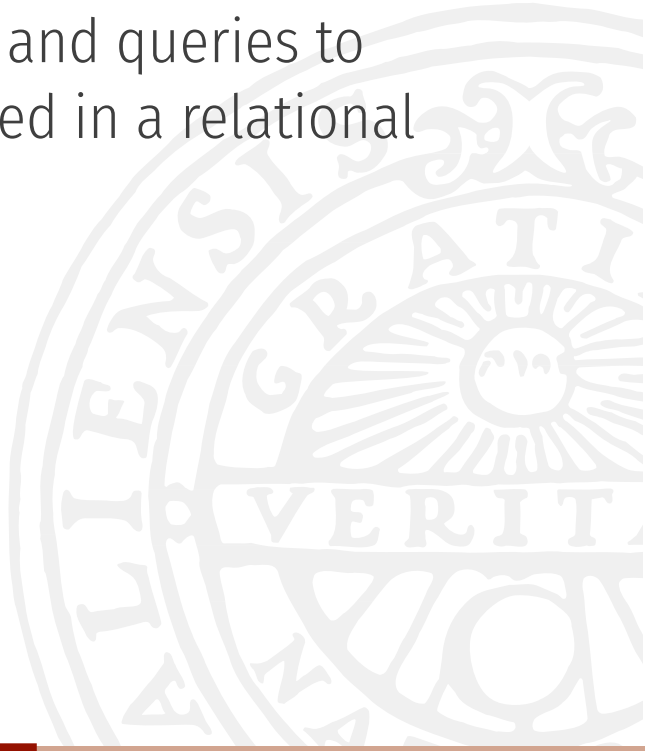
SQL

1DL301 Database Design I

INTENDED LEARNING OUTCOME

After completing all three lectures on SQL you should

- know the syntax and semantics of SQL, and
- be able to write and execute SQL statements and queries to retrieve, create, modify and remove data stored in a relational database.



WHAT IS SQL?

SQL—Structured Query Language—is a special-purpose language designed for querying and managing data in relational databases.

SQL is an ANSI (American National Standard Institute) standard.

A SIMPLE SELECT

```
1 SELECT attribute [, ...]
2 FROM table_reference [, ...]
3 [WHERE where_condition]
```

- `attribute` – specifies a column in the result,
- `table_reference` – the name of the table,
- `where_condition` – the condition that selected (returned) rows must satisfy.

No WHERE clause means all rows (= `WHERE TRUE`).

Attributes shown in the result and in the condition must be from the specified table(s), but the condition might use attributes which are not included in the result

SIMPLE EXAMPLES INVOLVING A SINGLE TABLE

Relation employee(id, first_name, last_name, hour_salary, department_id).

```
SELECT first_name, last_name
FROM employee
```

```
SELECT first_name, last_name
FROM employee
WHERE department_id = 3
```

It is possible to use * instead of explicitly writing all attributes:

```
SELECT *
FROM employee
WHERE last_name = "Smith"
```

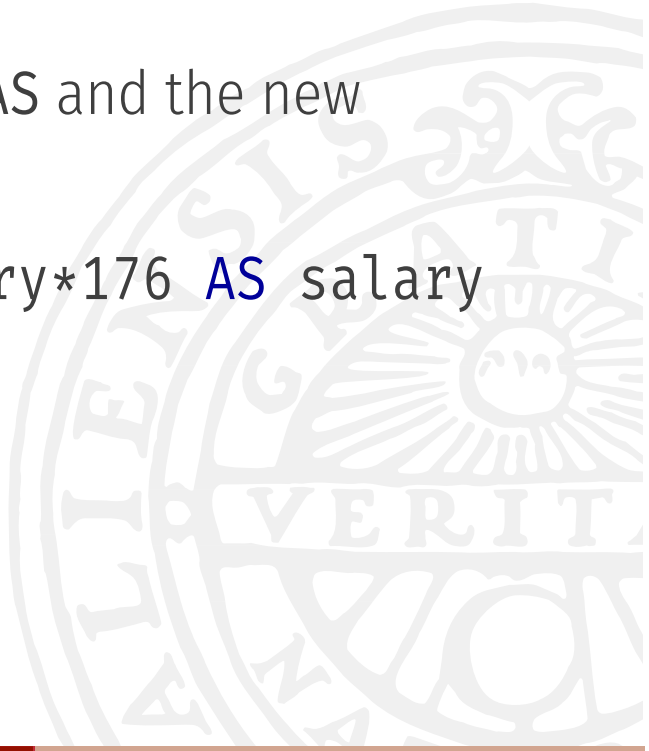
SIMPLE EXAMPLES INVOLVING A SINGLE TABLE, CONT'D

Instead of an attribute we can also use arithmetic expressions (involving attributes and/or constants):

```
1 SELECT first_name, last_name, hour_salary*176
2 FROM employee
```

