



SQL

Data Query Language

André Restivo

Index

Introduction	Selecting Data	Choosing Columns	Filtering Rows
Set Operators	Joining Tables	Aggregating Data	Sorting Rows
Limiting Data	Text Operators	Nested Queries	

Introduction

SQL

- Structured Query Language.
- A special purpose language to manage data stored in a **relational** database.
- Based on **relational algebra**.
- Pronounced *Sequel*

History

- Early 70's SEQUEL Developed at IBM
- 1986 SQL-86 and SQL-87 Ratified by ANSI and ISO.
- 1989 SQL-89
- 1992 SQL-92 Also know as SQL2.
- 1999 SQL:1999 Also known as SQL3 Includes regular expressions, recursive queries, triggers, non-scalar data types and some object-oriented expressions.
- 2003 SQL:2003 XML support and auto-generated values.
- 2006 SQL:2006 XQuery support.
- 2008 SQL:2008.
- 2011 SQL:2011.

Standard

- Although SQL is an ANSI/ISO standard, every database system implements it in a slightly different way.
- These slides will try to adhere to the standard as much as possible.
- Sometimes we'll deviate and talk specifically about PostgreSQL.

Selecting Data

SELECT and FROM

- SELECT and FROM are the most basic SQL query operators.
- They allows to specify which tables (FROM) and columns (SELECT) we want to retrieve from the database.
- The result of an SQL query is also a table.

Selecting all columns

To select all columns from a table we can use an *

```
SELECT * FROM employee;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

? Selects all columns from table employee

Choosing columns

Choosing columns

We can select only some columns

```
SELECT id, name FROM employee;
```

id	name
1	John Doe
2	Jane Doe
3	John Smith
4	Jane Roe
5	Richard Roe

? Selects columns id and name from table employee

Column operations

We can also perform any operations between columns

```
SELECT id, name, salary - taxes FROM employee;
```

id	name	salary - taxes
1	John Doe	800
2	Jane Doe	700
3	John Smith	850
4	Jane Roe	800
5	Richard Roe	900

? Selects columns id, name and the difference between salary and taxes from table employee

Renaming columns

Any column can be renamed using the AS operator

```
SELECT id AS num, name, salary - taxes AS net_salary FROM employee;
```

num	name	net_salary
1	John Doe	800
2	Jane Doe	700
3	John Smith	850
4	Jane Roe	800
5	Richard Roe	900

? Renaming column id as num and the difference between salary and taxes to net_salary

Filtering Rows

WHERE

- The **WHERE** command allows us to filter which rows we want in our result table according to a condition.
- The condition can use any comparison operator (<, >, <=, <>, ...) and can be composed using AND, OR and NOT.

Example

```
SELECT * FROM employee WHERE dep_num = 2 OR salary <= 900;
```

id	name	salary	taxes	dep_num
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
5	Richard Roe	900	0	NULL

? Employees from department 2 or a salary lower or equal to 900

Example

To test if a value is null, we have to use the special IS NULL operator.

```
SELECT * FROM employee WHERE dep_num IS NULL;
```

id	name	salary	taxes	dep_num
5	Richard Roe	900	0	NULL

? Employees from department 2 or a salary lower or equal to 900

Use IS NOT NULL to select rows where a certain attribute is not null.

Removing duplicates

We can remove duplicates from the final result by using the DISTINCT operator

```
SELECT DISTINCT salary FROM employee;
```

salary
1000
800
1200
900

? Selects the different salaries in the database

Set operators

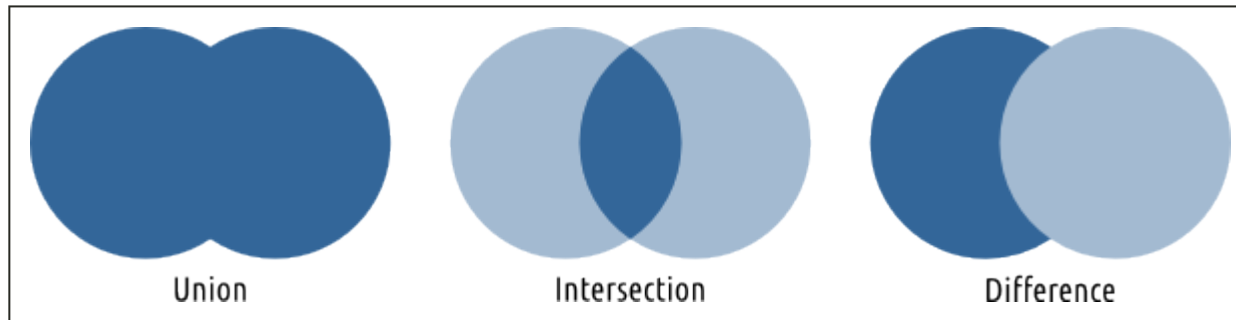
Set operators

Two tables are compatible for being used in a set operation if they have the same **number of columns** and the **type** of each column is **compatible**:

- The **UNION** between two tables (R1 and R2) is a table that includes all lines that are present in R1 or R2.
- The **INTERSECT**ion between two tables (R1 and R2) is a table that includes all lines that are present in R1 and R2.
- The difference (**EXCEPT**) between two tables (R1 and R2) is a table that includes all lines that are present in R1 but not in R2.

With all these operators, **duplicate** rows are **eliminated** automatically.

Union, Intersection and Difference



Example

