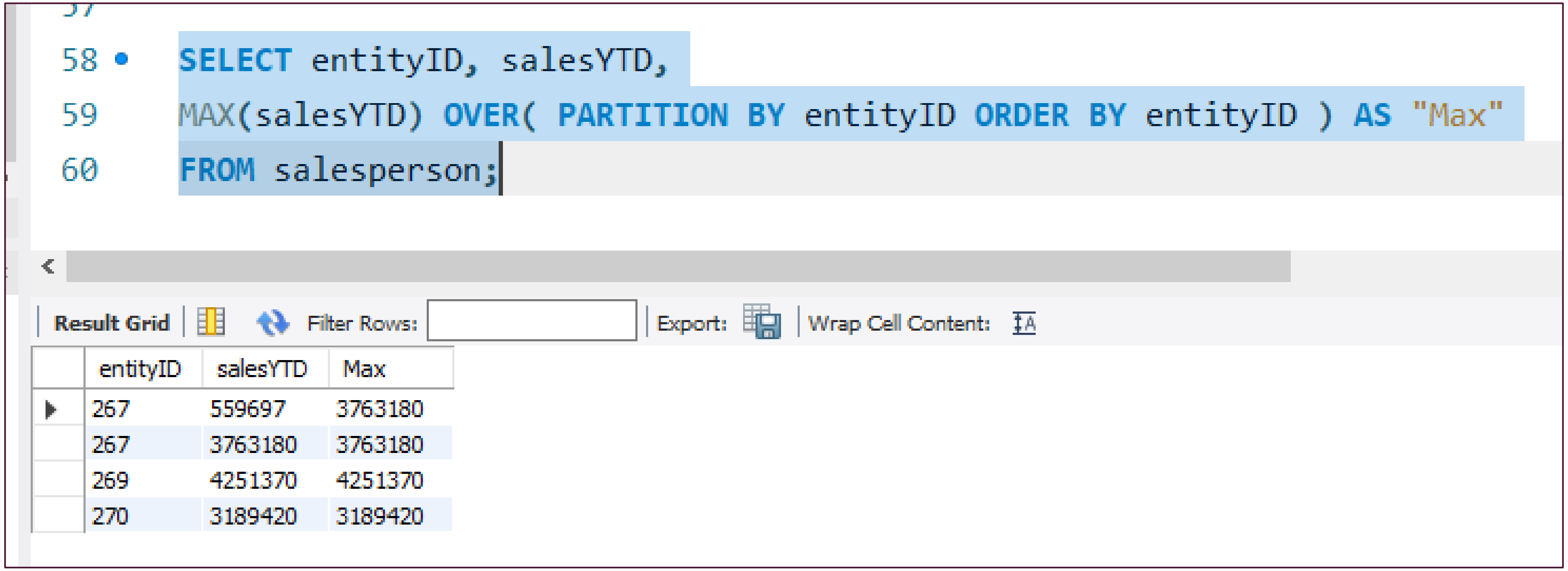
MIN



```
SELECT entityID, salesYTD,
MIN(salesYTD) OVER( PARTITION BY entityID ORDER BY entityID ) AS "Min"
FROM salesperson;
```

entityID	salesYTD	Min
267	559697	559697
267	3763180	559697
269	4251370	4251370
270	3189420	3189420

MAX



```
SELECT entityID, salesYTD,
MAX(salesYTD) OVER( PARTITION BY entityID ORDER BY entityID ) AS "Max"
FROM salesperson;
```

entityID	salesYTD	Max
267	559697	3763180
267	3763180	3763180
269	4251370	4251370
270	3189420	3189420