We can rename a column in the result by adding AS and the new name after the attribute name or the expression:

```
1 SELECT first_name, last_name, hour_salary*176 AS salary
2 FROM employee
```



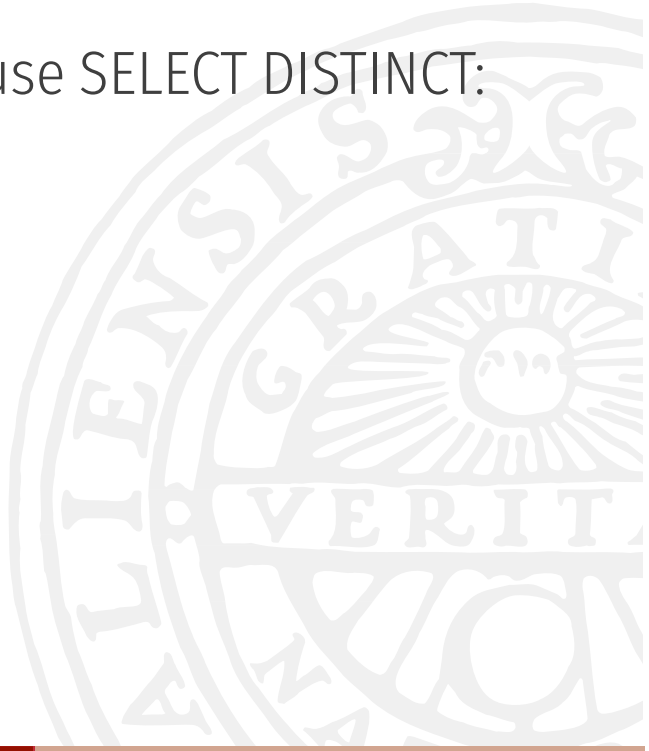
SELECT DISTINCT

Departments with the hour salary higher than 230:

```
1 SELECT department_id
2 FROM employee
3 WHERE hour_salary > 230
```

To remove duplicate rows from the result, we can use SELECT DISTINCT:

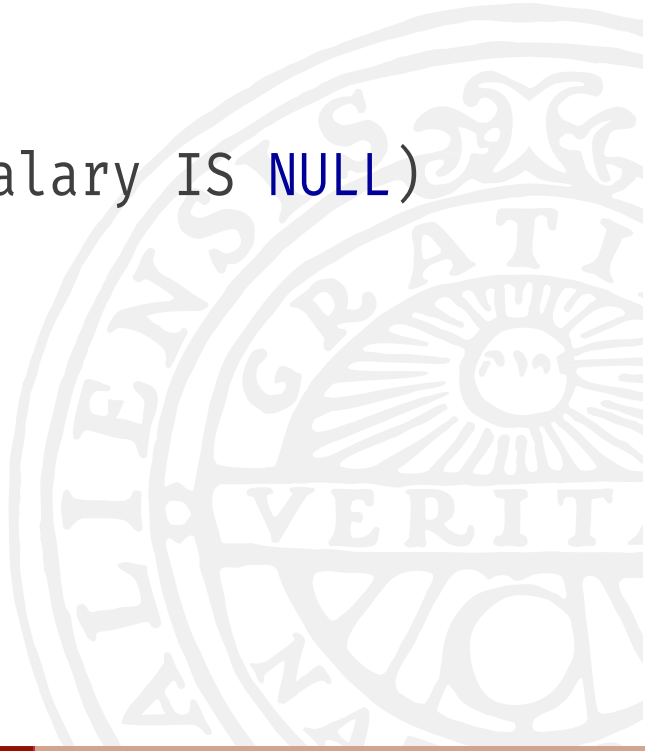
```
1 SELECT DISTINCT department_id
2 FROM employee
3 WHERE hour_salary > 230
```



WHERE CLAUSE

Logical connectives AND, OR and NOT can be used to create a more complex `where_condition`. The condition may also contain arithmetic expressions involving attributes and/or constants:

```
1 SELECT first_name, last_name
2 FROM employee
3 WHERE (hour_salary*76 < 30000 OR hour_salary IS NULL)
4   AND department_id = 2
```



IS [NOT] NULL

Some special operators (in addition to =, <, <=, >, >=, <>):

- `IS NULL` – the value is not specified/known, e.g., `hour_salary IS NULL`
- `IS NOT NULL` – the attribute has a concrete value

Never use `= NULL` nor `<> NULL`!!!

```
1 SELECT first_name, last_name
2 FROM employee
3 WHERE hour_salary = NULL
```

OTHER OPERATORS

- `[NOT] BETWEEN ... AND ...` – the value is [not] between the given values, e.g.,
  `hour_salary BETWEEN 100 AND 200`
- `[NOT] IN` – true if the value is [not] in the given set, e.g.,
  `department_id IN (1, 2, 3)`
- `[NOT] LIKE pattern` – [not] matching values against the given pattern with wildcards:
  - `_` = any character
  - `%` = any substring

  For example, `last_name LIKE "K%"` matches all rows where the *last name* starts with K

Note: This is not an exhaustive list.

