IN3020/4020 – Database Systems
Spring 2021, Week 1.2

SQL QUERIES (SELECT and a bit more)

Egor V. Kostylev (with M. Naci Akkøk)

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

SQL

Details: https://www.postgresql.org/docs/current/sql.html

SQL is the inter-galactic data-speak with several sub-speaks:

- Data Definition Language (**DDL**):
  CREATE-queries, creating schemas
- Data Query Language (**DQL**):
  SELECT-queries
- Data Manipulation Language (**DML**):
  INSERT/UPDATE, write-queries
- Data Control Language (**DCL**):
  GRANT/REVOKE, access & user management

An example

Projection refers to that subset of the set of all columns found in a table, that you want returned. It can range anywhere from 0** up to the complete set. There are two "sets" in a table that correspond to a table's two dimensions. Each table has a set of columns as well as a set of rows. **Selection** returns rows.

```
SELECT p.Name, m.Id FROM Person p JOIN Movie m ON
      p.Id = m.ActorId
WHERE m.DirectorId = 1234;
```

- Select-project-join-query:

  ```
  SELECT attributes FROM table WHERE condition
  ```

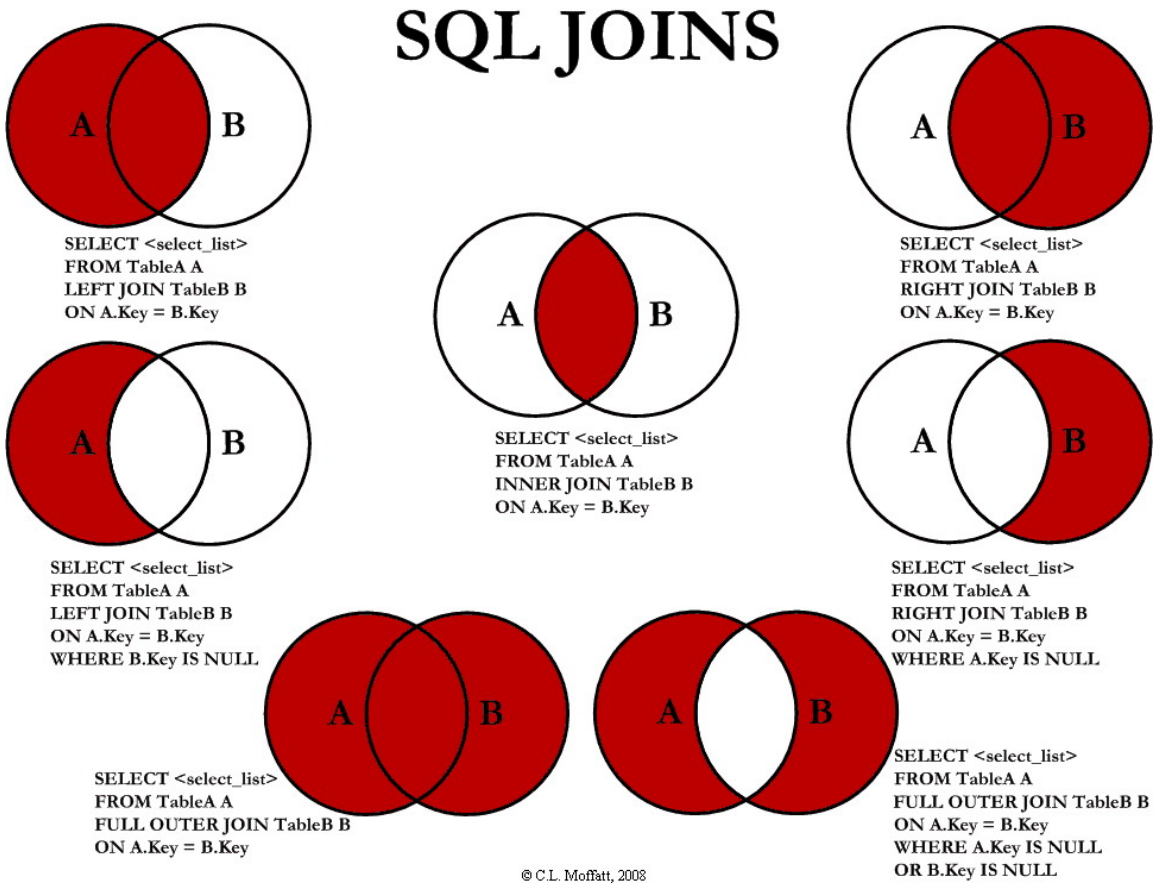
  here `[Person p JOIN Movie m ON p.Id = m.ActorId]` is a table expression

