

SQL

- [SQL](#)
- [Basic](#)
- [Database Normalization](#)
- [Subsets of SQL commands](#)
 - [DDL](#)
 - [DML](#)
 - [DCL](#)
 - [TCL](#)
- [Datatypes](#)
- [Statements](#)
 - [SELECT](#)
 - [SELECT functions](#)
 - [Clauses](#)
 - [Aggregate functions](#)
 - [examples of some statements](#)
- [Constraints](#)
- [Keys](#)
 - [Unique key](#)
 - [Primary key](#)
 - [Composite primary key](#)
 - [Foreign key](#)
- [Trigger](#)
- [Subquery](#)
- [CTE](#)
- [Operators](#)
 - [Logical operators](#)
 - [Comparison operators](#)
 - [Arithmetic operations](#)
 - [Concatenation](#)
- [Column operations](#)
- [IF conditions](#)
- [Where](#)
 - [REGEX](#)
- [Dates](#)
- [Union](#)
- [Joins](#)
 - [Inner joins](#)
 - [Left \(outer\) join](#)
 - [Right \(outer\) join](#)
 - [Full \(outer\) join](#)
 - [Multi-table joins](#)
 - [Self join](#)
- [CASE WHEN](#)

- [Pivot](#)
 - [Wide -> long](#)
 - [Long -> wide](#)
- [Export query to CSV](#)
- [Procedures](#)
- [Views](#)
- [Transaction](#)
- [Isolation levels](#)
- [Denormalisation](#)
- [Relationships](#)
 - [One-to-one](#)
 - [One-to-many](#)
 - [Many-to-many](#)
 - [Many-to-one](#)
 - [Self-referencing](#)
- [PostgreSQL](#)

Table of contents generated with [markdown-toc](#)

Basic

Table - organised set of data in the form of rows and columns.

A few types of schema in a relational database:

- [Star schema](#):
- [Snowflake schema](#)

Basic commands:

Command	Function
<code>\! cd</code>	list current dir
<code>\! dir</code>	list files in the current dir
<code>\i file.sql</code>	import file
<code>\?</code>	print methods
<code>\l</code> or MySQL <code>SHOW DATABASES;</code>	list databases
<code>\c database_name</code> or MySQL <code>use database_name;</code>	connect to a database
<code>\d</code> or MySQL <code>SHOW TABLES;</code>	check which tables are present
<code>\dt</code>	show tables ONLY, without <code>id_seq</code>
PostgreSQL <code>\d second_table, \d+ second_table;</code> MySQL <code>DESCRIBE tablename</code>	check columns and details of a table in a database

Database Normalization

Database normalization is the process of **structuring a relational database** in accordance with a series of so-called **normal forms** in order to reduce data redundancy and improve data integrity. It was first proposed by British computer scientist Edgar F. Codd as part of his relational model.

Denormalized dataset - all the data is combined in one dataset, without adhering to the database rules. To enter to a database, data has to have data integrity and adhere to some rules of good database design.

Normalization of a database table - structuring it in such a way that it doesn't and cannot express *redundant information*.

The main purpose of database normalization is to avoid complexities, eliminate duplicates, and organize data in a consistent way.

There are some normal forms (NF) which start with the most important (dangerous) at 1NF and continue to the less dangerous ones with increasing number. These are basically like safety assessment levels, starting from broader one to the more detailed ones.

1NF - eliminates repeating groups:

- Row order should NOT be used to convey information in a table;
- Atomicity: a single cell can only contain a single value of the same data type (within the column);
- Every table has to have primary key (one or several);
- Every row should be unique and not be repeated;
- A repeating group of data items should NOT be stored on a single row; instead, should be stored in a separate table referencing the main table via key;

2NF - eliminates redundancy:

- The table has to be in 1NF;
- All non-key attribute in a table must depend on the entire primary key (one or several) within that table; if it only depends on one of the primary keys, then it doesn't belong in this table;
- Relationship between tables has to be formed with foreign keys;

3NF - eliminates transitive partial dependency:

- The table has to be in 2NF;
- every non-attribute in a table should depend on the key, the whole key, and nothing but the key; that is to say, there should not be dependencies between attributes that are not part of primary key;

An excellent example is given here: <https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/#:~:text=The%20First%20Normal%20Form%20%E2%80%93%201NF,-For%20a%20table&text=there%20must%20be%20a%20primary,each%20row%20in%20the%20table>

Subsets of SQL commands

SQL is a hybrid language that contains 4 languages at once - DDL, DML, DCL, DQL

Read more: <https://www.scaler.com/topics/ddl-dml-dcl/>

DDL

Data Definition Language, shortly termed DDL, is a subset of SQL commands that define the structure or schema of the database; commands used to modify or alter the structure of the database.

Commands:

Command	Explanation
CREATE	Create a new database
ALTER	ALTER command alters the database structure by adding, deleting, and modifying columns of the already existing tables, like renaming and changing the data type and size of the columns.
DROP	The DROP command deletes the defined table with all the table data, associated indexes, constraints, triggers, and permission specifications.
TRUNCATE	Deletes all the data / rows and records from an existing table, including the allocated spaces for the records. Unlike the DROP command, it does not delete the table from the database. It works similarly to the DELETE statement without a WHERE clause; also TRUNCATE is faster than DELETE.
RENAME	...

Database commands:

```
CREATE DATABASE database1;

-- Rename a database:
ALTER DATABASE first_database RENAME TO second_database;

-- Delete a database:
DROP DATABASE second_database;
```

Table commands:

```
-- General form
CREATE TABLE table1(
column1 DATATYPE CONSTRAINTS,
column2 DATATYPE CONSTRAINTS);
-- Create a new table
CREATE TABLE IF NOT EXISTS tablename;
-- Create an empty table
CREATE TABLE table1();
-- Some examples
CREATE TABLE table1(id SERIAL PRIMARY KEY, first_name VARCHAR(50) NOT NULL, gender VARCHAR(7) NOT NULL, date_birth DATE NOT NULL);
```

```

CREATE TABLE table1(id BIGSERIAL NOT NULL PRIMARY KEY);

-- Rename a table
ALTER TABLE table1
RENAME TO table2;

-- Delete records from a table, but leave the table
-- TRUNCATE - like DELETE, but doesn't have a possible IF clause
TRUNCATE table1;
TRUNCATE table1, table2;

-- Delete the table and all the rows inside it
DROP TABLE table1;
DROP TABLE IF EXISTS table1;

-- General form
ALTER TABLE table1
ADD COLUMN column1 DATATYPE CONSTRAINTS DEFAULT 'default',
ADD COLUMN column2 DATATYPE CONSTRAINTS REFERENCES table2(column1);
-- Add a column example
ALTER TABLE table1
ADD COLUMN name VARCHAR(30) NOT NULL UNIQUE;

-- Rename a column
ALTER TABLE table1
RENAME COLUMN column1 TO column2;

-- Change datatype of a column
ALTER TABLE characters ALTER COLUMN date_of_birth SET DATA TYPE VARCHAR(10); #
Change datatype of a column
-- Restart the auto-incrementing values
ALTER SEQUENCE person_id_seq RESTART WITH 10; # or 1
-- Add foreign key
ALTER TABLE <table_name> ADD FOREIGN KEY(<column_name>) REFERENCES
<referenced_table_name>(<referenced_column_name>);

-- Delete a column
ALTER TABLE table1
DROP COLUMN column1;

-- Drop a constraint for a column
ALTER TABLE table1 DROP CONSTRAINT constraint_name; # Drop a named constraint
ALTER TABLE table1 ALTER COLUMN column1 DROP NOT NULL; # Drop not null constraint

-- Add a column by concatenating two other columns (NOTE: this is not the most
optimal solution, but it's the one that works for me):
ALTER TABLE table1 ADD COLUMN full_name VARCHAR(30);
UPDATE table1 SET full_name = first_name || ' ' || last_name;

```

DML

Data Manipulation Language, shortly termed DML, is an element in SQL language that deals with managing and manipulating data in the database. DML commands are SQL commands that perform operations like storing data in database tables, modifying and deleting existing rows, retrieving data, or updating data.