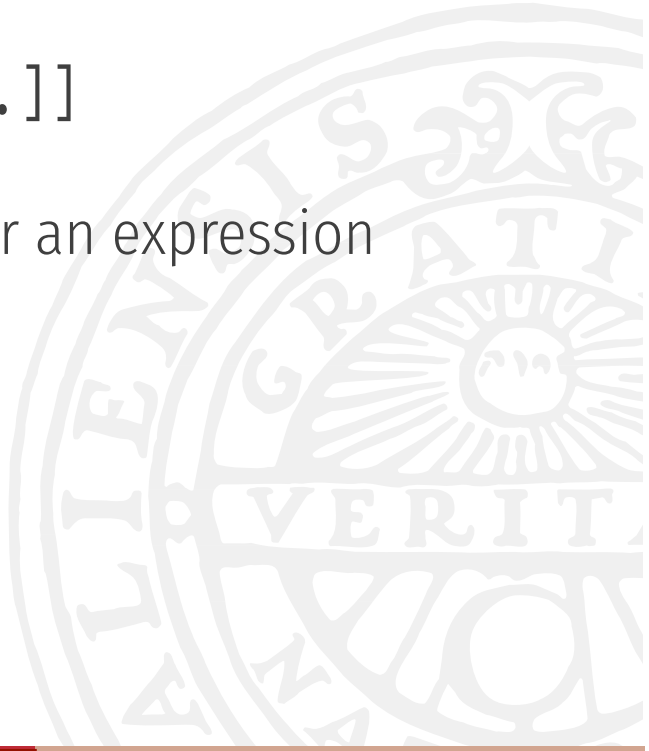
ORDER BY CLAUSE

Order of rows in the result can be given by adding the `ORDER BY` clause:

```
SELECT select_expr [, ...]
FROM table_reference [, ...]
[WHERE where_condition]
[ORDER BY order_expr [ASC | DESC] [, ...]]
```

`order_expr` can be an attribute, a column alias or an expression with attributes.

Default order is ascending (ASC).



ORDER BY CLAUSE, EXAMPLES

Sort by the last name, in descending order:

```
SELECT *
FROM employee
ORDER BY last_name DESC
```

Sort employees by their whole name:

```
SELECT id, first_name, last_name, hour_salary
FROM employee
ORDER BY last_name, first_name
```

RELATIONAL PRODUCT BY AN EXAMPLE

A_1	A_2
1	2
3	4

$\times$

B_1	B_2
α	β
γ	δ

$=$

A_1	A_2	B_1	B_2
1	2	α	β
1	2	γ	δ
3	4	α	β
3	4	γ	δ

RELATIONAL PRODUCT BY AN EXAMPLE

Table *employee*:

id	last_name	first_name	department_id
1	Alnes	Bernt	1
2	Fjeldal	Mads	1
5	Nymo	Ingvar	2

Table *department*:

id	title
1	Planning
2	Production A

`department_id` is a FK referencing `department(id)`

```sql
SELECT *
FROM employee, department
```

id	last_name	first_name	department_id	id	title
1	Alnes	Bernt	1	1	Planning
1	Alnes	Bernt	1	2	Production A
2	Fjeldal	Mads	1	1	Planning
2	Fjeldal	Mads	1	2	Production A
5	Nymo	Ingvar	2	1	Planning
5	Nymo	Ingvar	2	2	Production A

INNER JOINS

We want to show the id, first name, last name and department's title for all employees.

Idea: Do the relational product and take only the rows where employee's department_id is equal to department's id (a "join condition"):

```
1 SELECT employee.*, department.title
2 FROM employee, department
3 WHERE employee.department_id = department.id
```

id	last_name	first_name	department_id	title
1	Alnes	Bernt	1	Planning
2	Fjeldal	Mads	1	Planning
5	Nymo	Ingvar	2	Production A

TABLE ALIASES

If an attribute's name is unique (no other involved table has an attribute with the same name), the table reference can be dropped:

```
1 SELECT employee.*, title
2 FROM employee, department
3 WHERE department_id = department.id
```

Tables can be referenced by using aliases:

```
1 SELECT e.*, d.title
2 FROM employee AS e, department AS d
3 WHERE e.department_id = d.id
```

Note: AS can be dropped for table aliases.

