Kurs
Datenbankgrundlagen und Modellierung

Sebastian Maneth, Universität Bremen
maneth@uni-bremen.de

Sommersemester 2023

22.5.2023
Vorlesung 5: SQL Wrap Up & Functional Dependencies

Agenda

1.) RECAP
    — outer joins
    — subqueries

2.) NOT IN / NOT EXISTS

3.) UNION (ALL) / EXCEPT (ALL)

4.) ANY / ALL

5.) Functional Dependencies

6.) Armstrong Axioms

Kurs **Datenbankgrundlagen und Modellierung**

17.4. Vorlesung 1 — Intro
24.4. V2 — ER, SQL
1.5. keine Vorlesung
4.5. Fragestunden finden statt
8.5. V3 — SQL
10.5. Ü1
11.5. Fragestunden
15.5. V4 — SQL
17.5. Ü2
22.5. V5 — SQL
24.5. Ü3
25.5. Fragestunden
29.5. V6 — funct. dependencies
31.5. Ü4
1.6. Fragestunden
5.6. V7 — normal forms
7.6. Ü5
8.6. Fragestunden
12.6. V8 — modelling Intro
14.6. Ü6
15.6. Fragestunden
19.6. V9 — class diagrams
21.6. Ü7
22.6. Fragestunden
26.6. V10 — state charts
28.6. Ü8
29.6. Fragestunden
3.7. V11 — sequence diagrams
5.7. Ü9
6.7. Fragestunden
10.7. V12 — Klausurvorbereitung

Kurs **Datenbankgrundlagen und Modellierung**

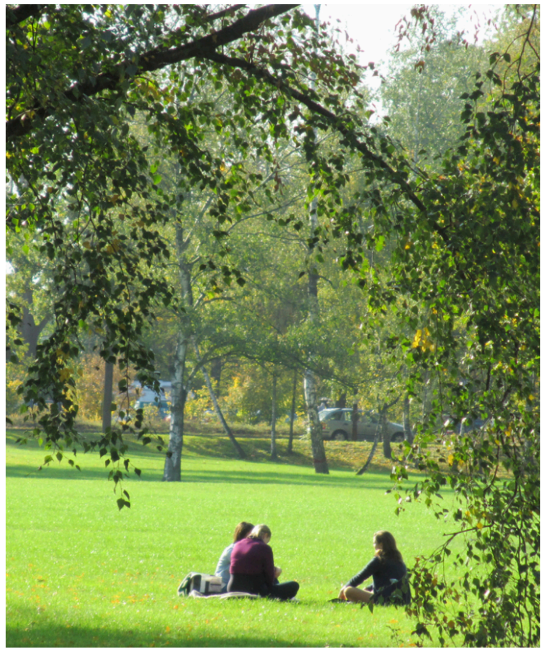
17.4. Vorlesung 1 — Intro
24.4. V2 — ER, SQL
1.5. keine Vorlesung
4.5. Fragestunden finden statt
8.5. V3 — SQL
10.5. Ü1
11.5. Fragestunden
15.5. V4 — SQL
17.5. Ü2
22.5. V5 — SQL
24.5. Ü3
25.5. Fragestunden
29.5. V6 — funct. dependencies
31.5. Ü4
1.6. Fragestunden
5.6. V7 — normal forms
7.6. Ü5
8.6. Fragestunden
12.6. V8 — modelling Intro
14.6. Ü6
15.6. Fragestunden
19.6. V9 — class diagrams
21.6. Ü7
22.6. Fragestunden
26.6. V10 — state charts
28.6. Ü8
29.6. Fragestunden
3.7. V11 — sequence diagrams
5.7. Ü9
6.7. Fragestunden
10.7. V12 — Klausurvorbereitung

Akademischer Kalender

© International Office

Wintersemester 2023/24

Semesterdauer	01.10.2023 - 31.03.2024
Orientierungswochen (inklusive Deutsch Intensivkurs) - optional	25.09. – 13.10.2023
Vorlesungszeit	16.10.2023-02.02.2024
Prüfungszeitraum	s.u.
Veranstaltungsfreie Tage:	
Tag der Deutschen Einheit Reformationstag	03.10.2023 31.10.2023
Weihnachtsferien	23.12.2023 – 05.01.2024



