Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CHAI
Humanities-Centered AI

Ralf Möller

Databases
Structured Query Language SQL (Part 2)

Overview

1. **Introduction**
   - Models and abstraction
   - Database systems
   - Applications and data independence
2. **Entity-relationship model (ER model)**
3. **Relational data model (RDM)**
   - From ER to RDM
4. **Relational schema design theory**
   - Relational algebra as a query language
   - Functional dependencies
   - Normal forms
5. **Structured Query Language (SQL)**
   - Data definition
   - Query specification
   - Data manipulation
6. **Query processing**
   - Architecture
   - Indexing
   - Query plans, optimization
7. **Transactions**
   - Transaction processing, schedules, locks
   - Recovery
8. **Other types of databases**
   - Vector databases
   - Data stream query languages


Duplicate elimination
and other special features

Example data: Project database

Duplicate elimination

Elimination of duplicates in query results using the keyword distinct:

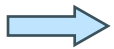
```
select distinct SupDept
from Departments;
```

SupDept
MSW
NULL

Here: Conversion of a result table into a result **set**

Detection and avoidance of null values in columns using the predicate is null or is not null

```
select distinct SupDept
from Departments
where SupDept is not null;
```

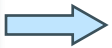


SupDept
MSW

Sorting

Sorted display of query results using the order by clause with the options asc (ascending) and desc (descending):

```
select *
from Departments
where SupDept = 'MSW'
order by ID asc;
```



Ergebnistabelle

ID	*Name*	*SupDept*
MFSW	Mainframe SW	MSW
PCSW	PC SW	MSW
UXSW	Unix SW	MSW

Sorting can cover multiple columns:

- Ascending sorting of all departments according to the abbreviation of their parent department
- Subsequently, within a parent department, the departments are sorted in descending order according to their abbreviation.

```
select *
from Departments
order by SupDept asc,
         ID desc;
```

Aggregation functions

- Use in the select clause of an SQL application
- Calculation of aggregated values (e.g., sum of all values in a column of a table)
- Example: Sum and maximum of the budgets for all projects

```
select sum(p.Budget),
       max(p.Budget)
from Projects p;
```

p.Budget

sum	*max*
600.000	300.000

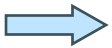
- Also includes functions for minimum (min), average (avg), and counting the table values in a column (count) or the number of tuples (count(*)).
- Example: Number of tuples in the Departments relation (including zero values and duplicates)

```
select count(*)
from Departments;
```

count(*)
5

- One-time counting of values possible (only non-null values)

```
select
  count(distinct SupDept)
from Departments;
```



count(*)
1

Naming query results

- Saving the query result in a separate, persistent table
- Example:
  - Snapshot of the data at the time of the query.
  - Can be reused independently of changes to the source data.

```
create table SW_SubDepartments as
select Name, ID from Departmnts where SupDept = 'MSW';
```

- Saving in a temporary table
- Example:
  - When the transaction or connection ends, data in the temporary table is automatically deleted.

```
create temporary table SW_SubDepartments as ...;
```

Naming queries

Definition of a view

- Not the result but the query is named
- For every occurrence, the query is evaluated again
  `SW_SubDepartments` is used as a standard table
- Direct use of a query result as a range relation in a complex query

```
create view SW_SubDepartments as
select Name, ID
from Departments where SubDept = 'MSW';
```

```
select u.name, p.nr
from SW_SubDepartments u,
     Project_implementations pi
where u.ID = pi.ID;
```

```
select u.Name, p.Nr
from (select Name, ID
       from Departments
       where SupDept = 'MSW') u,
     Project_implementations pi
where u.ID = pi.ID;
```

Visibility rules and column name conflicts (1)

Visibility rules for local names (e.g., column and range variable names) within calculation-oriented SQL queries:

All global names of SQL objects (e.g., tables, views, schemas, catalogs) are visible in the sub-expressions P, S, T1, ..., Tn.

```
select P
from T1, ..., Tn
where S;
```

In the selection predicate S and in the projection list P, the local names of all columns of all area tables Ti are also visible.

A local name overrides a global name.

Visibility rules and column name conflicts (2)

Definition of local scope variables (correlation names, alias names):

To avoid name conflicts between column names in different tables and between column names and global names

Use of local scope variables to qualify column names using dot notation in the selection predicate and the projection list

The aim of using scope variables in SQL queries:

Readability: Which scope table does a column name belong to?