If a query joins a table with the same table, using aliases is the only way how to reference the tables!

ALTERNATIVE SYNTAX FOR JOINS

Alternative syntax for joins:

```
1 SELECT employee.*, department.title
2 FROM employee
3 JOIN department ON employee.department_id = department.id
```

Conditions not specifying how the join is performed remain in the WHERE clause, e.g.,

```
1 SELECT employee.*, department.title
2 FROM employee
3 JOIN department ON employee.department_id = department.id
4 WHERE employee.last_name LIKE 'K%'
```

REMINDER: CREATE TABLE

```sql
CREATE TABLE Employee (
    Id int NOT NULL PRIMARY KEY,
    SSN char(10) NOT NULL UNIQUE,
    First_Name varchar(255) NOT NULL,
    Last_Name varchar(255) NOT NULL,
    Salary decimal(10, 2)
)
```

NOT NULL UNIQUE can be used for candidate keys.

If a key contains several attributes:

```sql
CREATE TABLE WorksAt (
    EmployeeId int NOT NULL,
    CompanyId int NOT NULL,
    PRIMARY KEY(EmployeeId, CompanyId)
)
```

REFERENTIAL INTEGRITY CONSTRAINTS IN SQL

```sql
CREATE TABLE Student (
    SSN char(10) NOT NULL PRIMARY KEY,
    First_Name varchar(255),
    Last_Name varchar(255)
);
CREATE TABLE Course (
    Code varchar(32) NOT NULL PRIMARY KEY,
    Title varchar(255)
);
CREATE TABLE Grade (
    StudentSSN char(10) NOT NULL,
    CourseCode varchar(32) NOT NULL,
    Grade char(1),
    PRIMARY KEY (StudentSSN, CourseCode),
    FOREIGN KEY (StudentSSN) REFERENCES Student(SSN),
    FOREIGN KEY (CourseCode) REFERENCES Course(Code)
);
```

TIP: USE INTEGER PRIMARY KEYS

```sql
CREATE TABLE Student (
    Id int NOT NULL PRIMARY KEY,
    SSN char(10) NOT NULL UNIQUE,
    First_Name varchar(255),
    Last_Name varchar(255)
);
CREATE TABLE Course (
    Id int NOT NULL PRIMARY KEY,
    Code varchar(32) NOT NULL UNIQUE,
    Title varchar(255)
);
CREATE TABLE Grade (
    StudentId int NOT NULL,
    CourseId int NOT NULL,
    Grade char(1),
    PRIMARY KEY (StudentId, CourseId),
    FOREIGN KEY (StudentId) REFERENCES Student(Id),
    FOREIGN KEY (CourseId) REFERENCES Course(Id)
);
```

REFERENCE OPTIONS

```
1 FOREIGN KEY (...)
2 REFERENCES ...(...)
3 [ON DELETE reference_option]
4 [ON UPDATE reference_option]
```

Reference options: CASCADE, SET NULL, NO ACTION

- CASCADE = Deletes/update matching rows in the child tables too
- NO ACTION = Delete/update will fail if there are matching rows in the child tables
- SET NULL = FK for matching rows in the child tables will be changed to NULL (NULLs must be allowed)

(Child tables = tables with a FK referencing the table on which we run DELETE or UPDATE.)

Note: Some RDBMS also support SET DEFAULT.

INSERTING NEW DATA

```
INSERT INTO Student(Id, SSN, First_Name, Last_Name)
VALUES (1, "8302141234", "John", "White")
```

Specifying attributes is not necessary if the order of values is the same as in the relation schema:

```
INSERT INTO Student
VALUES (1, "8302141234", "John", "White")
```

Tip: Always include the attribute names.

Note: If some of the values being inserted are NULL (and NULLs are allowed):

```
INSERT INTO Course(Id, Code)
VALUES (2, "1DL301")
```

It is also possible to specify default values for attributes. Omitting values for such attributes will lead to using the defaults.

DELETING AND MODIFYING DATA

Deleting a tuple from a relation:

```
1 DELETE FROM Student WHERE Id=1
```

Modifying a tuple in a relation:

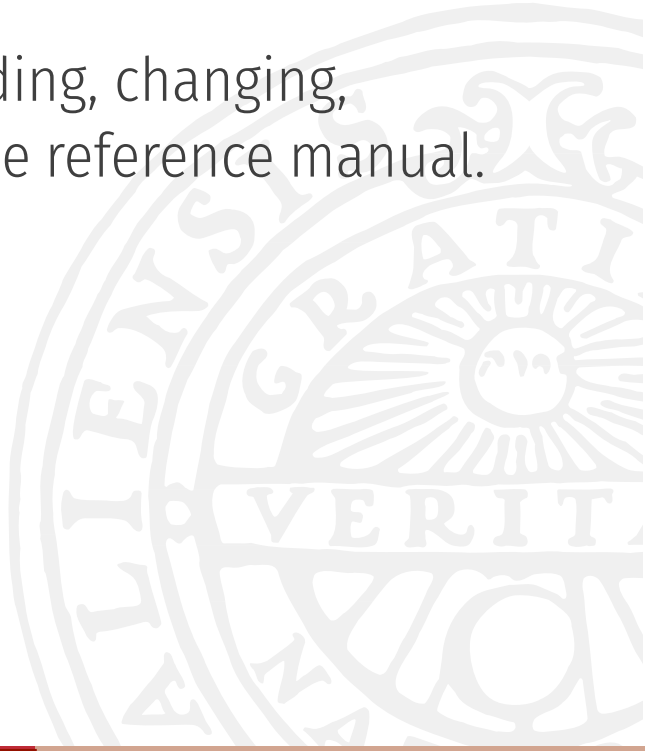
```
1 UPDATE Course
2 SET Title="Database Design I"
3 WHERE Id=2
```

REMOVING AND MODIFYING RELATIONS

Removing a relation:

```
1 DROP TABLE Student
```

`ALTER TABLE` can be used to modify the schema (adding, changing, removing the attributes) and constraints. Check the reference manual.



Part 2

SET OPERATIONS – UNION

```
SELECT ...
UNION [ALL]
SELECT ...
```

`UNION` combines the result sets from two (or more) `SELECT` queries. Duplicated rows are removed unless `UNION ALL` is used.

The result sets of the queries must have the same number of columns and compatible data types for each column.

Merged result might be ordered by adding an `ORDER BY` clause.

SET OPERATIONS – INTERSECT

```
1 SELECT ...
2 INTERSECT [ALL]
3 SELECT ...
```

`INTERSECT`: The query returns **distinct** rows that are in both result sets.

`INTERSECT ALL`: The query returns rows that are in both result sets. (If a given row is in the first result set c_1 times and in the second result set c_2 times, it will be in the result $\mathbf{min}(c_1, c_2)$ times.)

Note: Not used very often.

SET OPERATIONS – EXCEPT

```
1 SELECT ...
2 EXCEPT [ALL]
3 SELECT ...
```

EXCEPT: The query returns **distinct** rows from the first result set which are not in the second result set.

EXCEPT ALL: The query returns rows from the first result set which are not in the second result set. (If a given row is in the first result set c_1 times and in the second result set c_2 times, it will be in the result $c_1 - c_2$ times if $c_1 > c_2$, and not at all otherwise.)

Note: Not used very often.

SET OPERATIONS – EXAMPLE

```
1 SELECT * FROM T1 ... SELECT * FROM T2 ORDER BY 1
```

Rows in T1	Rows in T2	UNION	UNION ALL	INTERSECT	INTERSECT ALL	EXCEPT	EXCEPT ALL
A	A	A	A	A	A	C	A
A	A	B	A	B	A		C
A	B	C	A		B		
B	B	D	A				
C	D		A				
			B				
			B				
			B				
			C				
			D				

AGGREGATE FUNCTIONS

Aggregate functions, e.g., count(), sum(), avg(), min(), max(), can be used to calculate summary statistics based on rows selected by a WHERE clause.

The parameter of an aggregate function might be an attribute or an expression.

It is also possible to calculate statistics based on distinct values, i.e., `count(DISTINCT hour_salary)`.

If you use an aggregate function in an SQL query (without grouping), all columns in the result must be aggregates.

AGGREGATE FUNCTIONS, CONT'D

Aggregate functions only consider non-null values (with exception of `count(*)` which returns the number of matched rows).

Example: `SELECT salary ...` returns rows:
45, 36, 40, 45, 50, NULL, 73, 40, NULL, 46

`count(*)` = 10
Number of rows is 10.

`count(salary)` = 8
45, 36, 40, 45, 50, NULL, 53, 40, NULL, 46

`count(distinct salary)` = 6
45, 36, 40, 45, 50, NULL, 53, 40, NULL, 46

AGGREGATE FUNCTIONS, EXAMPLES

The number of employees and the average salary:

```
SELECT count(*), avg(hour_salary)
FROM employee
```

The number of employees and the average salary in the department with ID 1:

```
SELECT count(*), avg(hour_salary)
FROM employee
WHERE department_id = 1
```

GROUPING

We can divide selected rows into groups based on the value of one or several attributes (or expressions) and calculate summary statistics (aggregates) in each group.

GROUP BY CLAUSE

Grouping can be given by adding the `GROUP BY` clause:

```
SELECT select_expr [, ...]
FROM table_reference [, ...]
[WHERE where_condition]
[GROUP BY group_expr [, ...]]
[ORDER BY order_expr [ASC | DESC] [, ...]]
```

The `group_expr` can be an attribute, an expression or a column alias.