- We can create tables via joins and conditions via operators and logic

Joins

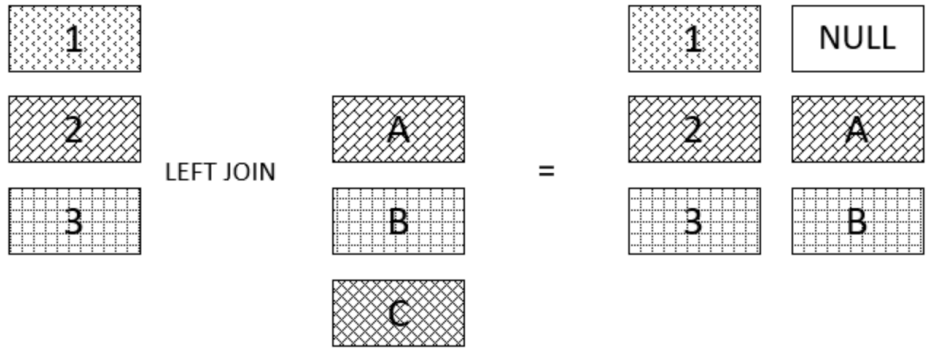
- We have `NULL` in SQL, which means that we have some extra join-types:
  - **`JOIN`** (`or` **`INNER JOIN`**)
  - **`LEFT JOIN`** (`or` **`LEFT OUTER JOIN`**)
  - **`RIGHT JOIN`** (`or` **`RIGHT OUTER JOIN`**)
  - **`FULL JOIN`** (`or` **`FULL OUTER JOIN`**)
- **`JOIN`** puts together only the tuples that satisfy the conditions
- `R` **`LEFT JOIN`** `S` **`ON`** `R.a = S.b`
  is a **`JOIN`** that keeps all tuples from `R`; those that don´t have a match in `S` get `NULL` values in all `S`-columns
- What is **`RIGHT`** and **`FULL`**?

How to understand joins



- Be careful with Venn-diagrams!
- https://blog.jooq.org/2016/07/05/say-no-to-venn-diagrams-when-explaining-joins/

The following illustrates the `LEFT JOIN` of two tables T1(1, 2, 3) and T2(A, B, C). The `LEFT JOIN` will match rows from the T1 table with the rows from T2 table using patterns:



Self-join

- We want to “print names of each employee and their manager” given `Employee(Id, Name)` and `Manager(empId, mgrId)`
- Wrong:

```
SELECT e.Name, e.Name FROM Employee e
  JOIN Manager ON empId=Id AND mgrId=Id;
```

- There is obviously something very wrong with this query. We need TWO names!

Self-join continued

- Try again: “print names of each employee and their manager” given `Employee(Id, Name)` and `Manager(empId, mgrId)`
- We need an extra copy of `Employee`:
- This is correct:

```
SELECT e.Name, s.Name FROM Employee e
JOIN Manager ON empId=e.Id
JOIN Employee s ON s.Id = mgrId;
```

Joins: The loopholes

- A not very helpful short form for JOIN:
  FROM `Table1, Table2, Table3` **WHERE** `<many many conditions>;`
- Easy to become lost;
  better to write a **JOIN ON** for each table
- Self and other joins:
  Remember that **WHERE** is evaluated on each row in the table that are created in **FROM** (same is valid for conditions in **ON**)

Conditions

- Boolean operators on values, combinable through logical operators
- In addition to columns/values, one can use scalar subqueries; i.e., a query that returns a table with one row and one column
- A loophole in case of conditions: `NULL`!
  `NULL = NULL` will evaluate to false!
- https://www.postgresql.org/docs/9.2/static/functions-comparison.html

NULL – the history

In 2009, C. A. R. Hoare said:

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

The NULL-problem

- `NULL` in an arithmetic expression that always gives `NULL` as result
- Comparison of `NULL` with a value gives “unknown” as result
  - Even `NULL` = `NULL` is false
  - Exception: **`IS DISTINCT FROM`** (which is approximately like <>, but it compares as if NULL was a value)
- It is illegal to use `NULL` explicitly as part of an expression. We can check if the result of a calculation is `NULL`:
  `X` **`IS`** `NULL`, `X` **`IS NOT`** `NULL`
- Own rules for grouping with `NULL` later

Managing NULL – Some useful concepts

- **`COALESCE`**`(Expr1, Expr2, ...)` returns first argument that is not `NULL`; and `NULL` if all are `NULL`.
- Typical use: `COALESCE(Address, 'not given')`

- **`NULLIF`**`(Expr1, Expr2)` returns `NULL` if the arguments are the same, or returns `Expr1`

More on Conditions

- **EXISTS**(*query*) — Boolean
- **SOME/ANY**(*query*) and **ALL**(*query*)

```
SELECT p.name FROM Person p WHERE p.savings > ALL
    (SELECT pq.savings FROM Person pq
        WHERE pq.age = 30));
```

- Can be used against multiple attributes via ROW:
  WHERE `ROW(A, B)` `>` **ALL** (*we need two columns here*)
- IMPORTANT: **ALL** compares one value/row at a time with a whole table. Can not be used to check multiple rows simultaneously against a table.

