IN3020&4020 – Database Systems (2024)
Part: Intro & Recap
Lecture 4: SQL Summary – 2

---

23 January

Egor V. Kostylev
IFI, University of Oslo
egork@ifi.uio.no



Materials to read

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapters 6&7
2. *Database systems – the complete book* by H. Garcia-Molina, J.D. Ullman, and J. Widom (2nd ed.): Section 6
3. PostgreSQL 14 tutorial: Part II

   `https://www.postgresql.org/docs/current/sql.html`
4. IN2090 – Databaser og datamodellering (Leif Harald Karlsen): Lectures 5, 6, 10, 11

   `https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/`
5. etc.

The difference between concrete SQL systems and versions are not essential today: we look at basics

Purpose of the lecture

I assume that you know SQL to some extent

- IN2090 (or equivalent) is a prerequisite

Knowledge of all details (esp., practical) are not important for us

- we learn the principles of how SQL works
- not the details of SQL usage

Today's lecture is largely a sync-up

- by no means a comprehensive SQL tutorial
- if you do not understand what I am talking about,
  come back to IN2090 (not much really)
- you can play with PostgreSQL (group sessions for details)

But also has material beyond IN2090

Lecture plan

1. Data Model of SQL (last time)
2. SQL Intro (last time)
3. SQL DDL and DML (last time)
4. SQL DQL
   - query syntax and semantics
   - constructs and keywords
   - basic and advanced examples
   - ...

4. SQL Data Query Language

4.1. Overview

Data Query Language (DQL): basics

DQL:

- the most important part of SQL (and whole DBMS)
- saying SQL we often mean the DQL sub-language
- realised via `SELECT` queries with basic syntax

```
SELECT <attribute list>
FROM <table>
[ WHERE <condition> ] ...
```

- '<table>' is a relation name or another query (but we do not need to write the whole relation here, only the name)
- '<attribute list>' is a list of attributes in the schema of the relation referred by '<table>'

Question: What is the semantics?

Answer (for SQL): SQL query inputs a database (set of relations) and outputs a bag(!) relation according to some rules (see below)

Query example

Schema (underline is a primary key here):

Tutorials [ID, site, tutorial, topic]
Students [name, tutorialID, startdate]

Query (select-project-join):

```
SELECT s.name, t.tutorial                ← projection
FROM Students s JOIN Tutorials t         ← join
     ON s.tutorialID = t.ID
WHERE t.topic = 'Databases';             ← selection
```

retrieves all student-tutorial pairs on topic databases

4.2 SELECT clause

SELECT clause comments

```
SELECT s.name, t.tutorial
...
```

- Selects columns from the (bag) relation constructed below
- Corresponds to projection in relational algebra (see next lectures), so s.name, t.tutorial are called projected attributes
- Relation name qualification (s., etc.) may be omitted if no ambiguity
- Shortcut * may be used for all attributes
- By default, returns a bag relation (with possible duplicates)
- DISTINCT can be used for duplicate elimination
- ORDER BY, LIMIT, etc. can be used at the end of the query for arranging tuples in the output
- Can do value invention:

```
SELECT s.ID + 5, d AS Day(t.startdate)
```

3.3. FROM clause

FROM clause comments

```
...
FROM Students s JOIN Tutorials t ON s.tutorialID = t.ID
...
```

- constructs a relation from other relations
- in general, can contain an arbitrary query (e.g., another `SELECT` query)
- in the most simple case, `FROM` clause is a relation name
- (possibly nested) operations can be used: `JOIN`, `UNION`, `INTERSECT`, etc.
- the most common is `JOIN` operation (and its variations)

JOIN (syn. INNER JOIN): example

Combines tuples from two relations in all possible ways that agree on declared attributes

```
Tutorials [INNER] JOIN Topics ON Tutorials.topicID = Topics.ID
```

on

Tutorials:

ID	*site*	*tutorial*	*topicID*
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
3	w3schools	CSS_3	3
4	w3resource	SQL_2003STD	1

Topics:

ID	*topic*
1	Databases
2	WebDev

gives

Tutorials ⋈ **Topics**:

Tutorials.ID	*site*	*tutorial*	*topicID*	*Topics.ID*	*topic*
1	w3schools	SQL_2003STD	1	1	Databases
2	w3schools	HTML_5	2	2	WebDev
4	w3resource	SQL_2003STD	1	1	Databases