A special case we have already seen: Using aggregates without a GROUP BY clause = one group consisting of all selected rows.

GROUP BY CLAUSE, CONT'D

When using a GROUP BY clause, columns in the result must be:

- group statistics (aggregates), or
- grouping attributes (or expressions using these attributes)

If SQL99's T301 is implemented, you can also use:

- attributes functionally dependent on grouping attributes

Note: Some systems, including MySQL, might allow using other attributes too, but they will choose a "random" value from all (potentially different) attribute values in each group.

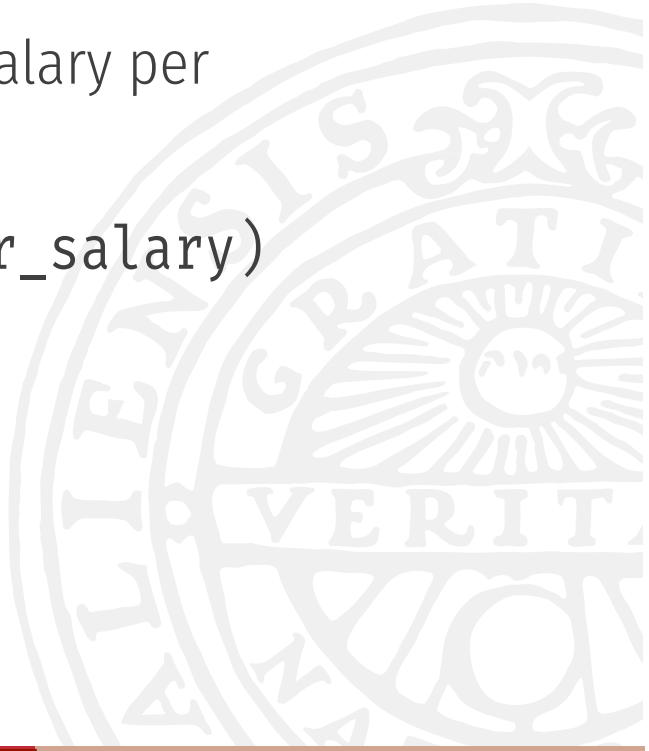
GROUP BY CLAUSE, EXAMPLES

The minimum and maximum salary per department:

```
1 SELECT department_id, min(hour_salary), max(hour_salary)
2 FROM employee
3 GROUP BY department_id
```

The total number of employees and the average salary per department:

```
1 SELECT department_id, count(*), avg(hour_salary)
2 FROM employee
3 GROUP BY department_id
```

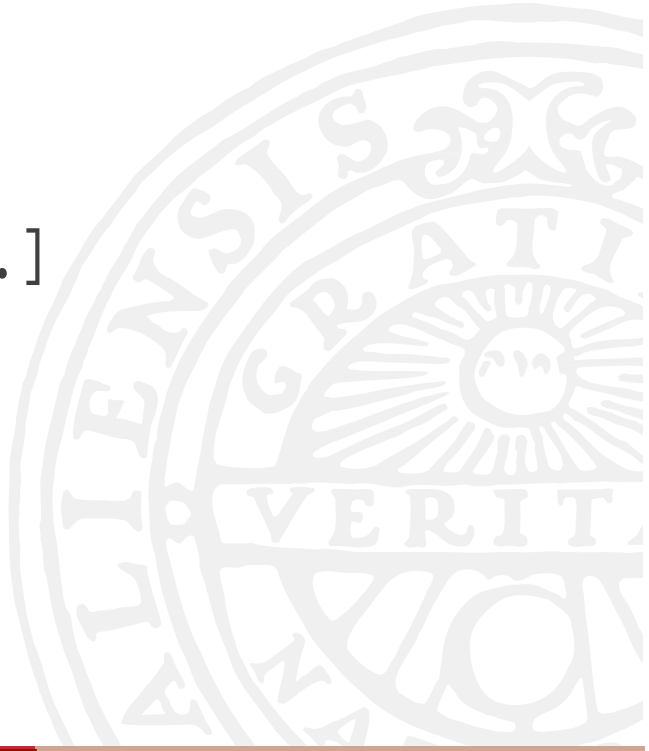


HAVING CLAUSE

A HAVING clause can be used to choose the groups that will be included in the result:

```
SELECT select_expr [, ...]
FROM table_reference [, ...]
[WHERE where_condition]
[GROUP BY group_expr [, ...]]
[HAVING having_condition]
[ORDER BY order_expr [ASC | DESC] [, ...]
```

No HAVING clause means all groups.