CASE EXPRESSION IN SQL

SQL Series Part 12

-Mayuri Dandekar

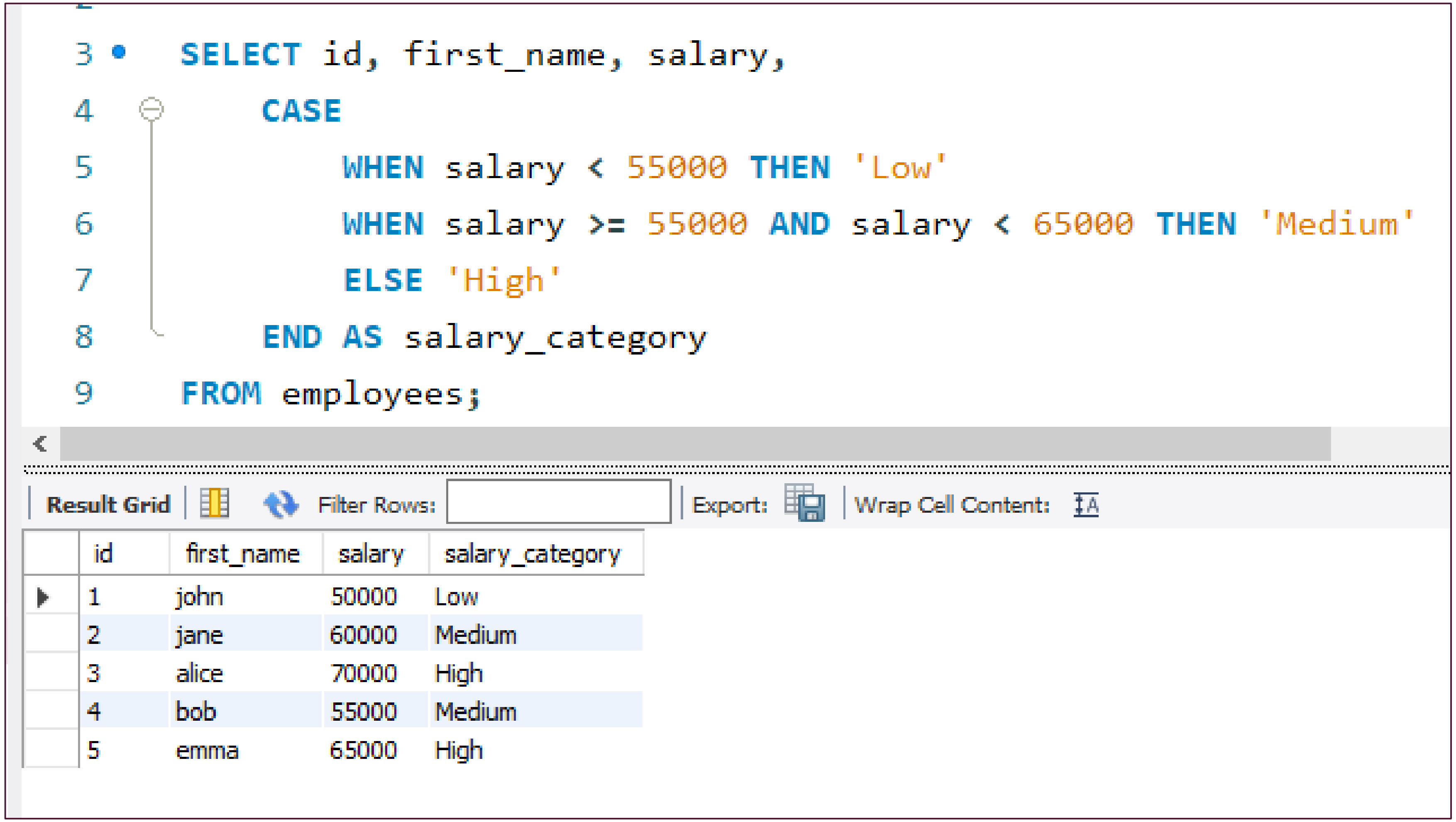
CASE EXPRESSION

In SQL, the `CASE` expression is a powerful tool that allows you to perform conditional logic within a query. It evaluates a list of conditions and returns one result based on the first condition that is true, similar to the `IF-THEN-ELSE` logic in programming languages.

The `CASE` expression can be used in SELECT, WHERE, ORDER BY, and GROUP BY clauses.

Syntax—

```
CASE Expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    WHEN valueN THEN resultN
    ELSE other_result
END;
```



```sql
SELECT id, first_name, salary,
    CASE
        WHEN salary < 55000 THEN 'Low'
        WHEN salary >= 55000 AND salary < 65000 THEN 'Medium'
        ELSE 'High'
    END AS salary_category
FROM employees;
```

id	first_name	salary	salary_category
1	john	50000	Low
2	jane	60000	Medium
3	alice	70000	High
4	bob	55000	Medium
5	emma	65000	High