Expressiveness: ⟶ reflexive queries (see below)

```
select P
from T1 X1, ...,
     Tn Xn
where S;
```

```
select e.*
from Employees e,
     Projects p
where e.Project =
      p.Nr;
```

Traversal of Structures

Visibility rules and column name conflicts (3)

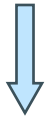
Example: Reflexive query

Here: Table of upper and lower divisions

```
select sup.Name as SupDept,
       sub.Name as SubDept
from Departments sup,
     Departments sub
where sub.SupDept = sup.ID;
```

Generalization:

Recursive queries ⟶ not possible with every system



SupDept	*SubDept*
MSW	Mainframe SW
MSW	Unix SW
MSW	PC SW

Graph traversal: fixed depth

Example: Determining all top-level departments of a department

```
select   sub.name as SubDept,
         sup1.name as firstSupDept,
         sup2.name as secondSupDept
         …
from     Departments sub,
         Departments sup1,
         Departments sup2
         …
where    sub.SupDept = sup1.ID
  and    sup1.SupDept = sup2.kurz
         …
```

Traversing depth is specified in the request, but should depend on the actual data

```
select   sub.name as SubDept,
         sup.name as SupDept
                  …
from     Departments sub,
         Departments sup,
         …
where    sub.SupDept = sup.ID

union

select   sub.name as SubDept,
         sup1.name as firstSupDept,
         sup2.name as secondSupDept
         …
from     Departments sub,
         Departments sup1,
         Departments sup2
         …
where    sub.SupDept = sup1.ID
  and    sup1.SupDept = sup2.kurz
…
```

For each hierarchy level, a separate query must be formulated and added to the overall result using a union.

Graph traversal: Varying depth

Example: Determining all top-level departments of a department

```
create recursive view SupDepartments
  select r.ID, r.SupDept
  from Departments r
union
  select sub.ID, sup.SupDept
  from Departments sup,
       Subdepartments sub
  where sup.ID = sub.SupDept
```

PostgreSQL-Syntax

- At the start of recursion, “subdivisions” only contains the tuples from the first subquery.
- Tuples resulting from the join in the second subquery are added to the extension of “Subdivisions” for the next iteration.
- The recursion is terminated as soon as the second subquery no longer returns any additional result tuples when using the results from the previous iteration ⟶ fixed point.

Grouping

Grouping

- Local composition of rows in a table into groups
  depending on values of so-called grouping attributes
- For all rows in a group, the values of grouping attributes are identical
- This way, a table is partitioned (each partition is called a group)
- SQL offset special query specifications applied to each group
- For each group, the overall result of a grouping query contains a tuple for the group result

Grouping: Example

For each super department, return the number of sub-departments

```
select supDept, count(ID)
from Departments
group by supDept;
```

ID	*Name*	*supDept*
MFSW	Mainframe SW	MSW
UXSW	Unix SW	MSW
PCSW	PC SW	MSW
MSW	Leitung SW	**NULL**
PERS	Personnel	**NULL**

Result table

Oberabt	*count(Kurz)*
MSW	3
NULL	2

Professors

ProfID	Name	Type	Room
2125	Socrates	C4	226
2126	Russel	C4	232
2127	Copernicus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Students

StudID	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Lectures

LectID	Title	Weekly_hours	Read_by
5001	Foundations	4	2137
5041	Ethics	4	2125
5043	Epistemology	3	2126
5049	Socratic Method	2	2125
4052	Logic	4	2125
5052	Philosophy of Science	3	2126
5216	Bioethics	2	2126
5259	Vienna Circle	2	2133
5022	Belief and Knowledge	2	2134
4630	Kant's 3 Critics	4	2137

prerequisite_lectures

Predecessor	Successor
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

enrolled

StudID	LectID
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

Assistants

AssID	Name	Field_of_expertise	Boss
3002	Platon	Theory of Forms	2125
3003	Aristoteles	Syllogistics	2125
3004	Wittgenstein	Language Games	2126
3005	Rheticus	Planetary Motions	2127
3006	Newton	Kepler's Laws	2127
3007	Spinoza	God and Nature	2126

examined

StudID	LectID	ProfID	Grade
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Aggregate functions and grouping