HAVING CLAUSE, EXAMPLE

What is the maximum salary in departments with at least 10 employees?

```
1 SELECT department_id, max(hour_salary) AS max_salary
2 FROM employee
3 GROUP BY department_id
4 HAVING count(*) >= 10
```

EXAMPLE STEP BY STEP

```
SELECT city, avg(price) FROM hotel WHERE country='SE'
GROUP BY city HAVING count(*) > 1
```

name	city	country	price
Akademihotellet	Uppsala	SE	696
Alexandra Hotel	Stockholm	SE	790
Anker Hotel	Oslo	NO	885
Best Western	Stockholm	SE	1880
Grand Hotel	Oslo	NO	1580
Nova Park	Knivsta	SE	2695
Park Inn	Uppsala	SE	1025
Radisson Blu	Uppsala	SE	1120
Rival	Stockholm	SE	1695
Smart Hotel	Oslo	NO	820

name
Akademihotellet
Park Inn
Radisson Blu

name
Alexandra Hotel
Best Western
Rival

name
Nova Park

Part 3

NESTED QUERIES, MOTIVATING EXAMPLE

Which employees have the lowest hour salary?

Attempt 1:

```
1 SELECT *
2 FROM employee
3 ORDER BY salary
4 LIMIT 1
```

Does this work?

NESTED QUERIES, MOTIVATING EXAMPLE, CONT'D

Which employees have the lowest hour salary?

Attempt 2:

```
1 SELECT min(salary)
```

returns 12780.

```
1 SELECT *
2 FROM employee
3 WHERE salary = 12780
```

NESTED QUERIES

In the WHERE clause, instead of a constant or a “hardcoded” value we can use a nested query which returns one value (single column, single row).

Attempt 3:

```
1 SELECT *
2 FROM employee
3 WHERE salary = (SELECT min(salary) FROM employee)
```

NESTED QUERIES, CONT'D

Similarly, instead of a list of constants, we can use a nested query which returns several values (single column, several rows).

Example:
Select departments with employees whose last name starts with K:

```
1 SELECT *
2 FROM department
3 WHERE id IN
4     (SELECT department_id
5      FROM employee
6      WHERE last_name LIKE 'K%')
```

CORRELATED NESTED QUERIES

A nested query can also use attributes from the tables used in the main (outer) query.

Such queries are called *correlated* or *synchronized*.

Example:

```
1 SELECT * FROM employee e
2 WHERE salary > (
3    SELECT avg(salary)
4    FROM employee
5    WHERE department_id=e.department_id
6 )
```

ANY AND ALL

```
expr (=|<>|<|<=|>|>=) ANY (nested_query)
```

True iff at least one of the values returned by the nested query meets the condition.

```
expr (=|<>|<|<=|>|>=) ALL (nested_query)
```

True iff all values returned by the nested query meet the condition.

`expr = ANY (nested_query)` is equivalent to `expr IN (nested_query)`.

`expr <> ALL (nested_query)` is equivalent to `expr NOT IN (nested_query)`.

[NOT] EXISTS

`EXISTS (nested_query)` – true iff the nested query returns at least one row.

`NOT EXISTS (nested_query)` – true iff the nested query does not return any row.

Example: Select the worst paid employees in each department:

```
1 SELECT *
2 FROM employee e
3 WHERE NOT EXISTS
4   (SELECT *
5    FROM employee
6    WHERE department_id = e.department_id
7      AND salary < e.salary)
```

USING THE RESULT OF A NESTED QUERY AS A TABLE

Example: Select the minimum of the average salaries in each department.

```
SELECT min(avg_hour_salary)
FROM (
  SELECT department_id,
         avg(hour_salary) AS avg_hour_salary
  FROM employee
  GROUP BY department_id
)
```