Commands:

Command	Explanation
SELECT	Fetches data or records from one or more tables in the SQL database. The retrieved data gets displayed in a result table known as the result set.
INSERT	Inserts one or more new records into the table in the SQL database.
UPDATE	Updates or changes the existing data or records in a table in the SQL database.
DELETE	Deletes the existing records (that can be specified with a WHERE clause and logical operators to delete selected rows from the database). Is redo-able.
MERGE	Deals with insertion, updation, and deletion in the same SQL statement.
CALL	Calls or invokes a stored procedure.
EXPLAIN PLAN	Describes the access path to the data. It returns the execution plans for the statements like INSERT, UPDATE, and DELETE in the readable format for users to check the SQL Queries.
LOCK TABLE	Ensures the consistency, atomicity, and durability of database transactions like reading and writing operations.

Table:

```
# Delete records from a table, but leave the table
# DELETE - has a possible IF clause
DELETE FROM table1;
DELETE FROM table1 WHERE column1 = value;
```

```
-- Insert a row in the default order of columns
INSERT INTO table1 VALUES ('Value1', 52, DATE '1995-05-04');
-- Insert a row with data for specified columns only
INSERT INTO table1 (column1, column2, column3) VALUES ('Value1', 52, DATE '1995-05-04');
-- Insert two rows
INSERT INTO table1 (column1, column2, column3) VALUES (...), (...);

-- Alter all rows
UPDATE table1
SET column1 = 10

-- Update an entry based on IF-condition
UPDATE table1
SET column1=5, column2=10
WHERE row='Rowname' AND row2='Rowname2';
```

```
-- Delete all rows
DELETE FROM table1;
-- Delete a row in which column has the specified value
DELETE FROM table1 WHERE column1='Value';

## Update rows

# Update values in a column - swap 'f' and 'm' values
UPDATE Salary SET sex = CASE WHEN sex = 'm' THEN 'f' ELSE 'm' END;
```

DCL

Data Control Language, shortly termed DCL, is comprised of those commands in SQL that deal with controls, rights, and permission in the database system. DCL commands are SQL commands that perform operations like giving and withdrawing database access from the user.

Command	Explanation
GRANT	Gives access privileges or permissions like ALL, SELECT, and EXECUTE to the database objects like views, tables, etc, in SQL.
REVOKE	Withdraws access privileges or permissions given with the GRANT command.

TCL

Transaction Control Language:

- COMMIT
- ROLLBACK
- SAVEPOINT

Datatypes

Datatype	Description
DATE	YYYY-MM-DD
TIMESTAMP	YYYY-MM-DD HH:MM:SS
INT	Whole number.
SERIAL	Auto-increments a number. The SERIAL type will make your column an INT with a NOT NULL constraint, and automatically increment the integer when a new row is added.
BIGSERIAL	Auto-increments a number
CHAR(30)	String (specified length). The string has to be EXACTLY the specified length, in this case, 30 characters - no more, no less. <i>Note: use single quotes, not doublequotes.</i> If you need to use a single apostrophe as part of the string, use it two times to escape: 'O' 'Brien'

Datatype	Description
VARCHAR(30)	String (max length). The string can have a length up to the specified limit, such as 10, 20, 25 characters, but no more than 30 characters. <i>Note: use single quotes, not doublequotes</i>
NUMERIC(4, 1)	Float with number of decimals (1). For MySQL, I think, it's DECIMAL(10, 4)
NULL	Null. <code>column IS NULL</code>
BOOLEAN	TRUE, FALSE

Statements

SELECT

```
-- General syntax
SELECT column1 AS "column title", column2 AS aliasHere, ..., columnN
-- or `*` to select the rows from all the columns in a table
-- or `table1.column1, table1.column2` to specify which table, especially useful
in joins
FROM table1
WHERE column2 = 'Value' -- allows us to specify a condition by using an operator
AND (column3 = 'Value2' OR column3 > 100);

-- we can also give aliases to the tables
SELECT o.OrderId, o.OrderDate, c.CustomerId, c.FirstName, c.LastName, c.Country
FROM Orders o
RIGHT JOIN Customers c
ON o.CustomerId = c.CustomerId
```

SELECT functions

These functions are used with SELECT:

```
-- Only print unique values from the column
SELECT DISTINCT(column1)
SELECT DISTINCT column1

-- COALESCE - print a value for NULL values
SELECT COALESCE(column1, 'Entry not found') FROM table1;

-- UPPER
SELECT UPPER(name)
-- Capitalise the first letter only
SELECT CONCAT(
    UPPER(SUBSTRING(name, 1, 1)),
    LOWER(SUBSTRING(name, 2, LENGTH(name) - 1))
) AS name
```



```
SELECT ROUND(AVG(column1))
```

Clauses

WHERE

The WHERE clause is used in a SELECT statement to filter rows based on specified conditions before the data is grouped or aggregated. It operates on individual rows and filters them based on the given conditions.

ORDER BY

```
-- General form
SELECT column1, column2, ..., columnN
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ... columnN [ASC|DESC];

-- Examples
SELECT *
FROM employees
ORDER BY salary DESC, age DESC;
```

Aggregate functions

Note: aggregate functions such as AVG, MIN, and MAX cannot be used in a WHERE clause directly - they have to be wrapped in a subquery.

```
-- General form
SELECT column1, column2, columnN aggregate_function(columnX)
FROM table
GROUP BY columns(s);

SELECT column1, aggregate_function(column2) AS alias

-- COUNT
-- Count the total number of rows
SELECT COUNT(*)
-- Count non-null values in a column
SELECT COUNT(column_name)
-- Count all female employees
SELECT COUNT(emp_id) FROM employee WHERE sex = 'F';
-- Count how many entries for each unique group in column 'sex' there are
SELECT COUNT(sex), sex FROM employee GROUP BY sex;
-- Count unique categories in a column
SELECT COUNT(DISTINCT sex) FROM employee;
SELECT COUNT(DISTINCT(sex)) FROM employee;

-- SUM
```

```
-- Sum all values in a column
SELECT SUM(column1)

-- Find the total sales of each salesman
SELECT SUM(total_sales), emp_id FROM works_with GROUP BY emp_id;

-- MIN, MAX
-- Print the max value of column2
SELECT MAX(column1)

-- Select the earliest date for each 'player_id' category
SELECT player_id, MIN(event_date) AS first_login
FROM Activity
GROUP BY player_id

-- AVG
SELECT AVG(column1)

-- GROUP BY
-- Find out the total salary paid out by each department
SELECT department_id, SUM(salary) as total_salary
FROM employees
GROUP BY department_id;

-- Group by can be used with joins:
SELECT u.name as NAME, SUM(t.amount) as BALANCE
FROM Users u
INNER JOIN Transactions t
ON u.account = t.account
GROUP BY u.name
HAVING SUM(t.amount) > 10000

-- HAVING clause
-- The HAVING clause was added to SQL to filter the results of the GROUP BY clause
since WHERE does not work with aggregated results. The syntax for the HAVING
clause is as follows:
-- The HAVING clause is used in combination with the GROUP BY clause in a SELECT
statement to filter rows based on specified conditions after the data is grouped
and aggregated. It operates on the result of the grouping operation and filters
the aggregated data.
SELECT column1, aggregate_function(column2)
FROM table
GROUP BY column1
HAVING aggregated_condition;

-- Find out which departments have a total salary payout greater than 50,000
SELECT department_id, SUM(salary) as total_salary
FROM employees
GROUP BY department_id
HAVING SUM(salary) > 50000;

-- STRING_AGG to concatenate strings
-- Below, the ORDER BY within the agg function is optional - it's just to sort the
concatenated names lexicographically within each concatenation group
SELECT id, STRING_AGG(name, ', ' ORDER BY name) AS names
FROM some_table
```

```
GROUP BY id
```

examples of some statements

SQL functions:

WHERE:

```
WHERE column1 != 2 OR column2 IS null;
WHERE column1 IN ('Value1', 'Value2', 'Value3')
-- Values between two dates
WHERE date BETWEEN DATE '1999-01-01' AND '2015-01-01'
-- Values alphabetically between two strings
WHERE column1 BETWEEN 'Alpha' AND 'Beta'
-- Odd number
MOD(columnName, 2) <> 0
-- Even number
MOD(columnName, 2) = 0

-- REGEXP
-- Value starts with 'a'
WHERE email LIKE 'a%'
-- Value ends with '.com'
WHERE email LIKE '%.com';

WHERE course NOT LIKE '_lgorithms';
-- case-insensitive
ILIKE, NOT ILIKE
-- Entries start with a vowel
SELECT DISTINCT(CITY) FROM STATION WHERE CITY ~ '^[AEIOUaeiou].*';
SELECT DISTINCT(CITY) FROM STATION WHERE CITY REGEXP '^[aeiou]';
```

HAVING

WHERE is used for filtering rows BEFORE any grouping or aggregation.

HAVING is used for filtering rows AFTER any grouping or aggregation.

If you have both a WHERE clause and a HAVING clause in your query, WHERE will execute first.

In order to use HAVING, you also need:

- A GROUP BY clause
- An aggregation in your SELECT section (SUM, MIN, MAX, etc.)

GROUP BY:

```
-- Count number of repetitions of unique categories in column `column1`
SELECT branch_id, COUNT(*) FROM branch_supplier GROUP BY branch_id
-- Count number of repetitions of unique categories in column `column1` where
count is greater than 3
SELECT branch_id, COUNT(*) FROM branch_supplier GROUP BY branch_id HAVING COUNT(*)
> 3;

-- Count how many unique categories each super_id has
SELECT super_id, COUNT(DISTINCT(emp_id)) FROM employee GROUP BY super_id;
```

- GROUP BY column1
- GROUP BY column1 HAVING COUNT(*) > 5 only group those values whose count is > 5
- select major_id, count(*) from students group by major_id; count unique values in column 'major_id'
- select major_id, min(gpa) from students group by major_id; view min value in each group within column major_id

OFFSET: skip n rows

LIMIT: show n first rows

Examples:

- SELECT * FROM table1; view table1
- SELECT * FROM characters ORDER BY character_id DESC; view the whole table ordered by 'character_id'; DESC or ASC
- SELECT * FROM person WHERE gender='Female' AND (country_of_birth='Poland' OR country_of_birth='China') ORDER BY first_name;
- SELECT column1, COUNT(*) FROM table GROUP BY column1; print count of each value in column1
- SELECT make, SUM(price) FROM car GROUP BY make;
- WHERE column1 < 'M' selects rows with column1 values before 'M' alphabetically

Constraints

Constraints are used to limit the data types for specific columns.

Constraint	Meaning
NOT NULL	Values in this column have to be present, i.e. cannot be NULL
CHECK	Check for a specified condition. E.g. constraint login_min_length check (char_length(login) >= 3) - check the minimum length of a login field.
DEFAULT	Sets a default value for each row in a column
PRIMARY KEY	Makes a specified column a PRIMARY KEY type.

Constraint	Meaning
FOREIGN KEY	Makes a specified column an external key. E.g. <code>constraint user_uuid_foreign_key foreign key (user_uuid) references users (uuid) on update cascade on delete cascade</code> - обязывает содержать значение в user_uuid только для существующей записи в таблице users и автоматически обновится если оно будет изменено в таблице users, а так же заставит запись удалиться при удалении записи о пользователе
REFERENCES table(column)	Make a foreign key referencing another table
BIGSERIAL	Integer that auto-increments
BOOLEAN	True / False, 'Yes' / 'No'
UNIQUE	Values in this column must be unique for each data point

Examples:

```
-- Add a NOT NULL constraint to the foreign key column, so that there will be no
Null rows
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

UNIQUE - makes sure that only unique values can be added in a column

```
ALTER TABLE table1 ADD CONSTRAINT constraint_name_here UNIQUE (column1) # Custom
constraint name
# or
ALTER TABLE table1 ADD UNIQUE (column1) # Constraint name defined by psql
```

DEFAULT - specify a default value for a column

```
CREATE TABLE table1 (column1 INT DEFAULT 'undecided')
```

CHECK - a column can only accept specific values

```
ALTER TABLE table1 ADD CONSTRAINT constraint_name CHECK (column1='Male' OR
column1='Female');
```

CONFLICT (CONSTRAINT) MANAGEMENT

```
ON CONFLICT (column1) DO NOTHING;
INSERT INTO ... VALUES ... ON CONFLICT (column1) DO UPDATE SET column1 =
```

```
EXCLUDED.column1; # If an entry exists, it will update with the value you give it
```

FOREIGN KEYS - can connect tables based on foreign keys

```
CREATE TABLE table1(column1 DATATYPE REFERENCES table2(column_of_table2);

ALTER TABLE table_name ADD COLUMN column_name DATATYPE REFERENCES
referenced_table_name(referenced_column_name); # to set a foreign key that
references a column from another table
ALTER TABLE table_name ADD FOREIGN KEY(column_name) REFERENCES
referenced_table(referenced_column); # set an existing column as a foreign key
ALTER TABLE character_actions ADD FOREIGN KEY(character_id) REFERENCES
characters(character_id);
```

Keys

Unique key

A unique key prevents duplicate values in a column and can store NULL values.

Primary key

Primary key:

- Serves as a **unique identifier** for each record in a table;
- IOW, it is an entry into the **Primary key** column that **inequivocally (uniquely)** identify each one row in a table, i.e. an ID for each data point / row.

Features:

- **Null** values are not accepted.
- Are indexed automatically.
- If you manually tried inserting a row with a primary key that already exists in the table, it would lead to an error, as no duplicate primary keys are allowed;
- By definition, primary key has two constraints - NOT NULL and UNIQUE;

```
-- NOTE: NOT A GOOD PRACTICE, but an example. Create a column with primary key
that you manually have to enter:
```

```
CREATE TABLE sounds (sound_id INT PRIMARY KEY);
```

```
-- You can create a table with a column with PRIMARY KEY constraint. As it is
SERIAL, you don't need to specify it when inserting new rows - it will be created
automatically as per the internal rules:
```

```
-- PostgreSQL
```

```
CREATE TABLE sounds (
  sound_id SERIAL PRIMARY KEY
);
```

```
-- MySQL
CREATE TABLE student (
    student_id INT AUTO_INCREMENT PRIMARY KEY
);
-- Or add a new column and set it up as a primary key
ALTER TABLE moon ADD COLUMN moon_id SERIAL PRIMARY KEY;

-- Adding a new column and setting it up as a primary key can also be done in two
steps:
ALTER TABLE table_name ADD COLUMN column1 SERIAL;
ALTER TABLE table1 ADD PRIMARY KEY (column1);
```

If you want to alter the primary key, you can do it like this. Check first the details of a table with the command `\d characters`:

```
mario_database=> \d characters
```

Column	Type	Collation	Nullable
character_id	integer		not null
name	character varying(30)		not null
homeland	character varying(60)		
favorite_color	character varying(30)		

```
Indexes:
    "characters_pkey" PRIMARY KEY, btree (name)
```

Then drop constraint:

```
ALTER TABLE characters DROP CONSTRAINT characters_pkey;
```

Other commands:

`ALTER TABLE table_name ADD PRIMARY KEY(column1, column2);` # create composite primary key (primary key from two columns)

`ALTER TABLE table1 DROP CONSTRAINT person_pkey` # Drop primary key constraint

Composite primary key

```
-- Uses more than one column as a unique pair.  
ALTER TABLE <table_name> ADD PRIMARY KEY(<column_name>, <column_name>);
```

Foreign key

A foreign key:

- Makes a connection between two tables via their joint column.
- A foreign key in one table usually references a primary key in another.
- Enforce data integrity, making sure the data confirms to some rules when it is added to the DB.

ON DELETE SET NULL: if in the table 1 a row is deleted, then in the table 2 that references that first table via foreign key the corresponding value is set to NULL;

ON DELETE CASCADE: if the row in the original table containing an id is deleted, then in a table referencing that table via a foreign key the entire row is deleted.