Let´s test our skills

- We can do a lot with what we have seen so far
- Let’s try:

```
Project(prnr, pname, customer, pmanager, startDate)
Employee(enr, name, title, bdate, pnr, empDate)
Timesheet(enr, date, prnr, hours)
Customer(cnr, canme, address)
```

- Find names and titles of all employees who worked on minimum one project that started after 2014 and was ordered by customer `ABC’

Divide-and-Conquer

```
Project(prnr, pname, customer, pmanager, startDate)
Employee(enr, name, title, bdate, pnr, empDate)
Timesheet(enr, date, prnr, hours)
Customer(cnr, cname, address)
```

Find names and titles of all employees who worked on minimum one project that started after 2014 and was ordered by customer `ABC'.

```
-- All projects with the correct conditions
WITH CorrectProjects AS (
  SELECT p.prnr FROM Project p JOIN Customer c ON p.customer = c.cnr WHERE
    p.startDate>'2014-12-31' AND c.cname = 'ABC'),
-- Who was on which projects
EmployeeOnProjects AS (
  SELECT e.name, e.title, t.prnr FROM Employee e JOIN Timesheet t ON
    t.enr = e.enr)
-- Put the two together
SELECT eop.name, eop.title FROM EmployeeOnProjects eop JOIN
   CorrectProjects cp ON eop.prnr = cp.prnr;
```

A slightly more challenging example

- Same schema

  `Project(prnr, pname, customer, pmanager, startDate)`
  `Employee(enr, name, title, bdate, pnr, empDate)`
  `Timesheet(enr, date, prnr, hours)`
  `Customer(cnr, cname, address)`

- Find names and titles of the employees working on all projects that started after 2014 and were ordered by customer `ABC'
- Tempting to try with **`ALL`**, but it won´t work
- **Hint:** ∀xP(x) ≡ ¬∃x¬P(x) ... Double negation. Not all but "any":
  We then get the employees that do not have any ...
  (correct projects they did not work in)

Still needs the "correct" projects!

Project(prnr, pname, customer, pmanager, startDate)
Employee(enr, name, title, bdate, pnr, empDate)
Timesheet(enr, date, prnr, hours)
Customer(cnr, cname, address)

```
--All projects satisfying condition
CREATE VIEW CorrectProjects AS (
     SELECT p.prnr FROM Project p JOIN Customer c ON
     p.customer = c.cnr
     WHERE p.startDate>'2014-12-31'
     AND  c.cname = 'ABC'
);
```

Find name and title of the employees working on all projects that started after 2014 and were for customer 'ABC'

Solution

```
Project(prnr, pname, customer, pmanager, startDate)
Employee(enr, name, title, bdate, pnr, empDate)
Timesheet(enr, date, prnr, hours)
Customer(cnr, cname, address)
```

```
--Employees that do NOT have any (correct project they have NOT worked on)
SELECT e.name, e.title FROM Employee e WHERE NOT EXISTS (
SELECT * FROM CorrectProjects cp WHERE NOT EXISTS (
SELECT * FROM Timesheet t WHERE t.prnr = cp.prnr AND t.enr = e.enr));
```

- Innermost `SELECT` is executed again for each employee and project:
  Called a correlated query (subquery, i.e., query nested in another query) or synchronized subquery
- NOTE: Because the **subquery** may be evaluated once for each row processed by the outer **query**, it can be slow
- Could have used other mechanisms like counting, LEFT JOIN etc.

Find name and title of the employees working on all projects that started after 2014 and were for customer 'ABC'

Aggregation

- `COUNT`, `SUM`, etc. Example:

```
SELECT p.Name, COUNT(m.id) AS mn FROM Person p JOIN Movie m ON
p.Id = m.ActorId
WHERE m.DirectorId = 1234 GROUP BY p.Id, p.Name HAVING mn>10;
```

- Aggregates accumulate many tuples (which?) into one tuple
- **`SELECT-FROM-WHERE`** is executed/calculated first
- Then, **`GROUP BY`** is read, and the table is divided into one group for each value with respect to whatever is in **`GROUP BY`**
- Then, aggregation is calculated for each group → giving a new table, one row per group
- And then the whole thing is filtered through the HAVING condition clause