Aggregate functions:
avg, max, min, count, sum

```
select avg (Semester)
from Students;
```

```
select Read_by, sum (Weekly_hours)
from Lectures
group by Read_by;
```

```
select Read_by, Name, sum (Weekly_hours)
from Lectures, Professors
where Read_by = ProfID and Type = 'C4'
group by Read_by, Name
having avg(Weekly_hours) >= 3;
```

Special features of aggregate operations

- One result tuple per group
- If an attribute is mentioned in the select clause,
  it must also be mentioned in the **group by** clause
- Ony this way, the value is uniquely determined (note that there is one result tuple per group)
- Thus, Name must also occur in **group b**y in the last example

Ausführen einer Anfrage mit group by

Lectures x Professors							
LectID	Title	Weekly_hours	Read_by	ProfID	Name	Type	Room
5001	Foundations	4	2137	2125	Socrates	C4	226
5041	Ethics	4	2125	2125	Socrates	C4	226
...	...	...	...	...	...	...	...
4630	Kant's 3 Critics	4	2137	2137	Kant	C4	7

⇓ **where**-Bedingung

LectID	Title	Weekly_ hours	Read_by	ProfID	Name	Type	Room
5001	Foundations	4	2137	2137	Kant	C4	7
5041	Ethics	4	2125	2125	Socrates	C4	226
5043	Epistemology	3	2126	2126	Russel	C4	232
5049	Socratic Method	2	2125	2125	Socrates	C4	226
4052	Logic	4	2125	2125	Socrates	C4	226
5052	Philosophy of Science	3	2126	2126	Russel	C4	232
5216	Bioethics	2	2126	2126	Russel	C4	232
4630	Vienna Circle	4	2137	2137	Kant	C4	7

⇓ Grouping

LectID	Title	Weekly_hours	Read_by	ProfID	Name	Type	Room
5041	Ethics	4	2125	2125	Socrates	C4	226
5049	Socratic Method	2	2125	2125	Socrates	C4	226
4052	Logic	4	2125	2125	Socrates	C4	226
5043	Epistemology	3	2126	2126	Russel	C4	232
5052	Philosophy of Sc.	3	2126	2126	Russel	C4	232
5216	Bioethics	2	2126	2126	Russel	C4	232
5001	Foundations	4	2137	2137	Kant	C4	7
4630	Kant's 3 Critics	4	2137	2137	Kant	C4	7

⇓ **Having** condition

VorlNr	Title	Weekly_hours	Read_by	ProfID	Name	Type	Room
5041	Ethics	4	2125	2125	Socrates	C4	226
5049	Socratic Method	2	2125	2125	Socrates	C4	226
4052	Logics	4	2125	2125	Socrates	C4	226
5001	Foundations	4	2137	2137	Kant	C4	7
4630	Kant's 3 Critics	4	2137	2137	Kant	C4	7

⇓ Aggregation (**sum**) and projection

Ergebnis

Read_by	Name	**sum** (Weekly_hours)
2125	Socrates	10
2137	Kant	8

Element test

Example

```
select Name
from Professors
where ProfID in (select Read_by
                 from Lectures)
```

```
select Name
from Professors
where ProfID not in (select Read_by
                     from Lectures)
```

Element tests with nested query oft replaceable with non-nested query with join

Quantification (restricted form)

Universal quantification:

- $\{ x \in R \mid \forall y \in S : x > y \}$
- Table of all Projects x, such that they have a higher Budget as all external Projects y

Existential quantification:

- $\{ x \in R \mid \exists y \in S : x > y \}$
- Table with all Projects x, which are member of at least on Project_implementation y

- **= any** synonym to **in**.

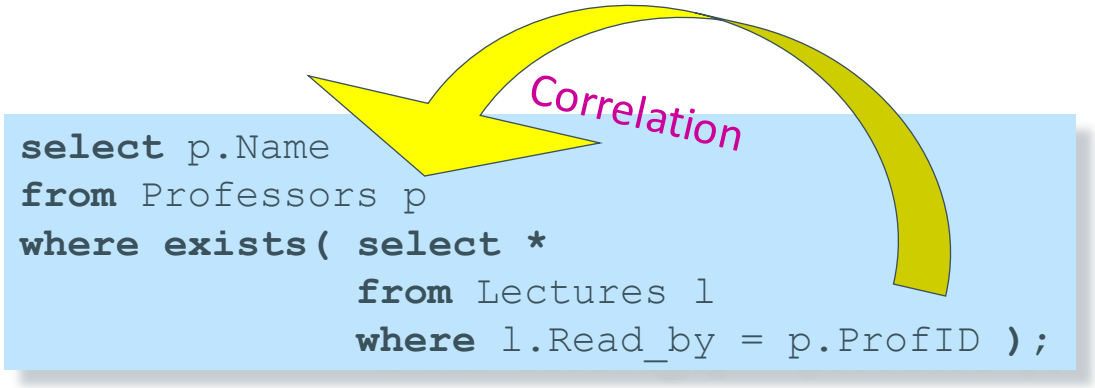
```
select *
from Projects x
where x.Budget > all
      (select y.Budget
       from External_projects y);
```

```
select *
from Projects x
where x.Budget > any
      (select y.Budget
       from External_projects y);
```

```
select *
from Projects as x
where x.Nr = any
      (select y.Nr
       from Project_implementations y);
```