```
-- Create foreign key upon creation of the table  
CREATE TABLE user_profiles (  
  profile_id INT PRIMARY KEY,  
  user_id INT UNIQUE,  
  profile_data VARCHAR(255),  
  FOREIGN KEY (user_id) REFERENCES users(user_id) -- ON DELETE SET NULL --or--  
ON DELETE CASCADE  
  
);  
  
-- Create a new column with the constraint of foreign key  
ALTER TABLE more_info  
ADD COLUMN character_id INT  
REFERENCES characters(character_id);  
  
-- You can set an existing column as a foreign key like this:  
ALTER TABLE table_name  
ADD FOREIGN KEY(column_name)  
REFERENCES referenced_table(referenced_column)  
ON DELETE SET NULL -- optional option  
;
```

Trigger

Defines a certain action when a certain operation is performed on a database.


```
-- Run this in the MySQL terminal
DELIMITER $$
CREATE
    TRIGGER my_trigger BEFORE INSERT
    ON employee
    FOR EACH ROW BEGIN
        INSERT INTO trigger_test VALUES('added new employee'); --
        `VALUES(NEW.attribute_name)` if you want to add attribute of the newly-inserted
        row
    END$$
DELIMITER ;
-- Now, every time a row is added to the table `employee`, a row is added into the
table `trigger_test` saying `added new employee`
```

Subquery

Subquery, aka nested queries, inner queries:

- are queries that are embedded within the context of another SQL query.
- Are powerful tools for performing complex data manipulations that require one or more intermediary steps
- Types (depending on where / in which clause the subquery is located):
 - SELECT subqueries
 - FROM subqueries
 - WHERE subqueries
 - HAVING subqueries

SELECT subqueries

```
-- General Form
SELECT column1, column2, columnN,
(SELECT agg_function(column) FROM table WHERE condition)
FROM table
```

FROM subqueries

Subqueries in the FROM clause create a temporary table that can be used for the main query. This allows the programmer to simplify the process by breaking the problem into smaller, more manageable parts.

```
-- General form
SELECT employee, total_sales
FROM (SELECT first_name || ' ' || last_name as employee, SUM(sales) as
total_sales
FROM sales
GROUP BY employee) as sales_summary
WHERE total_sales > 100000;
```

In this example, the subquery creates a temporary table aliased as `sales_summary`, which does the following:

- Concatenates each employee's first and last name (separated by a space). This concatenation is aliased as `employee`.
- Calculates the total sales for each employee.
- Groups the `total_sales` by employee.

WHERE subqueries

Subqueries in the WHERE clause are used to filter rows based on conditions detailed in a subquery.

This method is useful when you don't already have access to the condition on which you want to filter your query.

Scalar example:

```
-- Suppose that we have a table called employees with employee_id, first_name,
last_name, salary, and department_id columns. If we want to find all employees who
earn more than the average salary, we can use a subquery:
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Non-scalar example:

```
-- Suppose that we are using the same dataset as before with the first_name,
last_name, and salary fields. We want to return the first name, last name, and
salary of employees whose first name begins with the letter 'J':
SELECT first_name, last_name, salary
FROM employees
WHERE salary > ANY (SELECT salary FROM employees WHERE first_name LIKE 'J%');
```

HAVING subqueries

The HAVING clause is used to filter the results of a GROUP BY query based on conditions involving aggregate functions. The subquery is executed for each group and filters the groups based on the specified condition.

```
SELECT CustomerID, AVG(TotalAmount) AS AverageTotalAmount
FROM Orders
GROUP BY CustomerID
HAVING AVG(TotalAmount) > (SELECT AVG(TotalAmount)
FROM Orders);
```

Using multiple SELECT statements, where the output of one query gets passed on to another query.

```
-- Find names of all employees who have sold over 30,000 to a single client
SELECT employee.first_name, employee.last_name
FROM employee
WHERE employee.emp_id IN (
    SELECT works_with.emp_id
    FROM works_with
    WHERE works_with.total_sales > 30000
);

-- Find all clients who are handled by the branch that Michael Scott manages
SELECT client.client_name
FROM client
WHERE branch_id = (
    SELECT employee.branch_id
    FROM employee
    WHERE employee.first_name = 'Michael' AND employee.last_name = 'Scott'
);
```

Use IDs from one table to use in querying another table

```
SELECT column1 as 'Column 1' FROM table1 WHERE table1.id NOT IN (SELECT
customer_id FROM table2);
```

CTE

CTE, common table expressions

CTEs are similar to subqueries.

CTEs are also temporary tables typically that are formulated at the beginning of a query and only exist during the execution of the query. This means that CTEs cannot be used in other queries beyond the one in which you are using the CTE. While CTEs and subqueries are both used in similar circumstances (such as when you need to produce an intermediary result), there are a couple of factors that tip off CTEs: • They are typically created at the beginning of a query using the WITH operator • They are followed by a query that queries the CTE Alternatively, subqueries are a query within a query, nested within one of a query's clauses.

```
WITH alias AS ( <Put query here>
)
... <Query that queries the alias>

-- A more concrete example
WITH customer_totals AS (
    SELECT CustomerID, SUM(TotalAmount) AS total_sales
    FROM Orders
    GROUP BY CustomerID
)
```

```
SELECT c.CustomerID, c.total_sales, o.avg_order_amount
FROM customer_totals c
JOIN (
  SELECT CustomerID, AVG(TotalAmount) AS avg_order_amount
  FROM Orders GROUP BY CustomerID )
ON c.CustomerID = o.CustomerID;
```

Operators

Operators are usually used with a WHERE statement.

Logical operators

Operator	Meaning
AND	Shows data if all the conditions separated by AND are TRUE.
OR	Shows data if any of the conditions separated by OR is TRUE.
NOT	Shows data if the condition after NOT is not true.
BETWEEN ... AND ...	Return values that are between the two values. WHERE salary BETWEEN 5000 AND 10000
IN	TRUE if the operand is equal to one of a list of expressions
LIKE	TRUE if the operand matches a pattern

Some examples:

```
SELECT column1, column2 FROM table1 WHERE condition1 AND condition2 AND
condition3;

WHERE NOT condition;
```

Comparison operators

Can be used for comparing numbers or strings.

Operator	Meaning
<, <=, >, >=	
=	equals
<>	not equal

Note: NULL value indicates an unavailable or unassigned value. The value NULL does not equal zero (0), nor does it equal a space (' '). Because the NULL value cannot be equal or unequal to any value, you cannot perform any comparison on this value by using operators such as '=' or '<>'.

Therefore, use `Column IS NULL` or `NOT NULL`

Arithmetic operations

```
SELECT 10 + 2;
ROUND(column1, decimalplaces);
```

Operator	Meaning
<code>-, +, *, /</code>	
<code>^</code>	power
<code>%</code>	modulo
<code>MIN(column1)</code>	minimum value of a column
<code>SUM(column1)</code>	sum of all values in a column
<code>AVG(column1)</code>	average of a column's values
<code>CEIL(5.9), FLOOR(5.1)</code>	round up a value
<code>ROUND(<number_to_round>, <decimals_places>)</code> or <code>round(15.51235312, 2)</code>	round a value to the nearest whole number
<code>COUNT(*)</code>	count number of rows
<code>mod(column1,2)</code>	Check the remainder of the division. In this case, remainder is zero if the number is even.