```
SELECT * FROM employee WHERE salary >= 1000;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3

```
SELECT * FROM employee WHERE id_dep = 2;
```

id	name	salary	taxes	dep_num
2	Jane Doe	800	100	2
3	John Smith	1200	350	2

Union

```
SELECT * FROM employee WHERE salary >= 1000;  
UNION  
SELECT * FROM employee WHERE id_dep = 2;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3

? Employees that have a salary larger or equal to 1000 or work on department 2

Intersection

```
SELECT * FROM employee WHERE salary >= 1000;  
INTERSECT  
SELECT * FROM employee WHERE id_dep = 2;
```

id	name	salary	taxes	dep_num
3	John Smith	1200	350	2

? Employees that have a salary larger or equal to 1000 and work on department 2

Difference

```
SELECT * FROM employee WHERE salary >= 1000;  
EXCEPT  
SELECT * FROM employee WHERE id_dep = 2;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
4	Jane Roe	1000	200	3

? Employees that have a salary larger or equal to 1000 and do not work on department 2

Joining Tables

Cartesian product

- The cartesian product allows us to combine rows from **different tables**.
- To use it, we just have to indicate which tables we want to combine using commas to separate them.
- The result is a table containing the columns of all tables and **all possible combinations** of rows.
- Also known as **CROSS JOIN**.

Example

```
SELECT * FROM employee;
```

id	name	salary	taxes	dep_num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

```
SELECT * FROM department;
```

num	name
1	Marketing
2	Sales
3	Production

Cartesian product

```
SELECT * FROM employee, department;
```

```
SELECT * FROM employee CROSS JOIN department;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	1	Marketing
3	John Smith	1200	350	2	1	Marketing
4	Jane Roe	1000	200	3	1	Marketing
5	Richard Roe	900	0	NULL	1	Marketing
1	John Doe	1000	200	1	2	Sales
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	2	Sales
5	Richard Roe	900	0	NULL	2	Sales
1	John Doe	1000	200	1	3	Production
2	Jane Doe	800	100	2	3	Production
3	John Smith	1200	350	2	3	Production
4	Jane Roe	1000	200	3	3	Production
5	Richard Roe	900	0	NULL	3	Production

Solving ambiguities

When selecting from more than one table, columns with the same name might lead to ambiguities.

```
SELECT id, name, name FROM employee, department;
```

To solve them we must use the table name before the column name.

```
SELECT id, employee.name, department.name FROM employee, department;
```

Solving ambiguities

```
SELECT id, employee.name, department.name FROM employee, department;
```

id	name	name
1	John Doe	Marketing
2	Jane Doe	Marketing
3	John Smith	Marketing
4	Jane Roe	Marketing
5	Richard Roe	Marketing
1	John Doe	Sales
2	Jane Doe	Sales
3	John Smith	Sales
4	Jane Roe	Sales
5	Richard Roe	Sales
1	John Doe	Production
2	Jane Doe	Production
3	John Smith	Production
4	Jane Roe	Production
5	Richard Roe	Production

Joining using WHERE

```
SELECT * FROM employee, department WHERE dep_num = num;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	1	Marketing
3	John Smith	1200	350	2	1	Marketing
4	Jane Roe	1000	200	3	1	Marketing
5	Richard Roe	900	0	NULL	1	Marketing
1	John Doe	1000	200	1	2	Sales
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	2	Sales
5	Richard Roe	900	0	NULL	2	Sales
1	John Doe	1000	200	1	3	Production
2	Jane Doe	800	100	2	3	Production
3	John Smith	1200	350	2	3	Production
4	Jane Roe	1000	200	3	3	Production
5	Richard Roe	900	0	NULL	3	Production

Joining using WHERE

```
SELECT * FROM employee, department WHERE dep_num = num;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	3	Production

? Employees and their departments

Join using JOIN ... ON

- Instead of using a cartesian product followed by the WHERE keyword, we can use the more specific keywords: JOIN ON.
- These keywords allow us to specify simultaneously **which tables** to join and with which joining **condition**.
- Separates regular row filtering from joining conditions.
- Makes joining lots of tables **easier** to understand.