Quantifizierung mit exists

Example: Return all professors reading a lecture



Negated existential quantifier

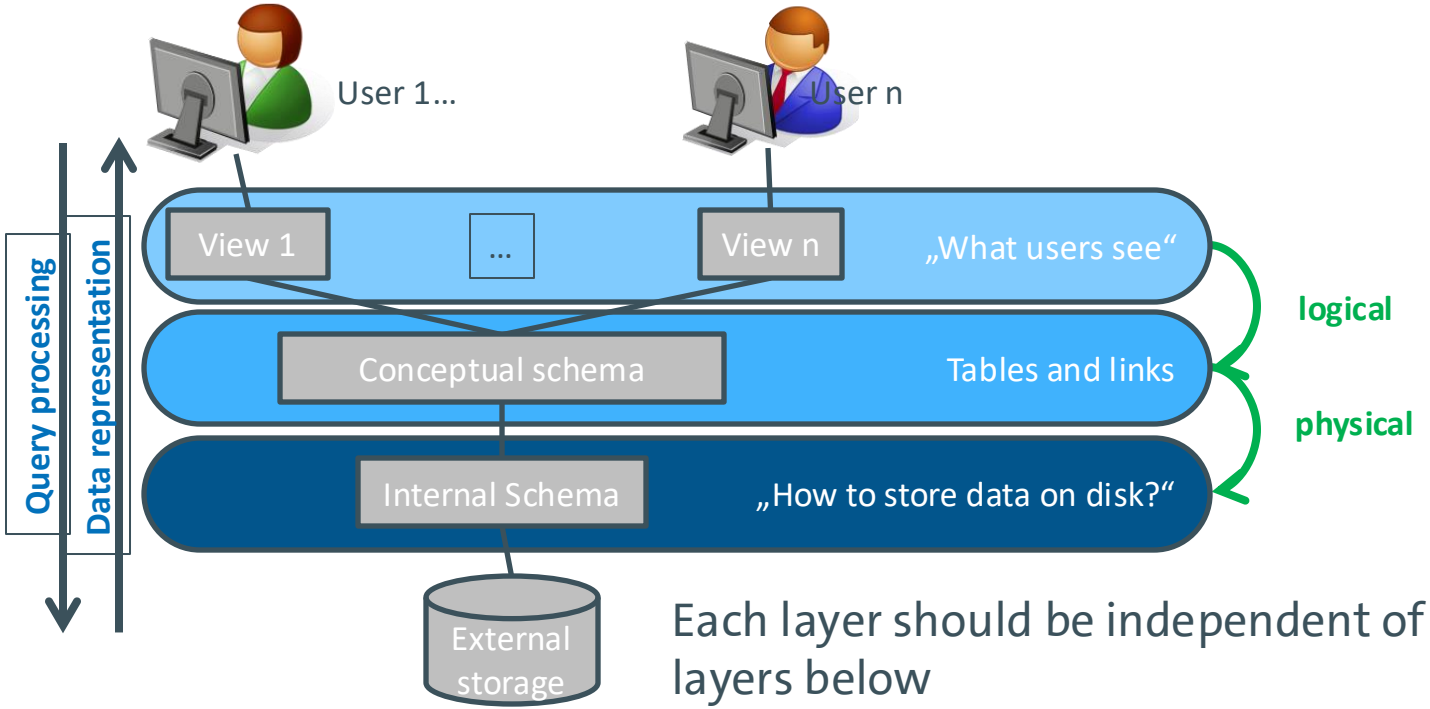
```
select p.Name
from Professors p
where not exists( select *
                  from Lectures l
                  where l.Read_by = p.ProfID );
```

Realization with set comparison

Uncorrelated sub-query:
Often executed in a more efficient way (nested query evaluated only once)

```
select Name
from Professors
where ProfID not in ( select Read_by
                      from Lectures );
```

Views for data abstraction?