Examples:

- `SELECT column1 * 10`
- `SELECT MAX(column1)`

Concatenation

Concatenate two columns:

```
SELECT first_name || '-' || last_name AS column_name
FROM employee;
```

Column operations

```
SELECT col1 / col2 AS alias1
-- or
```

```
round( SUM(rating::dec / position::dec)::dec / COUNT(rating)::dec, 2) AS quality
```

IF conditions

```
# From table 'Employee', calculate bonus for each employee_id.
# Bonus = 100% salary (if ID is odd and employee name doesn't start with 'M'),
# else bonus = 0.
## Solution 1
SELECT employee_id, CASE WHEN employee_id % 2 = 1 AND name NOT LIKE 'M%' THEN
salary ELSE 0 END AS bonus FROM Employees;
## Solution 2
SELECT employee_id, if(employee_id % 2 = 1 AND name NOT LIKE 'M%', salary, 0) AS
bonus FROM Employees;
```

Where

REGEX

There are two ways of writing regular expressions in SQL:

- ~: True REGEXP
- LIKE: simplified REGEXP; is not as powerful, but typically faster than regular expressions.

LIKE

Sign	Meaning
%	any character, any number of times
_	exactly 1 character

These are used with the SQL keyword LIKE

```
SELECT * FROM courses WHERE course LIKE '_lgorithms';
-- Find any clients who are an LLC
SELECT * FROM client WHERE client_name LIKE '%LLC';
-- Find employees born in october
SELECT * FROM employee WHERE birth_date LIKE '____-10-%';

-- names starting with 'W'
LIKE 'W%'
-- the second letter is 'e'
LIKE '_e%'
-- values with a space in them
LIKE '% %'
```

REGEXP

```
SELECT * FROM table1 WHERE name ~ '^Grandfather.+|.+.parents.+'
```

Dates

```
-- Gives YYYY-MM-DD HH:MM:SS.MSMS
SELECT NOW();
-- Get years of a person from his birthday
SELECT AGE(NOW(), date_of_birth);
-- returns 2022-03-16
SELECT CURDATE(); select CURRENT_DATE
-- returns current date formatted as UNIX
select UNIX_TIMESTAMP()

-- return records where date equals to specified date
select * from personal_data where birthday = '1977-05-04'
select * from personal_data where birthday = '1977-05-04'::date;

-- INTERVAL: `YEARS`, `MONTHS`, `DAYS`
NOW() - INTERVAL '1 YEAR'; # Time a year ago
-- Select birthdays between 1977-05-04 and 30 days before that
SELECT * FROM personal_data WHERE birthday < '1977-05-04'::date AND birthday >
'1977-05-04'::date - INTERVAL '30 DAYS';
```

Typecasting: `::DATE`, `::TIME`

```
SELECT NOW()::DATE # YYYY-MM-DD
SELECT NOW()::TIME # HH:MM:SS.MSMS
(NOW()::DATE + INTERVAL '10 MONTHS')::DATE
```

Extracting fields: `DAY`, `DOW`, `MONTH`, `YEAR`, `CENTURY`

```
SELECT EXTRACT (YEAR FROM NOW());

-- Thus, we can order birthdays based only on month and date
-- For example, table like this:
-- id | name          | date      |
-- ----+-----+-----+
-- 21 | Person 1      | 1971-11-21 |
-- 23 | Person 2      | 1989-12-29 |

SELECT * FROM notable_dates ORDER BY EXTRACT(MONTH FROM date), EXTRACT(DAY FROM
```

```
date) DESC;

-- Another example
-- Select all rows where date is 2020
select * from notable_dates where extract (year from date) = 2020;
```

Select a part of a date:

- 'year', 'month', 'day', 'hour', 'minute', 'second'

```
SELECT date_part('year', (SELECT date_column_name))
```

Union

UNION combines the results from several SELECT statements.

Rule:

- You have to have the same number of columns in the two statements that are joined by the **UNION** statement
- The columns being concatenated have to have the same data type

```
-- Return a list of employee names and then branch names located below the first list
SELECT first_name -- can also specify the name of the common column, e.g. `AS name_of_the_union_column`
FROM employee
UNION
SELECT branch_name
FROM branch;

-- Find a list of all clients and branch suppliers ids
SELECT client_name, branch_id -- to increase clarity, can specify the table: `client.branch_id`
FROM client
UNION
SELECT supplier_name, branch_id -- same: `branch_supplier.branch_id`
FROM branch_supplier;
```

Joins

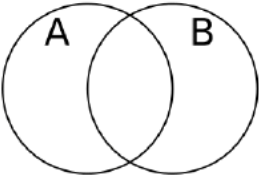
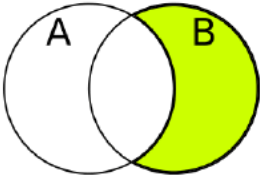
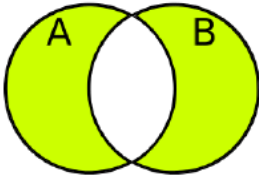
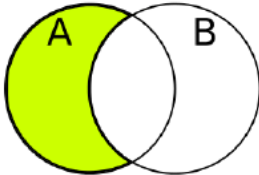
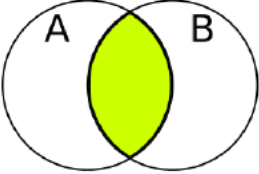
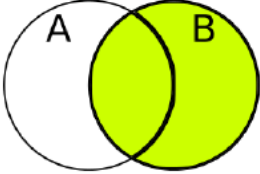
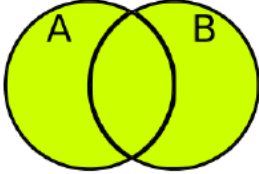
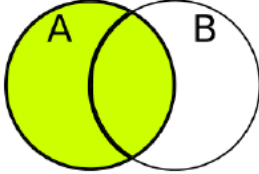
JOIN is a command for linking rows from two or more tables based on a column common for all of them.

There are two main categories of joins:

- **INNER JOIN**: will only retain the data from the two tables that is related to each other (that is present in both tables, like an overlap of the Venn diagram);
- **OUTER JOIN**: will additionally retain the data that is not related from one table to the other; iow, combines values from the two tables, even those with NULL values.

General form:

```
SELECT * FROM table1 -- or SELECT table1.id, table2.id2
JOIN table2 ON relation;
```

SQL JOINS			
No join (\emptyset)	[Exclusive] Right Join ($\sim A$)	[Exclusive] Full Join ($A \oplus B$)	[Exclusive] Left Join ($\sim B$)
 <code>SELECT * FROM A RIGHT JOIN B ON A.key = B.key WHERE B.key IS NULL</code>	 <code>SELECT * FROM A FULL JOIN B ON A.key = B.key WHERE B.key IS NULL OR A.key IS NULL</code>	 <code>SELECT * FROM A LEFT JOIN B ON A.key = B.key WHERE B.key IS NULL</code>	 <code>SELECT * FROM A INNER JOIN B ON A.key = B.key</code>
Inner Join ($A \cap B$)	[Inclusive] Right Join (B)	[Inclusive] Full Join ($A \vee B$)	[Inclusive] Left Join (A)
 <code>SELECT * FROM A RIGHT JOIN B ON A.key = B.key</code>	 <code>SELECT * FROM A FULL JOIN B ON A.key = B.key</code>	 <code>SELECT * FROM A LEFT JOIN B ON A.key = B.key</code>	 <code>SELECT * FROM A INNER JOIN B ON A.key = B.key</code>

Now let's consider two tables and how they can be joined on the `student_id` column:

Table `student`:

student_id	name	age
1	John Stramer	50
2	John Wick	35
3	Jack Bauer	45

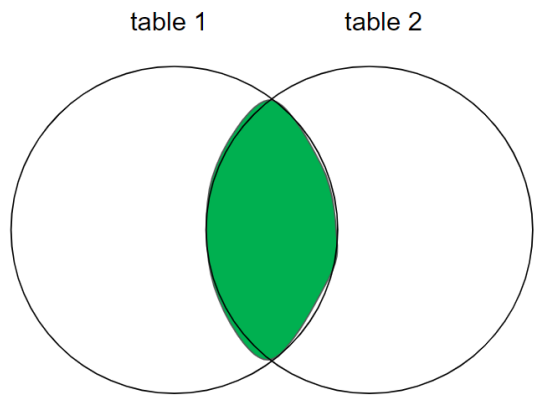
Table `course`:

course_id	student_id
1	1
1	2
2	1

course_id	student_id
3	10

Inner joins

Intersection of two tables, meaning all rows that exist for both.