OUTER JOINs : example

Combines tuples from two relations in all possible ways that agree on declared attributes; if left (or right) does not match anything, filled with nulls

```
Tutorials LEFT [OUTER] JOIN  Topics ON Tutorials.topicID=Topics.ID
Topics RIGHT [OUTER] JOIN Tutorials ON Tutorials.topicID=Topics.ID
```

on

Tutorials:

ID	*site*	*tutorial*	*topicID*
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
3	w3schools	CSS_3	3
4	w3resource	SQL_2003STD	1

Topics:

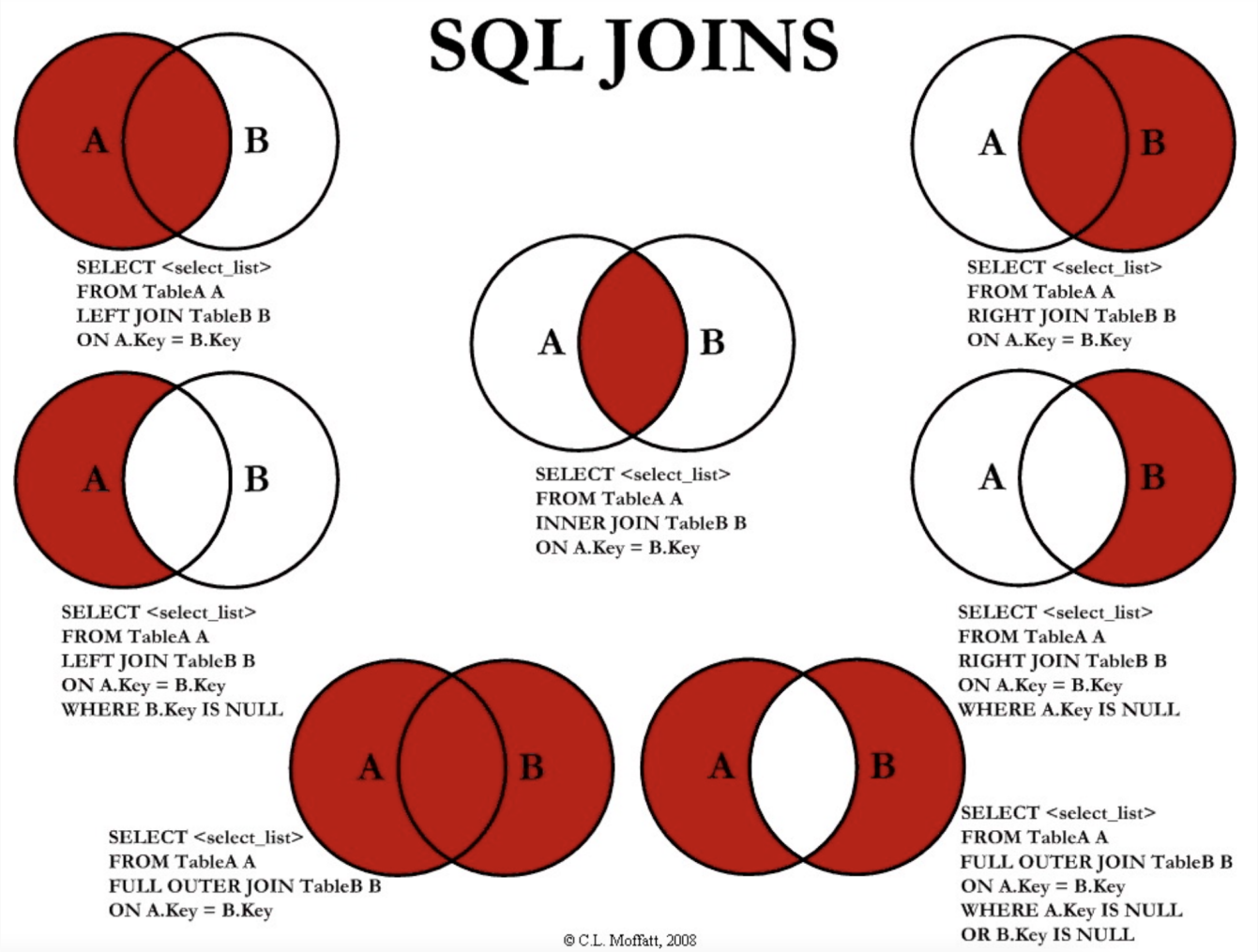
ID	*topic*
1	Databases
2	WebDev
25	Asronomy

gives

Tutorials ⋈ **Topics**:

Tutorials.ID	*site*	*tutorial*	*topicID*	*Topics.ID*	*topic*
1	w3schools	SQL_2003STD	1	1	Databases
2	w3schools	HTML_5	2	2	WebDev
3	w3schools	CSS_3	3	NULL	NULL
4	w3resource	SQL_2003STD	1	1	Databases

Joins (confusing) visualisation



Question: What does the area in the circles represent?

Self-join exercise

It is common to join a table with itself (directly or indirectly).

Schema: Employees[Id, Name] and Managers[empId, mgrId]

Query: 'Give the names of each employee and his/her manager'

A (very) naive wrong solution:

```
SELECT e.Name, e.Name
FROM Employees e JOIN Managers
    ON empId=Id AND mgrId=Id;
```

We need two 'copies' of Employees

Self-join exercise

It is common to join a table with itself (directly or indirectly).

Schema: Employees[Id, Name] and Managers[empId, mgrId]

Query: 'Give the names of each employee and his/her manager'

A correct solution:

```
SELECT e.Name, s.Name
FROM Employees e JOIN Managers
    ON empId=e.Id JOIN Employees s
        ON s.Id = mgrId;
```

Set- (bag-) theoretic operations

Relations (i.e., tables) are sets (or bags) of tuples

We can apply usual set-theoretic operations to them:

- set-theoretic (duplicates eliminated): `UNION`, `EXCEPT`, `INTERSECT`
- bag-theoretic (duplicates preserved): `UNION ALL`, `EXCEPT ALL`, `INTERSECT ALL`

Example:

```
SELECT * FROM Employees UNION Interns;
```

Notes:

- complex queries can be used as arguments
- arguments must have the same schema

Subqueries

We can use subqueries in the `WHERE` clause:

```
SELECT ... FROM
    ... (SELECT ... FROM Table WHERE ...)  ...;
```

Subqueries can be correlated:

```
SELECT * FROM Tab1 JOIN
  (SELECT Tab2.a FROM Tab2 WHERE Tab1.b+5 > Tab2.c) ON Tab1.a=Tab2.a;
```

Silly query in this case, may be a lot more difficult

Often rewritable into usual joins, but not always

Homework: Try to come to an example

Hint: play with bags

WITH-subqueries and views

Subqueries can be used inside other queries

(we will see more examples)

We can give them names for convenience:

```
WITH SubqueryName1 AS (SELECT ...),
     SubqueryName2 AS (SELECT ...)
SELECT ... SubqueryName1 ... SubqueryName1 ... ;
```

If a subquery is used in several queries, we can give it a name only once—that is, create a virtual relation:

```
CREATE VIEW View1 AS (SELECT ...);
```

We can then use View1 in other queries as usual tables (relations)

But no table is created in reality, it is still just a name of a subquery

Recursive SQL

Typical recursive query (<reqursive-query> can use `Relation`):

```
WITH RECURSIVE Relation (Att1, ..., Attn) AS (
    <non-reqursive-query>   UNION [ALL]   <reqursive-query>
) SELECT ... FROM Relation ...;
```

Evaluation procedure:

1. Relation is initialised with the evaluation of <non-reqursive-query>
2. <reqursive-query> is evaluated on the current state of Relation, union is taken the current state of Relation, and the result is loaded into Relation
3. Step 2 is repeated until there is no change (i.e., fix-point state is reached)
4. Outer SELECT is evaluated on the fix-point state of Relation

Recursive SQL

Typical recursive query (<reqursive-query> can use `Relation`):

```
WITH RECURSIVE Relation (Att1, ..., Attn) AS (
    <non-reqursive-query>   UNION [ALL]   <reqursive-query>
) SELECT ... FROM Relation ...;
```

Observations:

- corresponds to fix-point extension of first-order logic
- cannot be expressed via other operators
- Relation is getting bigger (more tuples) at each step

- Homework question: Does this process always stop?