© International Office

Sommersemester 2023

Semesterdauer	01.04.2023 - 30.09.2023
Orientierungswochen	20.03.2023 - 06.04.2023
Vorlesungszeit	11.04.2023 – 14.07.2023
Prüfungszeitraum	s.u.
Veranstaltungsfreie Tage:	
Karfreitag Ostern	07.04.2023 09./10.04.2023
Tag der Arbeit Christi Himmelfahrt Pfingsten	01.05.2023 18.05.2023 28./29.05.2023

Kurs **Datenbankgrundlagen und Modellierung**

17.4. Vorlesung 1 — Intro
24.4. V2 — ER, SQL
1.5. keine Vorlesung
4.5. Fragestunden finden statt
8.5. V3 — SQL
10.5. Ü1
11.5. Fragestunden
15.5. V4 — SQL
17.5. Ü2
22.5. V5 — SQL
24.5. Ü3
25.5. Fragestunden
~~29.5. V6 — funct. dependencies~~
31.5. Ü4
1.6. Fragestunden
5.6. V6 — normal forms
7.6. Ü5
8.6. Fragestunden

12.6. V7 — modelling Intro
14.6. Ü6
15.6. Fragestunden
19.6. V8 — class diagrams
21.6. Ü7
22.6. Fragestunden
26.6. V9 — state charts
28.6. Ü8
29.6. Fragestunden
3.7. V10 — sequence diagrams
5.7. Ü9
6.7. Fragestunden
10.7. V11 — Klausurvorbereitung

Kurs **Datenbankgrundlagen und Modellierung**

17.4. Vorlesung 1 — Intro
24.4. V2 — ER, SQL
1.5. keine Vorlesung
4.5. Fragestunden finden statt
8.5. V3 — SQL
10.5. Ü1
11.5. Fragestunden
15.5. V4 — SQL
17.5. Ü2
22.5. V5 — SQL & funct. dependencies
24.5. Ü3
25.5. Fragestunden
~~29.5. V6 — funct. dependencies~~
31.5. Ü4
1.6. Fragestunden
5.6. V6 — normal forms
7.6. Ü5
8.6. Fragestunden
12.6. V7 — modelling Intro
14.6. Ü6
15.6. Fragestunden
19.6. V8 — class diagrams
21.6. Ü7
22.6. Fragestunden
26.6. V9 — state charts
28.6. Ü8
29.6. Fragestunden
3.7. V10 — sequence diagrams
5.7. Ü9
6.7. Fragestunden
10.7. V11 — Klausurvorbereitung

General Form of an SQL Query:

```
SELECT   list of attributes
FROM     list of tables
WHERE    condition over attributes
GROUP BY list of attributes
HAVING   condition over aggregates
ORDER BY list of attributes
LIMIT    number
```

aggregate functions:

```
COUNT    VARIANCE
SUM      STDDEV
AVG      BIT_OR
MAX      BIT_AND
MIN
```

1. Outer Joins

— introduce NULLs, when there is no “join partner”
— LEFT / RIGHT / FULL outer joins

1. Outer Joins

From last year's (2022) exam paper:

- introduce NULLs, when there is no "join partner"
- LEFT / RIGHT / FULL outer joins

```
CT            name      |     ing
                ----------+------------
                 White R. | vodka
                 White R. | milk
                 White R. | kahlua
                 Black R. | vodka
                 Black R. | kahlua
                 Nojito   | lime juice
                 Nojito   | club soda
                (7 rows)

IG              ing       | alc
               ------------+-----
                vodka      |  40
                milk       |   0
                kahlua     |  20
                lime juice |   0
                club soda  |   0
               (5 rows)
```

Teil 1.e. (1.0 Punkte)

```
SELECT name, COUNT(alc) AS cnt FROM
  ( SELECT name, alc FROM
    CT NATURAL JOIN IG WHERE alc>0 )
  NATURAL RIGHT JOIN
  ( SELECT DISTINCT name FROM CT )
GROUP BY name;
```

What does this query compute?

1. Outer Joins

From last year's (2022) exam paper:

- introduce NULLs, when there is no "join partner"
- LEFT / RIGHT / FULL outer joins

```
CT          name    |     ing
         -----------+------------
          White R.  | vodka
          White R.  | milk
          White R.  | kahlua
          Black R.  | vodka
          Black R.  | kahlua
          Nojito    | lime juice
          Nojito    | club soda
         (7 rows)

IG          ing        | alc
         --------------+-----
          vodka        |  40
          milk         |   0
          kahlua       |  20
          lime juice   |   0
          club soda    |   0
         (5 rows)
```

Teil 1.e. (1.0 Punkte)

```
SELECT name, COUNT(~~alc~~ *) AS cnt FROM
  ( SELECT name, alc FROM
    CT NATURAL JOIN IG WHERE alc>0 )
  NATURAL RIGHT JOIN
  ( SELECT DISTINCT name FROM CT )
GROUP BY name;
```

What happens now?