Command:

```
SELECT * FROM student INNER JOIN course ON student.student_id = course.student_id;
```

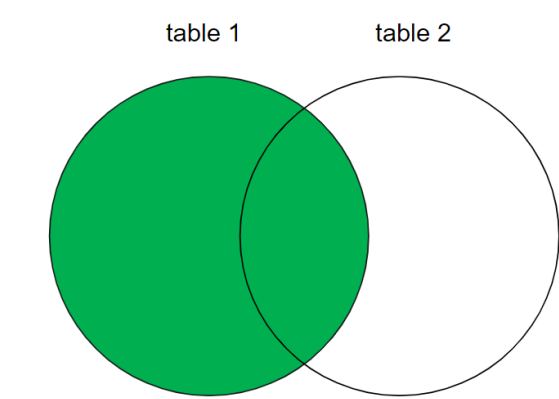
Output:

student_id	name	age	course_id	student_id
1	John Stramer	50	1	1
2	John Wick	35	1	2
1	John Stramer	50	2	1

 \x - toggle expanded display.

Left (outer) join

Will keep the unrelated data from the left (the first) table. Left join gets all rows from the left table, but from the right table - only rows that are linked to those of the table on the left. Missing data from the right table will have NULL values.



Command:

```
SELECT * FROM student LEFT JOIN course ON student.student_id = course.student_id;
```

Output:

student_id	name	age	course_id	student_id
1	John Stramer	50	1	1
2	John Wick	35	1	2
1	John Stramer	50	2	1
3	Jack Bauer	45	NULL	NULL

More examples:

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
-- Only show entries that don't have a car
SELECT * FROM person LEFT JOIN car ON car.id = person.car_id WHERE car.* IS NULL;
```

Note: we can change the join type from LEFT JOIN to RIGHT JOIN and vise versa as long as we also change the order of the tables

For example, these two statements should return the same result:

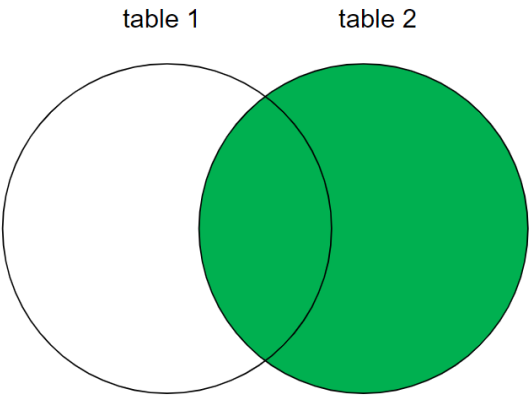
```
SELECT o.OrderId, o.OrderDate, c.CustomerId, c.FirstName, c.LastName, c.Country FROM Customers c
LEFT JOIN Orders o ON c.CustomerId = o.CustomerId;
```

and

```
SELECT o.OrderId, o.OrderDate, c.CustomerId, c.FirstName, c.LastName, c.Country FROM Orders o
RIGHT JOIN Customers c ON o.CustomerId = c.CustomerId
```

Right (outer) join

All rows from the second / right table + the rows that match the rows from the second table .



Command:

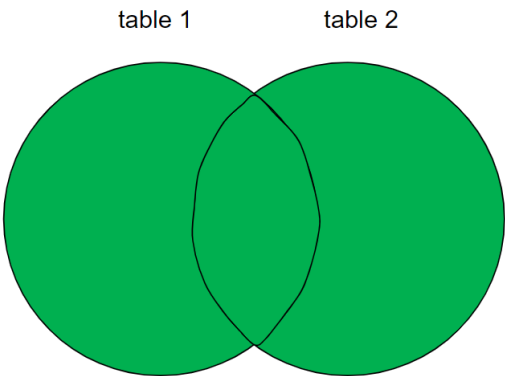
```
SELECT * FROM student RIGHT JOIN course ON student.student_id = course.student_id;
```

Output:

student_id	name	age	course_id	student_id
1	John Stramer	50	1	1
2	John Wick	35	1	2
1	John Stramer	50	2	1
NULL	NULL	NULL	3	10

Full (outer) join

Combine all values from the two tables, including those with NULL values.



Command:

```
SELECT * FROM student FULL JOIN course ON student.student_id = course.student_id;
```

Output:

student_id	name	age	course_id	student_id
1	John Stramer	50	1	1
2	John Wick	35	1	2
1	John Stramer	50	2	1
NULL	NULL	NULL	3	10
3	Jack Bauer	45	NULL	NULL

More examples:

```
select * from a FULL OUTER JOIN b on a.a = b.b;

SELECT * FROM table1 FULL JOIN table2 ON table1.id = table2.char_id;
```

Or, if the column has the same name:

```
SELECT * FROM table1 JOIN table2 USING (id_name)
```

```
-- We have two tables with primary and foreign key "employee_id", and we want to
show ids that are not used for inner join (because they are not in both tables)
SELECT absent_in_one AS employee_id
FROM (
  SELECT
  CASE
  WHEN e_emp_id IS NULL THEN s_emp_id
  WHEN s_emp_id IS NULL THEN e_emp_id
  ELSE NULL
  END AS absent_in_one
  FROM (
    SELECT e.employee_id AS e_emp_id, e.name AS e_name, s.employee_id AS
s_emp_id, s.salary AS s_salary
    FROM Employees e
    FULL JOIN Salaries s
    ON e.employee_id = s.employee_id
  )
)
WHERE absent_in_one IS NOT NULL
```

Or if we want to joint three tables:

```
SELECT columns FROM junction_table
FULL JOIN table_1 ON junction_table.foreign_key_column =
table_1.primary_key_column
FULL JOIN table_2 ON junction_table.foreign_key_column =
table_2.primary_key_column;
```

Multi-table joins

Example:

```
-- example 1
SELECT c.CustomerName, o.OrderDate, p.ProductName
FROM Customers c
INNER JOIN Orders o ON c.CustomerID = o.CustomerID
INNER JOIN Products p ON o.ProductID = p.ProductID;

-- example 2
SELECT c.CustomerName, o.OrderDate, p.ProductName
FROM Customers c
INNER JOIN Orders o ON c.CustomerID = o.CustomerID
LEFT JOIN Products p ON o.ProductID = p.ProductID;
```

Self join

Joining a table with itself. Can utilise inner, left, right, or full outer joins.

For example, let's consider the following table.

id	name	salary	managerId
1	Joe	70000	3
2	Henry	80000	4
3	Sam	60000	null
4	Max	90000	null

We can join each employee with their manager:

name1	salary1	name2	salary2
Joe	70000	Sam	60000
Henry	80000	Max	90000

This can be done by using the following command:

```
SELECT e1.name AS name1, e1.salary AS salary1, e2.name AS name2, e2.salary AS salary2
FROM Employee e1
JOIN Employee e2
ON e1.managerId = e2.id
```

CASE WHEN

Creating a new column / field based on a condition for the other columns.