Recursive SQL (classic) example

Schema: FlightRoutes[fromCity, toCity]

Query: Find all cities you can fly to from Oslo (with any number of flight changes)

Solution:

```
WITH RECURSIVE Destinations (destCity) AS (
    SELECT f.toCity, FROM FlightRoutes f
        WHERE f.fromCity = 'Oslo'
    UNION
    SELECT rf.toCity FROM Destinations d
        JOIN FlightRoutes rf ON d.destCity = rf.fromCity
) SELECT * FROM Destinations;
```

Question: Why does this always terminate? Answer: Outputs only cities mentioned in the database

Cannot be done with usual join queries, because the number of stops is not bound in advance

3.4. WHERE clause

WHERE clause comments

```
...
WHERE t.topic = 'Databases';
```

- Selects tuples (rows) from the relation constructed in the FROM clause according to the condition in the clause
- Corresponds to selection in relational algebra
- The condition can mention both constant values (e.g., 'Databases'), which are the same for every tested tuple, and attributes (e.g., t.topic), which are instantiated by the attribute value for each tuple
- Values can be combined to expressions by functions (+, external, etc.), Boolean operators (<, `LIKE`, etc.), logical operators (`AND`, `OR`, `NOT`), case expressions (`CASE WHEN ... THEN ... ELSE ... END`), etc.
- The condition can also mention subqueries (see below)
- Special care should be given to NULL (next slide)

The NULL problem

NULL in an value with special properties

- It is illegal to use NULL explicitly as part of an expression
- Usual comparison of NULL with any value evaluates to FALSE
- Both `NULL = NULL` and `NOT (NULL = NULL)` are FALSE
  (formally, three-valued logic is involved)
- Special comparison operators can be used:
  `X IS NULL`, `X IS NOT NULL`, `X IS DISTINCT FROM Y`
  (work with NULL as expected)
- Own rules for grouping & aggregation with NULL (see below)

Managing NULL: some useful concepts

`COALESCE(`Expr1, Expr2, ...`)` returns the 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 evaluate the same, or Expr1 otherwise

Conditions with subqueries: EXISTS expression

`EXISTS(SELECT ...)`: Boolean (TRUE/FALSE) condition that checks existence of answers of the subquery

- evaluates to TRUE if the query gives non-empty result

Example:

```
SELECT p.name FROM Persons p
    WHERE EXISTS(SELECT ...);
```

Question: Does it look a bit silly?
Can you come to an example of `...` so the query makes sense?

Answer: The subquery can be correlated with the outside
(as usual subqueries)

```
SELECT p.name FROM Persons p
    WHERE EXISTS(SELECT * FROM Addresses a WHERE p.name = a.name);
```

Conditions with subqueries: SOME

`SOME(SELECT ...)` (syn.: `ANY(SELECT ...)`) looks for matching values in the subquery:

- example query (best explanation)

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

- meaning: all people with savings more than at least one 30-years-old person

Can be used against multiple attributes via `ROW`:

`... WHERE ROW(Tab.a,Tab.b) > SOME(` two-column table here `)`

`IN` is a special case of `SOME`:

- `... IN(SELECT ...)` synonym of `... = SOME(SELECT ...)`

Conditions with subqueries: ALL

From the previous slide for comparison:

- example query:

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

- meaning: all people with savings more than at least one 30-years-old person

`ALL(SELECT ...)` 'symmetric' to `ALL`:

- example query (best explanation):

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

- meaning: all people with savings more than all 30-years-olds

Test our skills

Schema: Projects [prID, prName, customerID, prStartDate, . . .]
Customers [cID, cName, cAddress, . . .]
Employees [eID, eName, eTitle, eStartDate, . . .]
Timesheets [eID, tDate, prID, hours, . . .]

Query: Find names and titles of all employees who worked on at least one project started after 2014 and was ordered by customer 'Alice'

'Divide-and-Conquer' solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID
          WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Who was on which projects

```
WITH EmployeeOnProjects AS (
SELECT e.eName,e.eTitle,t.prID FROM Employees e JOIN Timesheets t ON t.eID=e.eID)
```

- Put the two together

```
SELECT eop.eName,eop.eTitle
FROM EmployeeOnProjects eop JOIN CorrectProjects cp ON eop.prID=cp.prID;
```

Test our skills: more challenging

Schema: Projects [prID, prName, customerID, prStartDate, . . .]
Customers [cID, cName, cAddress, . . .]
Employees [eID, eName, eTitle, eStartDate, . . .]
Timesheets [eID, tDate, prID, hours, . . .]