```
SELECT * FROM employee, department WHERE dep_num = num;
```

Same as:

```
SELECT * FROM employee JOIN department ON dep_num = num;
```

Join using JOIN ... ON

```
SELECT * FROM employee JOIN department ON dep_num = num;
```

id	name	salary	taxes	dep_num	num	name
1	John Doe	1000	200	1	1	Marketing
2	Jane Doe	800	100	2	2	Sales
3	John Smith	1200	350	2	2	Sales
4	Jane Roe	1000	200	3	3	Production

Join using JOIN ... USING

If the columns used in the join operation have the same name, we can join them using in a simpler way with JOIN USING.

```
SELECT * FROM employee;
```

id	name	salary	taxes	num
1	John Doe	1000	200	1
2	Jane Doe	800	100	2
3	John Smith	1200	350	2
4	Jane Roe	1000	200	3
5	Richard Roe	900	0	NULL

```
SELECT * FROM department;
```

num	name
1	Marketing
2	Sales
3	Production

Join using JOIN ... USING

```
SELECT * FROM employee JOIN department USING(num);
```

id	name	salary	taxes	num	name
1	John Doe	1000	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1200	350	2	Sales
4	Jane Roe	1000	200	3	Production

Renaming tables

- We can also rename tables.
- This might be useful if we need to use the **same table twice** in the same query.
- Or if we want to make the table names **simpler** in a complicated query.

```
SELECT * FROM employee;
```

id	name	sup_id
1	John Doe	NULL
2	Jane Doe	1
3	John Smith	1
4	Jane Roe	NULL
5	Richard Roe	4

Renaming tables

```
SELECT * FROM employee JOIN employee AS supervisor ON employee.sup_id = supervisor.id;
```

id	name	sup_id	id	name	sup_id
2	Jane Doe	1	1	John Doe	NULL
3	John Smith	1	1	John Doe	NULL
5	Richard Roe	4	4	Jane Roe	NULL

Aggregating Data

Aggregate Functions

There are five special aggregate functions defined in the SQL language:

MIN, MAX, SUM, AVG and COUNT

value
1
2
NULL
3

```
SELECT MIN(value) FROM table; -- 1
SELECT MAX(value) FROM table; -- 3
SELECT SUM(value) FROM table; -- 6
SELECT AVG(value) FROM table; -- 2
SELECT COUNT(value) FROM table; -- 3 (counts non null values)
SELECT COUNT(*) FROM table; -- 4 (counts lines)
```

Aggregate Functions

When using aggregate functions, all rows are grouped into a single row.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT AVG(salary)      FROM employees; -- 1000
SELECT name, AVG(salary) FROM employees; -- Does not work
```

? You can no longer refer to columns without an aggregate function.

GROUP BY

Groups the rows into sets based on the value of a specific column or columns.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT AVG(salary) FROM employees GROUP BY num;
```

- All rows with same value in the *num* column get grouped together.
- The aggregate functions are used inside each group.

AVG(salary)
700
1150
1000

GROUP BY

Only columns used in GROUP BY expressions can be selected without using an aggregate function.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT num, AVG(salary) FROM employees GROUP BY num;
```

num	AVG(salary)
1	700
2	1150
3	1000

GROUP BY

Only columns used in **GROUP BY** expressions can be selected without using an aggregate function.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT num, d_name, AVG(salary) FROM employees GROUP BY num, d_name;
```

d_name	AVG(salary)
Marketing	700
Sales	1150
Production	1000

HAVING

Grouped rows can be filtered using the HAVING clause.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT num, d_name, AVG(salary) FROM employees
GROUP BY num, d_name HAVING AVG(salary) >= 1000;
```

d_name	AVG(salary)
Sales	1150
Production	1000

Sorting Rows

ORDER BY

- The order in which rows are sorted in a query result is unpredictable.
- You can sort the end result using the **ORDER BY** clause.

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

```
SELECT * FROM employees ORDER BY salary;
```

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
4	Jane Roe	1000	200	3	Production
3	John Smith	1500	350	2	Sales

ORDER BY

- By default values are sorted in **ascending** order.
- We can change the default order using the **ASC** and **DESC** clauses.

```
SELECT * FROM employees ORDER BY salary ASC; -- default
```

id	name	salary	taxes	num	d_name
1	John Doe	700	200	1	Marketing
2	Jane Doe	800	100	2	Sales
4	Jane Roe	1000	200	3	Production
3	John Smith	1500	350	2	Sales

```
SELECT * FROM employees ORDER BY salary DESC; -- default
```

id	name	salary	taxes	num	d_name
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production
2	Jane Doe	800	100	2	Sales
1	John Doe	700	200	1	Marketing