```
-- General view
CASE WHEN
condition1 THEN result1
WHEN condition2 THEN result2
WHEN conditionN THEN resultN
ELSE else_result
END AS alias;
```

Here is an example where we create a new field that will detail if a student passed or failed, based on their scores:

```
SELECT student_id, student_name, exam_score,
CASE WHEN exam_score >= 60 THEN 'Pass'
ELSE 'Fail'
END AS result
FROM students;
```

```
-- CASE WHEN can be used within a aggregate function
-- For example, take values where rating < 3 as 1 (otherwise, take as 0), and sum
them - that counts how many ratings there are with a value of less than 3
SUM(case when rating < 3 then 1 else 0 end)

-- An example: multiply by -1 if another column says "Buy", else take the original
value
SELECT stock_name,
CASE
    WHEN operation = 'Buy' THEN price * -1
    ELSE price
END AS capital_proc
FROM Stocks
```

Pivot

Wide -> long

From table:

name	sport	color	bonus
name1	basketball	green	10
name2	volleyball	red	5

To table:

name	category	value
name1	sport	basketball
name1	color	green
name1	bonus	10
name2	sport	volleyball
name2	color	red
name2	bonus	5

```
select
  name,
  'sport' as category,
  sport as value
from wideClient
union all
select
  name,
  'color' as category,
  color as value
from wideClient
union all
select
  name,
  'bonus' as category,
  bonus as value
from wideClient
```

Long -> wide

```
Input:
Department table:
+-----+-----+-----+
| id   | revenue | month |
+-----+-----+-----+
```


1	8000	Jan
2	9000	Jan
3	10000	Feb
1	7000	Feb
1	6000	Mar

+-----+-----+-----+

Output:

id	Jan_Revenue	Feb_Revenue	Mar_Revenue	...	Dec_Revenue
1	8000	7000	6000	...	null
2	9000	null	null	...	null
3	null	10000	null	...	null

Query:

```
SELECT
  id,
  MAX(CASE WHEN month='Jan' THEN revenue ELSE null END) AS Jan_Revenue,
  MAX(CASE WHEN month='Feb' THEN revenue ELSE null END) AS Feb_Revenue,
  MAX(CASE WHEN month='Mar' THEN revenue ELSE null END) AS Mar_Revenue,
  MAX(CASE WHEN month='Apr' THEN revenue ELSE null END) AS Apr_Revenue,
  MAX(CASE WHEN month='May' THEN revenue ELSE null END) AS May_Revenue,
  MAX(CASE WHEN month='Jun' THEN revenue ELSE null END) AS Jun_Revenue,
  MAX(CASE WHEN month='Jul' THEN revenue ELSE null END) AS Jul_Revenue,
  MAX(CASE WHEN month='Aug' THEN revenue ELSE null END) AS Aug_Revenue,
  MAX(CASE WHEN month='Sep' THEN revenue ELSE null END) AS Sep_Revenue,
  MAX(CASE WHEN month='Oct' THEN revenue ELSE null END) AS Oct_Revenue,
  MAX(CASE WHEN month='Nov' THEN revenue ELSE null END) AS Nov_Revenue,
  MAX(CASE WHEN month='Dec' THEN revenue ELSE null END) AS Dec_Revenue
FROM Department
GROUP BY id
ORDER BY id ASC
```

Export query to CSV

```
\copy (SELECT ...) TO '/Users/Desktop/file.csv' DELIMITER ',' CSV HEADER;
# Example
\copy (SELECT * FROM table1 WHERE first_name='Evgenii') TO
'/Users/evgen/Desktop/query2.csv' DELIMITER ',' CSV HEADER;
```

Procedures

In SQL, stored procedure is a set of statement(s) that perform some defined actions. We make stored procedures so that we can reuse statements that are used frequently. Below are the procedures for PostgreSQL.

Not sure if I can get a procedure to return information in a SELECT statement.

Check all procedures for postgresSQL

```
\df
```

Create a new procedure for PostgreSQL

```
CREATE PROCEDURE proc_1 ()  
LANGUAGE SQL  
AS $$  
SELECT * FROM table1;  
$$;
```

Run a procedure

```
CALL proc_1();
```

E.g. a procedure for inserting a new entry

```
# Create procedure  
CREATE PROCEDURE proc_insertrecord  
(var1 VARCHAR(30), var2 VARCHAR(30), var3 INT)  
LANGUAGE SQL  
AS $$  
INSERT INTO table1 (first_name, gender, age)  
VALUES (var1, var2, var3);  
$$;  
  
# Run procedure  
CALL proc_insertRecord ('Isabel2', 'weird', 10);
```

Delete a procedure

```
DROP PROCEDURE proc_1;
```

Views

A View is a kind of a table that is based on results of a previous SQL query. For example, you can save a table view upon running the inner join command, and then perform actions on that view table to not type in the join command over and over again.

Uses and advantages:

- Views can join and simplify multiple tables into a single virtual table;
- Views can act as aggregated tables
- Views can hide the complexity of data
- Views can provide extra security to a DBMS

After you create a view, it shows in the list of tables using the command `\d`. Nevertheless, this view is not a table; it simply is a result of a saved query.

```
# Create a view of a table
CREATE VIEW table1_view_males AS
SELECT * FROM table1 WHERE gender = 'Male';

# Show a table view
SELECT * FROM table1_view_males;

# Update a view
CREATE OR REPLACE VIEW view1 AS ...;

# Delete a view
DROP VIEW view1;
```

A more practical example

```
# Let's say you have a join query
SELECT table1.first_name, table1.gender, table1.age, table2.item
FROM table1
INNER JOIN table2 ON table1.first_name = table2.first_name;

# If you want to make an operation on it, instead of writing it out every time,
you can save it as a view and then perform that action on the view of the table
CREATE VIEW table1_table2_innerjoin AS
SELECT table1.first_name, table1.gender, table1.age, table2.item
FROM table1
INNER JOIN table2 ON table1.first_name = table2.first_name;

# So now, you can perform operations on that view object you created,
# for example, you can count rows
SELECT COUNT(*) FROM table1_table2_innerjoin;
```

Transaction

A transaction is a group of queries to DB that either complete successfully all together or are not completed at all.

A SQL transaction is a sequence of database operations that behave as a single unit of work. It ensures that multiple operations are executed in an atomic and consistent manner, which is crucial for maintaining database integrity. SQL transactions adhere to a set of principles known as ACID.

Primary statements used for managing SQL transactions:

- BEGIN TRANSACTION / START TRANSACTION
- COMMIT
- ROLLBACK

Example of a transaction: consider a bank database with two tables: Customers (customer_id, name, account_balance) and Transactions (transaction_id, transaction_amount, customer_id). To transfer a specific amount from one customer to another securely, you would use a SQL transaction as follows:

```
BEGIN TRANSACTION;

-- Reduce the balance of the sender
UPDATE Customers
SET account_balance = account_balance - 100
WHERE customer_id = 1;

-- Increase the balance of the receiver
UPDATE Customers
SET account_balance = account_balance + 100
WHERE customer_id = 2;

-- Insert a new entry into the Transactions table
INSERT INTO Transactions (transaction_amount, customer_id)
VALUES (-100, 1),
       (100, 2);