Query: Find names and titles of all employees who worked on all projects started after 2014 and was ordered by customer 'Alice'

Solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID
        WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Employees that do not have any (correct project they did not worked on):

```
 SELECT e.eName,e.eTitle FROM Employees e WHERE NOT EXISTS (
   SELECT * FROM CorrectProjects cp WHERE NOT EXISTS (
     SELECT * FROM Timesheets t WHERE t.prID=cp.prID AND t.eID=e.eID));
```

Test our skills: more challenging

Query: Find names and titles of all employees who worked on all projects started after 2014 and was ordered by customer 'Alice'

Solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID
        WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Employees that do not have any (correct project they did not worked on):

```
 SELECT e.eName, e.eTitle FROM Employees e WHERE NOT EXISTS (
   SELECT * FROM CorrectProjects cp WHERE NOT EXISTS (
     SELECT * FROM Timesheets t WHERE t.prID=cp.prID AND t.eID=e.eID));
```

Observations:

- Innermost SELECT is executed again for each employee and each project (twice correlated query)
- Thus, can be slow
- Could use other mechanisms (counting, LEFT JOIN, etc.)

3.5. Aggregation

Aggregation

A mechanism to accumulate values from many tuples into one using an aggregation function (`COUNT`, `COUNT(*)`, `MAX`, `MIN`, `SUM`, etc.)

Example:

```
SELECT p.Name, COUNT(m.Id) AS mCount
FROM Persons p JOIN Movies m ON p.Id=m.ActorId WHERE m.DirectorId=1234
GROUP BY p.Id, p.Name
     HAVING mCount > 10;
```

Question: What is the difference between `WHERE` and `HAVING`?

Evaluated in steps:

1. FROM-WHERE (i.e., the query 'inside', without SELECT)
2. the tuples divided into groups, one group for each tuple of grouping attributes (i.e., those in GROUP BY)
3. aggregated values are calculated for each group, giving a new table with one tuple per group
4. the whole thing is filtered through the HAVING condition clause
5. the attributes in SELECT are projected

Aggregation

A mechanism to accumulate values from many tuples into one using an aggregation function (`COUNT`, `COUNT(*)`, `MAX`, `MIN`, `SUM`, etc.)

Example:

```
SELECT p.Name, COUNT(m.Id) AS mCount
FROM Persons p JOIN Movies m ON p.Id=m.ActorId WHERE m.DirectorId=1234
GROUP BY p.Id, p.Name
     HAVING mCount > 10;
```

Question: What is the difference between `WHERE` and `HAVING`?

Comments on aggregation:

- all non-aggregates in the SELECT clause need to be in the GROUP BY clause
- aggregates can not be used in WHERE (only in HAVING)
- without GROUP BY, we always have exactly 1 row: aggregates for the whole table

NULLs in aggregation

NULL-values are ignored in aggregation

- even in the case of `COUNT`
- exception is `COUNT(*)`

Aggregation functions will return NULL if evaluated over an empty set of values (after the NULLs are eliminated)

- exception is `COUNT` and `COUNT(*)`

Test our skills: more challenging with aggregates

Schema: Projects [prID, prName, customerID, prStartDate, . . .]
Customers [cID, cName, cAddress, . . .]
Employees [eID, eName, eTitle, eStartDate, . . .]
Timesheets [eID, tDate, prID, hours, . . .]

Query: Find names and titles of all employees who worked on all projects started after 2014 and was ordered by customer 'Alice'

Solution with aggregates:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID
        WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Who was on correct projects:

```
WITH UniqueEmployeesOnCorrectProjects AS (
SELECT DISTINCT e.eID,e.eName, e.eTitle,t.prID FROM Employee e
 JOIN Timesheets t ON t.eID=e.eID JOIN CorrectProjects cp ON cp.prID=t.prID
```

- Put the two together:

```
SELECT uecp.eName, uecp.eTitle FROM UniqueEmployeesOnCorrectProjects uecp
 GROUP BY uecp.eID, uecp.eName, uecp.eTitle
 HAVING COUNT(uecp.prID) = SOME(SELECT COUNT(*) FROM CorrectProjects);
```

What have we learned?

A summary of the main database query language:

SQL