Each layer should be independent of the layers below

Update of views?

Examples of non-updatable views

```
create view How_hard_as_examiner(ProfID, Average_grade) as
select ProfID, avg(Grade)
from examimed
group by ProfID;
```

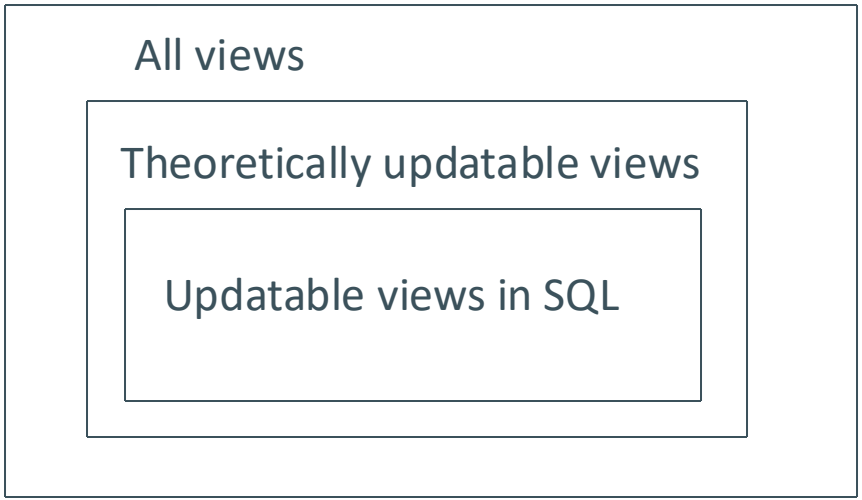
```
create view Lecture_view as
select Title, Weekly_hours, Name
from Lectures, Professors
where Read_by=ProfID;
```

```
insert into Lecture_view
values ('Nihilism', 2, 'Nobody');
```

Updatability of views

In SQL

- Ony one based relation
- Key must be given
- No aggregates, grouping, duplicate elimination



Integrity checking in SQL (besides primary/foreign keys and unique)

- **Self-defined domain names**
  - Base types with constraints

```
create domain Grade integer
  constraint Grade_defined check(value is not null)
  constraint Grade_between_1_and_6 check(value in(1,2,3,4,5,6));
```

- **Table constraints**
  - Define in table definitions

```
create table Table_name (...
  constraint Constraint_name
    check (Predicate))
```

- **Schema assertions**
  - Dynamically added to SQL schema (across tables)

```
create assertion Constraint_name
  check(Predicate);
```

- Constraints guarantee that in all database states the evaluation of the predicates return *true*
- In this case a constraint is called *fulfilled*

A database state is called *consistent* if all constraints declared in a schema are fulfilled.

Integrity of column values

- Avoid null values in a column
- Definition of sub-ranges
- Definition of format information for strings
- Definition of enumeration types

```
check(Age is not null)
```

```
check(Age >=0 and Age <=150)
```

```
check(Post_code like 'D-_____')
```

```
check(Grade in (1,2,3,4,5,6))
```

Integrity of rows

- A row table constraint, possibly referring to multiple attributes, must be fulfilled by every row

```
check(Expenditures <= Revenue)

check((BA, MA) in values(
  ('no', 'no')
  ('yes', 'yes')
  ('yes', 'yes')))
```

Table integrity

Refers to the whole table

Example

```
check((select sum(Budget) from Projekcs) >= 0)

check(exists(select * from Departments
             where SubDept = 'MSW'))
```

Overview

1. Introduction
   - Models and abstraction
   - Database systems
   - Applications and data independence
2. Entity-relationship model (ER model)
3. Relational data model (RDM)
   - From ER to RDM
4. Relational schema design theory
   - Relational algebra as a query language
   - Functional dependencies
   - Normal forms
5. Structured Query Language (SQL)
   - Data definition
   - Query specification
   - Data manipulation
6. **Query processing**
   - Architecture
   - Indexing
   - Query plans, optimization
7. **Transactions**
   - Transaction processing, schedules, locks
   - Recovery
8. **Other types of databases**
   - Vector databases
   - Data stream query languages