Consequences of the aggregation semantics

- If we have non-aggregates in **SELECT**, we need to have all those in **GROUP BY** (exception since pgSQL9.1: If PK is in **SELECT**)
- Aggregates can not be used in **WHERE**
- Without **GROUP BY**, we get maximum 1 row: aggregates the whole table. Seldom correct!

```
SELECT p.Name, COUNT(m.id) AS mn FROM Person p JOIN
Movie m ON p.Id = m.ActorId
WHERE m.DirectorId = 1234 GROUP BY p.Id, p.Name HAVING
mn>10;
```

Grouping example

Project(prnr, pname, customer, pmanager, startDate)
Employee(enr, name, title, bdate, pnr, empDate)
Timesheet(enr, date, prnr, hours)
Customer(cnr, cname, address)

```
--All projects satsifying condition
WITH CorrectProjects AS (
   SELECT p.prnr FROM Project p JOIN Customer c ON p.customer = c.cnr
       WHERE p.startDate>'2014-12-31' AND c.cname = 'ABC'
   ),  UniqueEmployeesOnCorrectProjects AS (
   --Who was on the correct projects
   SELECT DISTINCT e.enr, e.name, e.title, t.prnr FROM Employee e JOIN
   Timesheet t ON t.enr = e.enr JOIN CorrectProjects cp ON
   cp.prnr = t.prnr
   )
--Then we count number of correct projects per employee
SELECT uecp.name, uecp.title FROM UniqueEmployeesOnCorrectProjects uecp
GROUP BY uecp.enr, uecp.name, uecp.title
HAVING COUNT(uecp.prnr) = (SELECT COUNT(*) FROM CorrectProjects);
```

Issues with NULL in aggregation

- `NULL`-values are ignored in aggregation
  - Important in the case of `COUNT`
  - Exception is `COUNT(*)`

- Most aggregation functions will return `NULL` if evaluated over «no values»

SQL – The general form

```
SELECT [DISTINCT] <attributelist>
FROM <tableexpression>
[WHERE <where-conditions>]
    [GROUP BY <groupingattributes>
    [HAVING <aggregation-condistions>]
]
[ORDER BY <attribute> [asc | desc][,
<attribute> [asc | desc] ] ... ];
```

Recursive SQL

- Not really recursive but fixpoint-SQL
- Allows us to run a query Q on the result of the query Q repeatedly – over and over again
- Is used to build (nest-up) graph-like data

```
WITH RECURSIVE <tablename>(<attributelist>) AS (
      <non-recursive term>
      UNION | UNION ALL -- pick one
      <recursive term>
)
SELECT * FROM <tablename>;
```

Recursive SQL vs. Recursive Functions

- A recursive function is a function that calls itself
- Recursive SQL-query evaluates a query over and over again from its own output table
- If, in the recursive part, we join something with this table, it will grow

Recursive SQL Example

Given `Flight(fromCity, toCity, carrier, flightnr, price)`, find all cities one can travel to from NYC or Chicago with maximum 3 intermediate stops (connections). Return number of connections and total price.

```
WITH RECURSIVE Destination(from, to, nrConn, totPrice) AS (
 --Start-table as result of this query
SELECT f.fromCity, f.toCity, 0, f.price
FROM Flight f

WHERE f.fromCity = 'New York' OR f.fromCity = 'Chicago'
UNION ALL
-- Recursive term on Destination and inserts new tuples there!

SELECT d.from, g.toCity, d.nrConn + 1, d.totPrice + g.price FROM
Destination d JOIN Flight g ON d.to = g.fromCity
WHERE d.nrConn < 3

)
SELECT * FROM Destination;
```

Recursive SQL – The semantics

- Let `Init` and `Rec` be the non-recursive and recursive parts respectively (both queries)
- `Init` is evaluated first; result goes into the table with the name we have chosen (`Destination` in the example)
- `Rec` will then be evaluated form `Destination;`
  If it returns tuples that were not originally in `Destination`, they will be inserted, and `Rec` will be evaluated again; and over and over again, until nothing new is found

Array in recursive SQL queries

(handy for managing loops)

```
WITH RECURSIVE Destination(from, to, cities, nrConn, totPrice) AS
(
  SELECT f.fromCity, f.toCity, array[f.fromCity, f.toCity], 0, f.price
  FROM Flight f
  WHERE f.fromCity = 'New York' OR f.fromCity = 'Chicago'
  UNION ALL
  SELECT d.from, g.toCity, d.cities || g.toCity, d.nrConn + 1,
     d.totPrice + g.price
  FROM Destination d, Flight g
  WERE d.to = g.fromCity AND g.toCity <> ALL(d.cities)
)
SELECT from, to, nrConn, totPrice FROM Destination;
```

More on arrays in few minutes