COMMON TABLE EXPRESSION (CTE) IN SQL

SQL Series Part 13

-Mayuri Dandekar

COMMON TABLE EXPRESSION

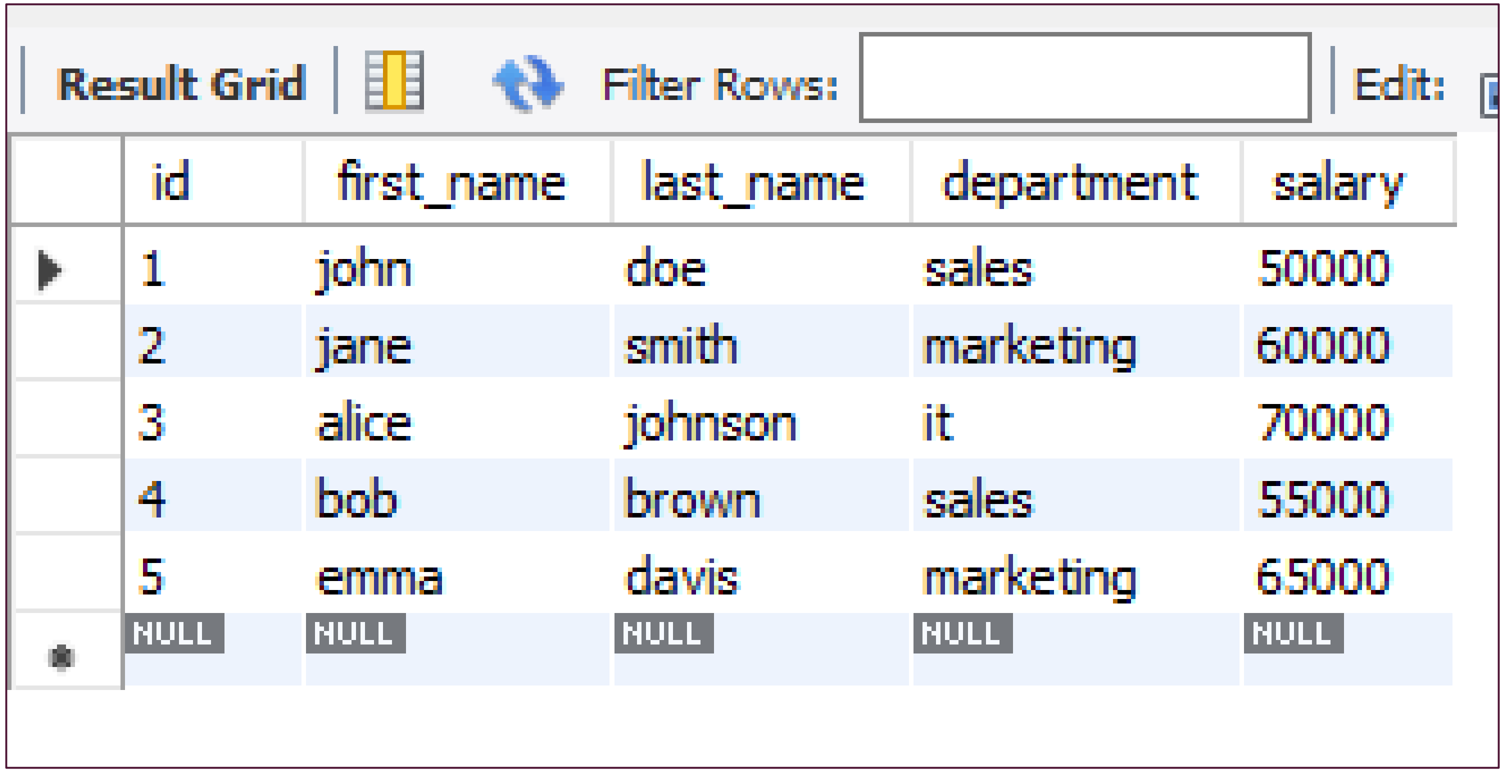
Common Table Expressions (CTEs) are temporary result sets that are defined within the scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement in SQL.

They provide a way to write more **readable and maintainable** queries by breaking them down into smaller, named, and reusable parts.