-- Check if the sender's balance is sufficient
IF (SELECT account_balance FROM Customers WHERE customer_id = 1) >= 0
    COMMIT;
ELSE
    ROLLBACK;
```

SQL transactions are crucial in various real-world scenarios that require multiple database operations to occur atomically and consistently. Below are some common examples:

- E-commerce: When processing an order that includes billing, shipping, and updating the inventory, it is essential to execute these actions as a single transaction to ensure data consistency and avoid potential double bookings, incorrect inventory updates, or incomplete order processing.
- Banking and financial systems: Managing accounts, deposits, withdrawals, and transfers require transactions for ensuring data integrity and consistency while updating account balances and maintaining audit trails of all transactions.

- Reservation systems: For booking tickets or accommodations, the availability of the seats or rooms must be checked, confirmed, and updated in the system. Transactions are necessary for this process to prevent overbooking or incorrect reservations.
- User registration and authentication: While creating user accounts, it is vital to ensure that the account information is saved securely to the correct tables and without duplicates. Transactions can ensure atomicity and isolation of account data operations.

Potential issues with SQL transactions:

- Isolation problems:
 - Dirty reads - where a transaction may see uncommitted changes made by some other transaction.
 - Non-repeatable reads: Before transaction A is over, another transaction B also accesses the same data. Then, due to the modification caused by transaction B, the data read twice from transaction A may be different. The key to non-repeatable reading is to modify: In the same conditions, the data you have read, read it again, and find that the value is different.
 - Phantom reads: When the user reads records, another transaction inserts or deletes rows to the records being read. When the user reads the same rows again, a new "phantom" row will be found. The key point of the phantom reading is to add or delete: Under the same conditions, the number of records read out for the first time and the second time is different.
- Deadlocks
- Lost updates
- Long-running transactions

Isolation levels

Read more: <https://blog.iddqd.uk/interview-section-databases/>

Transaction isolation levels are how SQL databases solve data reading problems in concurrent transactions.

The four isolation levels in increasing order of isolation attained for a given transaction, are READ UNCOMMITTED , READ COMMITTED , REPEATABLE READ , and SERIALIZABLE.

- **Read uncommitted:** one transaction can read the data of another uncommitted transaction.
 - Weakest isolation, but also the fastest;
 - Allows dirty reads, non-repeatable reads, phantoms
 - Is acceptable when 1) you are reading data that you know will never be modified in any way or 2) for non-critical summary reports
- **Read committed:** a transaction cannot read data until another transaction is committed.
 - Default for PostgreSQL
 - Prevents dirty reads
 - Allows non-repeatable reads, phantom reads
- **Repeatable read:** when starting to read data (transaction is opened), modification operations are no longer allowed. Solved non-repeatable read.
 - Default for MySQL
 - Prevents dirty reads, non-repeatable reads
 - Allows phantoms

- **Serializable:** Serializable is the highest transaction isolation level. Under this level, transactions are serialized and executed sequentially, which can avoid dirty read, non-repeatable read, and phantom read. However, this transaction isolation level is inefficient and consumes database performance, so it is rarely used.
 - Strongest isolation, but also the slowest
 - Prevents dirty reads, non-repeatable reads, and phantom reads

Denormalisation

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database. Note that denormalization does not mean 'reversing normalization' or 'not to normalize'. It is an optimization technique that is applied after normalization.

Basically, The process of taking a normalized schema and making it non-normalized is called denormalization, and designers use it to tune the performance of systems to support time-critical operations.

In a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in a database. For example, in a normalized database, we might have a Courses table and a Teachers table. Each entry in Courses would store the teacherID for a Course but not the teacherName. When we need to retrieve a list of all Courses with the Teacher's name, we would do a join between these two tables. In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place. The drawback is that if tables are large, we may spend an unnecessarily long time doing joins on tables. Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Pros of Denormalization:

- Improved query performance: retrieving data is faster since we do fewer joins;
- Reduced complexity: By combining related data into fewer tables, denormalization can simplify the database schema and make it easier to manage.
- Simplification of queries: Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.
- Easier Maintenance and Updates: Denormalization can make it easier to update and maintain the database by reducing the number of tables.
- Improved Read Performance: Denormalization can improve read performance by making it easier to access data.
- Better Scalability: Denormalization can improve the scalability of a database system by reducing the number of tables and improving the overall performance.

Cons of Denormalization:

- Reduced data integrity: By adding redundant data, denormalization can reduce data integrity and increase the risk of inconsistencies.
- Increased Complexity: While denormalization can simplify the database schema in some cases, it can also increase complexity by introducing redundant data.

- **Increased Storage Requirements:** By adding redundant data, denormalization can increase storage requirements and increase the cost of maintaining the database.
- **Increased Update and Maintenance Complexity:** Denormalization can increase the complexity of updating and maintaining the database by introducing redundant data.
- **Limited Flexibility:** Denormalization can reduce the flexibility of a database system by introducing redundant data and making it harder to modify the schema.

Relationships

Relationships in SQL are a way to establish connections between multiple tables. There are five different types of relationships between tables:

- One-to-one
- One-to-many
- Many-to-many
- Many-to-one
- Self-referencing

Read more: <https://www.geeksforgeeks.org/relationships-in-sql-one-to-one-one-to-many-many-to-many/>

One-to-one

- **Definition:** Each record in Table A is associated with one and only one record in Table B, and vice versa.
- **Setup:** Include a foreign key in one of the tables that references the primary key of the other table.

```
-- For example: Tables users and user_profiles, where each user has a single
corresponding profile.
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50));
CREATE TABLE user_profiles (
    profile_id INT PRIMARY KEY,
    user_id INT UNIQUE,
    profile_data VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES users(user_id));
```

One-to-many

- **Definition:** Each record in Table A can be associated with multiple records in Table B, but each record in Table B is associated with only one record in Table A.
- **Setup:** Include a foreign key in the "many" side table (Table B) that references the primary key of the "one" side table (Table A).

```
-- For example: Tables departments and employees, where each department can have
multiple employees, but each employee belongs to one department.
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
```

```
department_name VARCHAR(50));
CREATE TABLE employees (
  employee_id INT PRIMARY KEY,
  employee_name VARCHAR(50),
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES departments(department_id));
```

Example: each character from the Mario franchise is associated with multiple filenames that represent sounds, but each sound is only connected to one character.

In this example, the foreign key from table B (`sounds`) references the primary key from table A (`characters`):

```
mario_database=> SELECT * FROM characters FULL JOIN sounds ON
characters.character_id = sounds.character_id;
mario_database=>
```

character_id	name	homeland	favorite_color	sound_id	filename
1	Mario	Mushroom Kingdom	Red	1	its-a-me.wav
1	Mario	Mushroom Kingdom	Red	2	yippee.wav
2	Luigi	Mushroom Kingdom	Green	3	ha-ha.wav
2	Luigi	Mushroom Kingdom	Green	4	oh-yeah.wav
3	Peach	Mushroom Kingdom	Pink	5	yay.wav
3	Peach	Mushroom Kingdom	Pink	6	woo-hoo.wav
3	Peach	Mushroom Kingdom	Pink	7	mm-hmm.wav
1	Mario	Mushroom Kingdom	Red	8	yahoo.wav

Many-to-many

- Definition: Each record in Table A can be associated with multiple records in Table B, and vice versa.
- Setup: Create an intermediate (junction, linking) table that contains foreign keys referencing both related tables.

```
-- For example: Tables students and courses, where each student can enroll in
multiple courses, and each course can have multiple students.
CREATE TABLE students (
```



```

    student_id INT PRIMARY KEY,
    student_name VARCHAR(50));
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50));
CREATE TABLE student_courses (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id));

```

Many-to-one

Note: A Many-to-One relation is the same as one-to-many, but from a different viewpoint.

- Definition: Multiple records in table A can be associated with one record in table B.
- Setup: Create a Foreign key in "Many Table" that references to Primary Key in "One Table".

```

-- Example: Table Courses and Teachers, many courses can be taught by single
teacher.
CREATE TABLE Teachers (
    teacher_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255)
);
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(255),
    teacher_id INT,
    FOREIGN KEY (teacher_id) REFERENCES Teachers(teacher_id)
);

```

Self-referencing

- Definition: A table has a foreign key that references its primary key.
- Setup: Include a foreign key column in the same table that references its primary key.

```

-- For example : A table `employees` with a column `manager_id` referencing the
same table's `employee_id`.
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id));

```

PostgreSQL

Login: `psql --username=<username-here> --dbname=<dbname-here>`