**Queries**

1. \*\*SELECT Statement:\*\*

- Retrieves data from one or more tables.

- Example: `SELECT column1, column2 FROM table\_name WHERE condition;`

Une image contenant texte, menu, capture d’écran, document

Description générée automatiquement 2. \*\*WHERE Clause:\*\*

- Filters records based on a specified condition.

- Example: `SELECT \* FROM table\_name WHERE column\_name = value;`

3. \*\*ORDER BY Clause:\*\*

- Sorts the result set in ascending or descending order.

- Example: `SELECT \* FROM table\_name ORDER BY column\_name DESC;`

4. \*\*INSERT INTO Statement:\*\*

- Adds new records to a table.

- Example: `INSERT INTO table\_name (column1, column2) VALUES (value1, value2);`

5. \*\*UPDATE Statement:\*\*

- Modifies existing records in a table.

Une image contenant texte, menu, capture d’écran, document

Description générée automatiquement - Example: `UPDATE table\_name SET column\_name = new\_value WHERE condition;`

6. \*\*DELETE Statement:\*\*

- Removes records from a table based on a condition.

- Example: `DELETE FROM table\_name WHERE condition;`

7. \*\*GROUP BY Clause:\*\*

- Groups rows that have the same values into summary rows.

- Example: `SELECT column\_name, COUNT(\*) FROM table\_name GROUP BY column\_name;`

8. \*\*HAVING Clause:\*\*

- Filters the results of a GROUP BY clause based on a condition.

- Example: `SELECT column\_name, COUNT(\*) FROM table\_name GROUP BY column\_name HAVING COUNT(\*) > 1;`

9. \*\*JOIN Clause:\*\*

- Combines rows from two or more tables based on a related column.

- Example: `SELECT \* FROM table1 JOIN table2 ON table1.column\_name = table2.column\_name;`

10. \*\*UNION Operator:\*\*

- Combines the result sets of two or more SELECT statements.

- Example: `SELECT column\_name FROM table1 UNION SELECT column\_name FROM table2;`

11. \*\*DISTINCT Keyword:\*\*

- Returns unique values in a specified column.

- Example: `SELECT DISTINCT column\_name FROM table\_name;`

12. \*\*LIKE Operator:\*\*

- Searches for a specified pattern in a column.

- Example: `SELECT \* FROM table\_name WHERE column\_name LIKE 'pattern%';`

13. \*\*IN Operator:\*\*

- Specifies multiple values in a WHERE clause.

- Example: `SELECT \* FROM table\_name WHERE column\_name IN (value1, value2, ...);`

14. \*\*BETWEEN Operator:\*\*

- Filters the result set within a range.

- Example: `SELECT \* FROM table\_name WHERE column\_name BETWEEN value1 AND value2;`

15. LIMIT and OFFSET

SELECT columns

FROM table

LIMIT number\_of\_rows

OFFSET offset\_value; - `columns`: The columns you want to retrieve.

- `table`: The name of the table you're querying.

- `number\_of\_rows`: The maximum number of rows to retrieve.

- `offset\_value`: The number of rows to skip from the beginning of the result set.

16. SUBQUERIES

SELECT customer\_name, (SELECT COUNT(\*) FROM orders WHERE customer\_id = customers.customer\_id) AS total\_orders

FROM customers;

**PSQL**

\*\*1. Creating a Table:\*\***types** : BOOLEAN for Boolean values • INT for integers (4-byte) • SERIAL for an auto-incrementing identifier (4-byte), or AUTO INCREMENT with MySQL • REAL for floating-point numbers (4-byte) • NUMERIC for high-precision numbers (1000 digits) • TEXT or VARCHAR: text • VARCHAR(42): text of length at most 42 • BYTEA or BLOB for binary strings • TIMESTAMP for date and time (can be WITH TIME ZONE), DATE, etc.

CREATE TABLE employees (

employee\_id SERIAL,

first\_name VARCHAR (50),

hire\_date DATE

\*2. Inserting Data:\*\*

INSERT INTO employees (first\_name, last\_name, birth\_date, hire\_date)

VALUES ('John', 'Doe', '1990-05-15', '2015-08-20');

\*\*4. Updating Data:\*\*

UPDATE employees

SET hire\_date = '2017-04-25'

WHERE employee\_id = 1;

\*\*5. Deleting Data:\*\*

DELETE FROM employees

WHERE employee\_id = 2;

\*\*6. Primary Key and Constraints:\*\*

In the table creation example, we defined the `employee\_id` column as the primary key. This enforces uniqueness and provides a quick way to identify each record.

\*\*7. Data Types:\*\*

PostgreSQL supports various data types, including `VARCHAR`, `DATE`, `INT`, `SERIAL` (for auto-incrementing integers), and more. Choose appropriate data types for your table columns.

\*\*8. Indexes:\*\*

You can create indexes on columns to improve query performance. For example:

CREATE INDEX idx\_employee\_last\_name ON employees (last\_name);

This creates an index on the "last\_name" column of the "employees" table.

