Grouping

- general form of the select-from-where clause

  ```
  select
  from
  [where <condition>]
  [group by <group-by-expression> [, <group-by-expression>]*
   [having <condition>]]
  [order by <order-expression>]
  ```

- `group by` clause
  - A “group-by-expression” is an expression that refers only to those attributes that are *not* used for computing the aggregate. Tuples with equal values for the specified value are summarized in groups (partitions).
  - For each group the query produces a new tuple in the result relation. Hence, only attributes with one value per group are permitted after the `select` clause.

- `having` clause
  - choice of groups with respect to a condition which may contain only arguments with one value per group
- Determine the number of hours per week in which professors have given lectures.
  
  ```sql
  select held_by, sum(hpw) as number
  from lectures
  group by held_by
  ```

- Determine the number of hours per week of those lectures held by professors who predominantly give long lectures (> 2 hours per week on average).
  
  ```sql
  select held_by, sum(hpw) as number
  from lectures
  group by held_by
  having avg(hpw) > 2
  ```

- Determine the number of hours per week of those lectures held by C4 professors who predominantly give long lectures (> 2 hours per week on average).
  
  ```sql
  select held_by, name, sum(hpw) as number
  from lectures, professors
  where held_by = pers-id and rank = "C4"
  group by held_by, name
  having avg(hpw) > 2
  ```
Since in the result relation each group is represented by exactly one tuple, in the **select** clause only aggregate functions can appear, or attributes that are used for grouping, i.e., that are also used in the **group by** clause.

**Sorting**

- frequently a sorted output is required
  - → DBMS needs sort operator, sorting is expensive

- sorted output with the **order by** clause with respect to one or more attributes
  - **order by** [asc | desc] \( A_1, \ldots, [asc | desc] \ A_n \) \( A_i \) attribute
  - sorting order:
    - keyword **asc** = ascending (default)
    - keyword **desc** = descending

- The **order by** clause is the last clause in an SQL command.
example: Determine personell id, name and rank of all professors; sort the result tuples in descending order by rank and in ascending order by name.

\[
\text{select pers-id, name, rank from professors order by rank desc, name asc}
\]

<table>
<thead>
<tr>
<th>pers-id</th>
<th>name</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>2136</td>
<td>Curie</td>
<td>C4</td>
</tr>
<tr>
<td>2137</td>
<td>Kant</td>
<td>C4</td>
</tr>
<tr>
<td>2126</td>
<td>Russel</td>
<td>C4</td>
</tr>
<tr>
<td>2125</td>
<td>Sokrates</td>
<td>C4</td>
</tr>
<tr>
<td>2134</td>
<td>Augustinus</td>
<td>C3</td>
</tr>
<tr>
<td>2127</td>
<td>Kopernikus</td>
<td>C3</td>
</tr>
<tr>
<td>2133</td>
<td>Popper</td>
<td>C3</td>
</tr>
</tbody>
</table>

- attribute \textit{rank} main sorting condition, attribute \textit{name} minor sorting condition
Nested queries

- In the `where` clause and in the `from` clause of an SQL statement further SQL statements can appear. This is called a **nested query**.

- In the `where`-clause we differentiate whether the result of a subquery yields a scalar value or a relation.

- **Scalar subqueries**
  - example: Which students with a semester number less than the average are there?
    ```sql
    select name, sem
    from students
    where sem < (select avg(sem) from students)
    ```
  - Scalar subqueries in SQL92 are even allowed in the `select` clause of a query. In Oracle this feature is currently not supported.

- **Scalar subqueries with exists**
  - In the `where` clause also subqueries are allowed that yield a boolean value. These are indicated by the keyword **exists**.
  - The condition “[not] exists <subquery>” is true if the subquery is not empty [empty].
- Set-valued subqueries
  - The keyword [not] in tests if an attribute [does not take] takes a value of a set.
  - If the task is to test whether an attribute is in a certain relationship to all elements of a set, the keyword all can be used.

- queries with forall quantifiers
  - mathematical law: \((\forall x : \varphi(x)) \iff (\exists x : \neg\varphi(x))\). Hence, all queries containing a forall quantifier can be transformed to equivalent queries only containing existential quantifiers.
  - example: Which students attend all lectures offered by professor Curie?

```sql
select s.name from students as s
where not exists
  (select id from lectures, professors
    where pers-id = held_by and name = "Curie")
except
  (select l.id from attends as a, lectures as l
    where l.id = a.id and a.reg-id = s.reg-id))
```
- subqueries in the `from` clause
- Since an SQL query creates a relation, a query can also be used in the `from` clause.
- example: Output the ids of those lectures that are attended by more than 20 students.

```sql
select id
from (select id, count(*) as number from attends group by id)
where number > 20
```
- possible to explicitly use a join operator in SQL92 in the `from` clause by means of the keywords
  + `cross join` for the Cartesian (cross) product,
  + `natural join` for the natural join,
  + `join` or `inner join` for theta join,
  + `left outer join`, `right outer join` or `full outer join` for outer join analogously to the operators of the relational algebra: also tuples of the left, the right or both relations, which do not fulfil the join, are inserted into the result relation
  + `union join`: some kind of full outer join where no comparison is performed. Both schemas are concatenated. Tuples are united and supplemented by null values.
– example
+ select * from $R_1$, $R_2$ where $R_1.A \theta R_2.B$

can be explicitly formulated as theta join as follows:
select * from $R_1$ join $R_2$ on $R_1.A \theta R_2.B$

join condition is explicitly specified behind the on clause

Null values

- A special value null for an attribute in a relation indicates that the value is unknown.
- SQL uses a three-valued logic with the values true, false and unknown.
- Logical expressions yield the following results:

<table>
<thead>
<tr>
<th>not</th>
<th>true</th>
<th>false</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>unknown</td>
<td></td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>and</th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td></td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>
In the \textbf{where} clause only those tuples are selected where the filter condition yields \textit{true}. Additionally the condition \textit{where A is null} allows to select all tuples with a null value in attribute \textit{A}.

For grouping \textit{null} is considered as a self-contained value.

For sorting \textit{null} is always interpreted as value of highest priority.

\textbf{Recursive queries}

example: Which lectures must be attended to understand the lecture \textit{“The Vienna Circle”}?

\begin{verbatim}
select predecessor
from is_precondition_of, lectures
where successor = id and title = “The Vienna Circle”
\end{verbatim}

Query returns only immediate predecessors.
example: Which lectures are required for the immediate predecessors?

```sql
select predecessor
from is_precondition_of
where successor in ( select predecessor
    from is_precondition_of, lectures
    where successor = id and title = "The Vienna Circle")
```

SQL, relational algebra and relational calculus do not offer possibilities for an efficient computation of recursive queries.