Syntax—

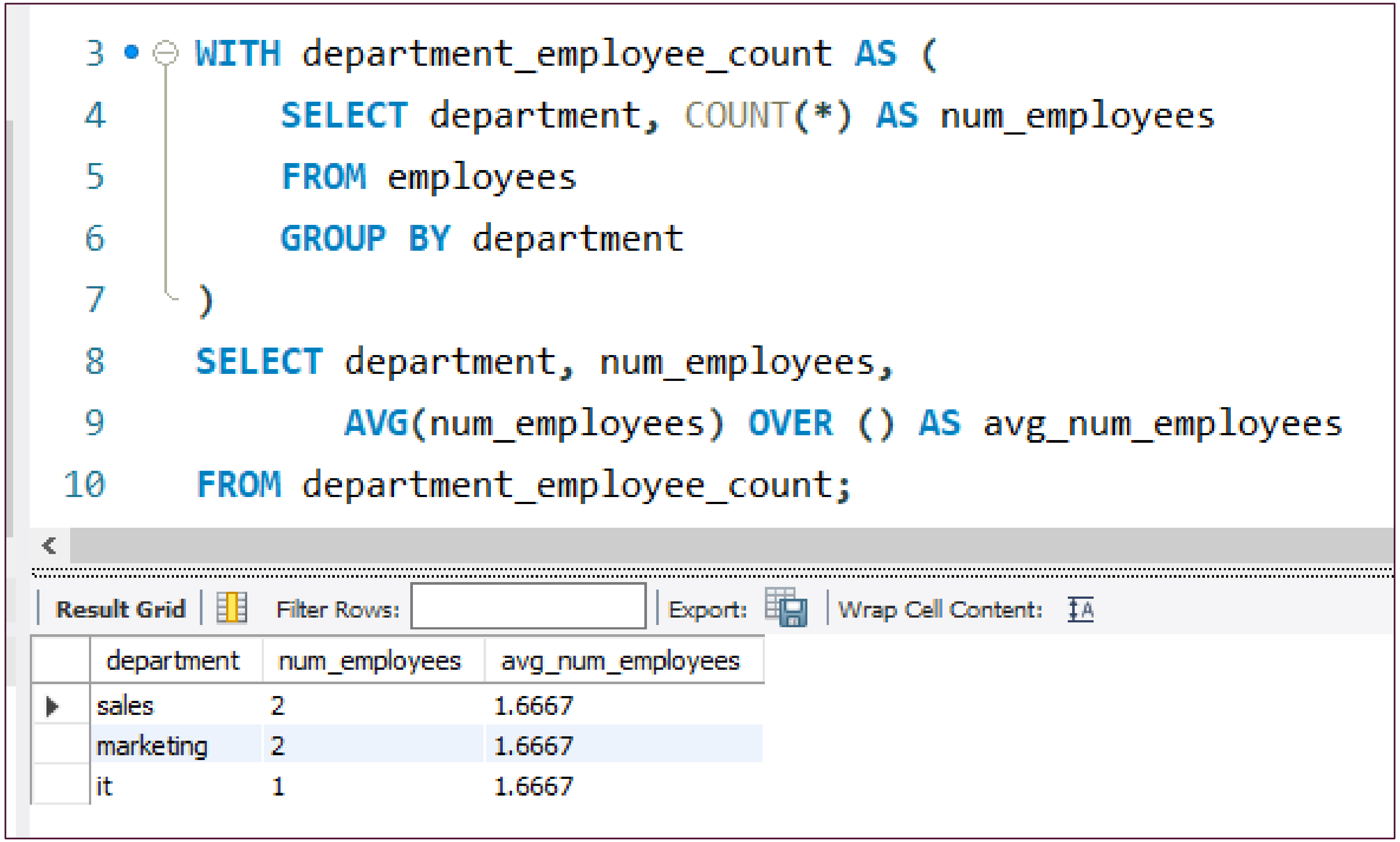
```
WITH my_cte AS (
    SELECT a,b,c
    FROM Table1 )
SELECT a,c
FROM my_cte
```

SA,MPLE DATASET



id	first_name	last_name	department	salary
1	john	doe	sales	50000
2	jane	smith	marketing	60000
3	alice	johnson	it	70000
4	bob	brown	sales	55000
5	emma	davis	marketing	65000
NULL	NULL	NULL	NULL	NULL

Find the average number of employees per department



```sql
WITH department_employee_count AS (
    SELECT department, COUNT(*) AS num_employees
    FROM employees
    GROUP BY department
)
SELECT department, num_employees,
       AVG(num_employees) OVER () AS avg_num_employees
FROM department_employee_count;
```

department	num_employees	avg_num_employees
sales	2	1.6667
marketing	2	1.6667
it	1	1.6667

THANK YOU!!

-Mayuri Dandekar