\*\*9. Foreign Keys:\*\*

You can establish relationships between tables using foreign keys. For instance, if you have a "departments" table, you can link it to the "employees" table with a foreign key.  
-- Add a foreign key to the "students" table  
ALTER TABLE students  
ADD COLUMN course\_id INT REFERENCES courses(course\_id);  
**\*\*`ALTER TABLE` Command**  
1. \*\*Add a Column:\*\*  
- `ALTER TABLE employees ADD COLUMN email VARCHAR(100);  
2. \*\*Modify a Column:\*\*  
- `ALTER TABLE employees ALTER COLUMN hire\_date SET DATA TYPE DATE;`  
3. \*\*Rename a Column:\*\*  
 - `ALTER TABLE employees RENAME COLUMN first\_name TO first\_name\_new;`  
4. \*\*Add a Primary Key:\*\*  
- `ALTER TABLE employees ADD CONSTRAINT pk\_employee\_id PRIMARY KEY (employee\_id);`  
5. \*\*Add a Foreign Key:\*\*  
- `ALTER TABLE employees ADD CONSTRAINT fk\_department\_id FOREIGN KEY (department\_id) REFERENCES departments(department\_id);`  
6. \*\*Add an Index:\*  
 - `ALTER TABLE employees ADD INDEX idx\_last\_name (last\_name);`  
7. \*\*Drop a Column:\*\*  
- `ALTER TABLE employees DROP COLUMN email;`  
8. \*\*Rename a Table:\*\*  
- `ALTER TABLE employees RENAME TO staff;`

**Cours intro databases**

Properties of DataBase Management Systems • **Logical integrity** The DBMS should verify that update keep data in a consistent state with regard to the constraints on the structure of data. • **Physical integrity** The DBMS should try to stay coherent in the case of events like losing power, etc. • **Data sharing** Multiple users can access the data while preserving the logical and physical integrity. • **Standardized** A DBMS should use a standardized interface so one could swap one DBMS vendor by another vendor without any major change to the code.  
Some standards for databases **• Relational** simple but powerful model (tables), the one we will focus on! • **XML,** recursive data, complex queries, used to be hyped. • **Json/Documents**, similar to XML but the fad moved to this one. • **Graph** data is modeled as a graph, complex queries. • **Object** complex hierarchical data model, inspired by OOP • Key-Value simple queries and simple data model, performance oriented. • **Olap-cube** Oriented for performance of analytical queries.  
In summary A variety of properties Independence, Integrity, etc. and variety of standards Relational, Graph, Key-value, etc. but always handles data and is accessible through some interface

**Course on entity-relationship diagrams**

Une image contenant texte, capture d’écran, Police

Description générée automatiquementTotal participation: 0 is not acceptable, every entity must be in a relationship → Represented by a double line in an ER diagram  
 • Partial participation (default): 0 is acceptable, some entities are not in a relationship  
double line : o is not acceptable  
**Une image contenant texte, diagramme, capture d’écran, ligne

Description générée automatiquement**

Complete: “each employee is either a professor or a secretary” • Not complete: “there can also be other kinds of employees” •   
Disjoint: “an employee cannot be both a professor and a secretary” • Not disjoint: “an employee can be both”

**Functional dependencies and normal forms**  
• A relation is in Boyce-Codd Normal Form (BCNF) if for every non-trivial FD A1 . . . An → B1 . . . Bm that it satisfies, then A1 . . . An is a superkey

**Evaluation algorithms**

Different types of indexes:  
• **Hash** for equality tests only : Pros • Generally the fastest index (but with a small margin) • Especially good on large datasets with complex types like strings • Small space overhead Cons • At least two seeks required • Cannot retrieve data in sorted order • Can only check equality  
• **B-tree** for arbitrary comparisons : . Les arbres B permettent de créer des indexes sur certains champs d’une table pour pouvoir rapidement accéder aux données. Leur structure est optimisée pour un stockage sur disque ou SSD où la mémoire se manipule par blocks, en allouant un grand nombre de valeurs dans chaque nœud. Les opération suivantes : — recherche — insertion — suppression ont pour complexité moyenne et en pire cas O(ln(n)) . Pros • Quite fast (especially on simple types) • Retrieve data in order with very few seeks • Can be used for prefixes / subkeys; a B-tree on (x, y) can be used to test whether a given x exists or retrieve the corresponding y. Cons • Larger number of seeks required • Larger space overhead (especially for complex data types)  
•**Inverted Index query** Given a set of sets X1, . . . , Xn and an item i report the integers e such that i ∈ Xe Implementation Typically a B-tree or a hash index giving for each i the set of e such that i ∈ Xe . Usage They can be used for indexing JSON fields or getting records where some word appears in a string.  
**•Bitmap indexes** Principle For each value v store a bit array of length n (the number of records) where the i-th bit is set when the record has value v Typical use case • (Rarely) as an index built for values of low cardinality (gender, binary values, enum) • (Very often) built for intermediate results of complex conditions (e.g. x < 10 or y > 42 and z < 1).