ORDER BY

- We can also sort by more than one column.
- When this happens, we first sort using the first column and if there is a tie we use the next column.

```
SELECT * FROM employees ORDER BY taxes DESC, salary ASC;  
SELECT * FROM employees ORDER BY taxes DESC, salary;      -- equivalent
```

id	name	salary	taxes	num	d_name
3	John Smith	1500	350	2	Sales
1	John Doe	700	200	1	Marketing
4	Jane Roe	1000	200	3	Production
2	Jane Doe	800	100	2	Sales

Limiting Data

LIMIT

The number of rows returned by a query can be limited using the **LIMIT** clause.

```
SELECT * FROM employees ORDER BY salary LIMIT 2;
```

id	name	salary	taxes	num	d_name
3	John Smith	1500	350	2	Sales
4	Jane Roe	1000	200	3	Production

OFFSET

When using the **LIMIT** clause, we can also indicate how many rows to skip using the **OFFSET** clause.

```
SELECT * FROM employees ORDER BY salary LIMIT 2 OFFSET 1;
```

id	name	salary	taxes	num	d_name
4	Jane Roe	1000	200	3	Production
2	Jane Doe	800	100	2	Sales

? Here the first line (with a salary of 1500) was skipped.

Pagination

LIMIT and OFFSET can be used to paginate our results.

? Example for page 3 with 10 results per page.

```
SELECT * FROM employees ORDER BY id LIMIT 10 OFFSET 20;
```

? Generically for page N with R results per page.

```
SELECT * FROM employees ORDER BY id LIMIT R OFFSET R * (N - 1);
```

A Complete Query

A Complete Query

- A query will always start with the **SELECT** clause.
- All the other clauses are optional.
- But they always follow the same order.

```
SELECT c1, c2, SUM(c3), AVG(c4) AS c5
FROM t1 JOIN
    t2 ON t1.foreign_key = t2.primary_key JOIN
    t3 USING (join_column)
WHERE condition_1 AND (condition_2 OR condition_3)
GROUP BY c1, c2
HAVING AVG(c4) > 10
ORDER BY AVG(c4)
LIMIT 5 OFFSET 10
```


Text Operators

LIKE

The LIKE operator can be used to compare strings using simple patterns.

- The % particle means zero or more characters
- The _ character means exactly one character.

id	name
1	John Doe
2	Jane Doe
3	Jean Doe
4	Jennifer Doe
5	William Doe

```
SELECT * FROM people WHERE name LIKE '% Doe'      -- All five rows
SELECT * FROM people WHERE name LIKE 'J% Doe'      -- John, Jane, Jean and Jennifer
SELECT * FROM people WHERE name LIKE 'J__ Doe'     -- John, Jane and Jean
SELECT * FROM people WHERE name LIKE '_e%'         -- Jean and Jennifer
```

? The ILIKE operator is similar but also ignores case.

SIMILAR TO

To be done

POSIX Regular Expressions

To be done

Nested Queries

Queries as Tables

The result of a query is a table. This allows us to use queries in the same places we use tables.

```
SELECT *  
FROM (SELECT * FROM  
      employees JOIN  
      department USING(d_num)  
      WHERE department.name = 'Logistics') AS logistic_employees JOIN  
works ON logistic_employees.id = works.id  
WHERE works.hours > 10
```

? *Subqueries must be named using the AS particle.*

Subquery Expressions

Subquery expressions can be used to combine two queries in the **WHERE** or **HAVING** clauses.

There are several subquery expressions:

```
expression IN subquery  
expression NOT IN subquery  
expression operator ANY (subquery) -- Same as SOME  
expression operator ALL (subquery)
```

IN and NOT IN

```
expression IN subquery  
expression NOT IN subquery
```

These expressions test if a value (or values) exist (or not) in the subquery.

```
SELECT *                -- Employees whose id exists in the subquery  
FROM employees  
WHERE employee.id IN (  
    SELECT emp_id        -- The ids of employees that work more  
    FROM works          -- than 10 hours in projects  
    GROUP BY emp_id  
    HAVING SUM(hours) > 10  
)
```


ANY

expression operator ANY (subquery) -- Same as SOME

The **ANY** expression compares a value with every row of the subquery and returns true if at least one of the comparisons returns true.

```
SELECT *                -- Employees that have a larger salary than at least
FROM employees          -- one logistics department employee
WHERE salary >= ANY (
    SELECT salary        -- All the salaries from logistic department employees
    FROM employees JOIN
        departments ON employees.d_num = department.num
)
```

? = ANY is the same as IN

ALL

expression operator ALL (subquery)

The **ALL** expression compares a value with every row of the subquery and only returns true if **all** of the comparisons returns true.

```
SELECT *           -- Employees that have a larger salary than all of
FROM employees     -- the logistics department employees
WHERE salary >= ALL (
    SELECT salary   -- All the salaries from logistic department employees
    FROM employees JOIN
        departments ON employees.d_num = department.num
)
```

? <> ALL is the same as NOT IN