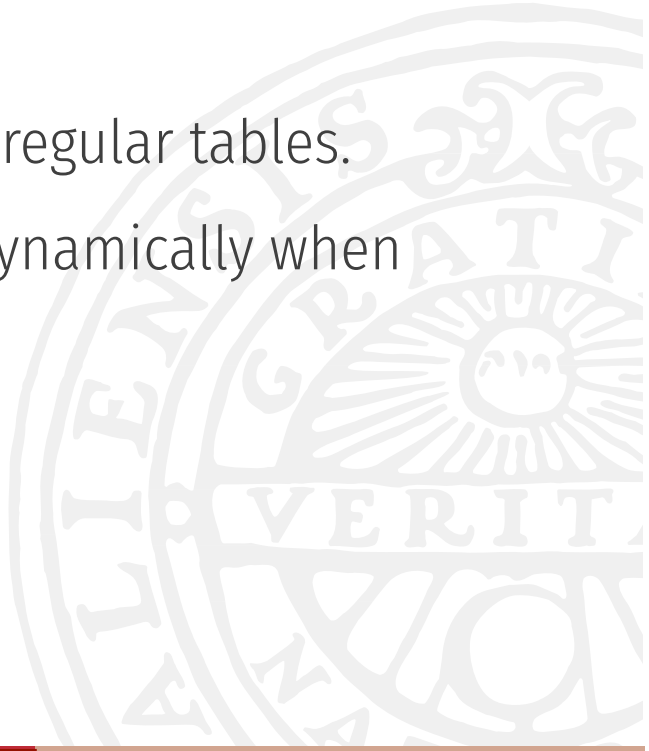
VIEWS

A view is a named “virtual” table representing the result set of a given SELECT query.

```
CREATE VIEW view_name AS
SELECT ...
```

A view can be used in queries in the same way as regular tables.

The content of this “virtual” table is determined dynamically when such a query is executed.



VIEWS, EXAMPLE

```
1 CREATE VIEW employee_and_department AS
2 SELECT employee.id, first_name, last_name, title
3 FROM employee, department
4 WHERE department_id = department.id;
5
6 SELECT * FROM employee_and_department
```

OUTER JOINS

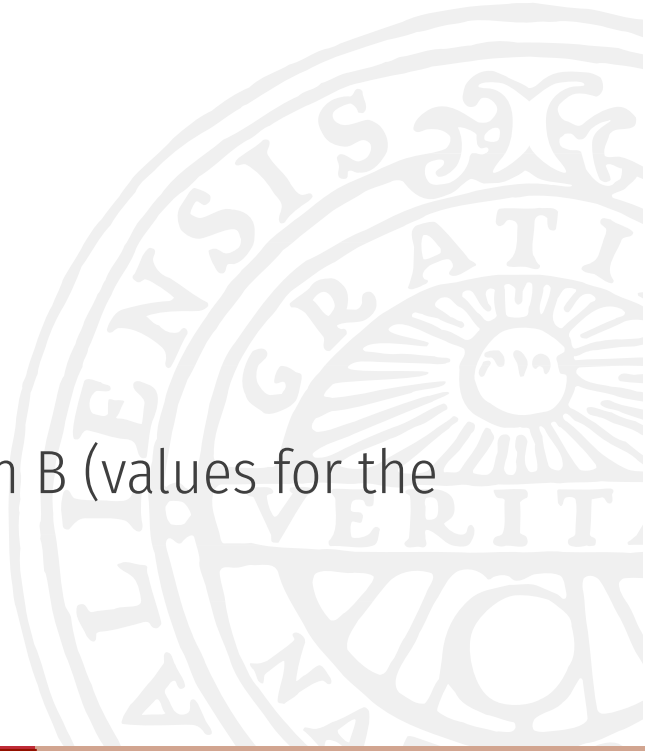
Inner joins like

```
1 SELECT *
2 FROM A
3 JOIN B ON B.attr_b = A.attr_a
```

returns “only” combinations of rows in A and B that match the condition.

```
1 SELECT *
2 FROM A
3 LEFT JOIN B ON B.attr_b = A.attr_a
```

includes also all rows in A with no matching row in B (values for the attributes in B in the result set are NULL)



OUTER JOINS, CONT'D

```sql
SELECT *
FROM A
RIGHT JOIN B ON B.attr_b = A.attr_a
```

includes rows in B with no matching row in A (values for the attributes in A in the result set are NULL).

OUTER JOINS, EXAMPLE

Table *employee*:

id	last_name	department_id
1	Nymo	23
2	Tveten	24

Table *department*:

id	title
23	IT
24	Math
25	Physics

```
1 SELECT d.title, e.id, e.last_name
2 FROM department d
3 LEFT JOIN employee e ON e.department_id = d.id
  =
1 SELECT d.title, e.id, e.last_name
2 FROM employee e
3 RIGHT JOIN department d ON e.department_id = d.id
```

title	id	last_name
IT	1	Nymo
Math	2	Tveten
Physics	NULL	NULL

REMINDER: INSERTION, UPDATE AND DELETION

Insertion:

```sql
INSERT INTO employee(id, first_name, last_name)
VALUES (11, "Peter", "Hansen")
```

Update:

```sql
UPDATE employee
SET hour_salary = hour_salary + 20
WHERE department_id = 4
```

Deletion:

```sql
DELETE FROM employee
WHERE id = 11
```

Warning: `DELETE FROM table_name` (without the WHERE clause) will remove all rows in the table!

FURTHER RESOURCES

- SQL Tutorial at w3schools.com
  https://www.w3schools.com/sql/
- MySQL Reference Manual
  https://dev.mysql.com/doc/

ACKNOWLEDGEMENT

The slide is derived from
Jan Kudlicka's DB1-Fall19 class.