Subqueries

Anywhere in a query where
— a table name may appear, we may also place a nested query (a “subquery”)
— a value may appear, we may place a query that returns **one value**

Return ingredients with the maximum alcohol value:

```
dblp=# select * from ingredients;
  ingr  | alcohol
--------+---------
 Kahlua |    20.0
 Milk   |     0.0
 Vodka  |    40.0
 Rum    |    40.0
(4 rows)

dblp=# select max(alcohol) from ingredients;
 max
------
 40.0
(1 row)

dblp=# select ingr from ingredients where alcohol=40;
 ingr
-------
 Vodka
 Rum
(2 rows)

dblp=#
```

Return ingredients with the maximum alcohol value:

smaneth — screen -R — 70×23

```
dblp=# select * from ingredients;
  ingr   | alcohol
---------+---------
 Kahlua |    20.0
 Milk   |     0.0
 Vodka  |    40.0
 Rum    |    40.0
(4 rows)

dblp=# select max(alcohol) from ingredients;
 max
------
 40.0
(1 row)

dblp=# select ingr from ingredients where alcohol=40;
 ingr
-------
 Vodka
 Rum
(2 rows)

dblp=#
```

Much more elegant: **nest** this query here!

smaneth — screen -R — 70×23

```
dblp=# select ingr from ingredients where alcohol=(select max(alcohol)
 from ingredients);
 ingr
-------
 Vodka
 Rum
(2 rows)

dblp=#
```

nested subquery
— placed in brackets

smaneth — screen -R — 70×23

```
dblp=# select ingr from ingredients where alcohol=(select max(alcohol)
 from ingredients);
 ingr
-------
 Vodka
 Rum
(2 rows)

dblp=# select ingr from ingredients where alcohol=(select alcohol from
 ingredients);
ERROR:  more than one row returned by a subquery used as an expression
dblp=#
```

Subqueries

Anywhere in a query where
— a table name may appear, we may place a nested query (a “subquery”)

Nested queries (that return more than one value) are
best demonstrated in conjunction with the **NOT IN / IN** keywords

Non-Monotonic Behavior

SQL queries that use only the constructs introduced above are **monotonic** (↗ slide 104).

→ If further tuples are **inserted** to the database, the query result can only **grow**.

Some real-world queries, however, demand **non-monotonic** behavior.

- *E.g., "Return all non-alcoholic cocktails (i.e., those without any alcoholic ingredient)."*
  → Insertion of a new *ConsistsOf* tuple could "make" a cocktail alcoholic and thus invalidate a previously correct answer.

Such queries **cannot** be answered with the SQL subset we saw so far.

Not true for us because using HAVING and aggregates we already expressed non-monotonic queries.

Non-monotonic queries

Consider this query:

Return all Cocktails that do not contain Vodka.

How would you write it in SQL?

```
CT            name    |    ing
           ----------+------------
            White R. | vodka
            White R. | milk
            White R. | kahlua
            Black R. | vodka
            Black R. | kahlua
            Nojito   | lime juice
            Nojito   | club soda
           (7 rows)

IG            ing        | alc
           ------------+-----
            vodka      |  40
            milk       |   0
            kahlua     |  20
            lime juice |   0
            club soda  |   0
           (5 rows)
```

IN / NOT IN

Consider this query:

Return all Cocktails that do **not** contain Vodka.

```
CT          name    |    ing
          ----------+-----------
           White R. | vodka
           White R. | milk
           White R. | kahlua
           Black R. | vodka
           Black R. | kahlua
           Nojito   | lime juice
           Nojito   | club soda
          (7 rows)

IG          ing      | alc
          -----------+-----
           vodka      |  40
           milk       |   0
           kahlua     |  20
           lime juice |   0
           club soda  |   0
          (5 rows)
```

```
select Name from Cocktails
where ingr='Vodka'
```

returns Cocktails that contain Vodka

IN / NOT IN

Consider this query:

Return all Cocktails that do **not** contain Vodka.

```
CT           name     |     ing
             ----------+------------
              White R. | vodka
              White R. | milk
              White R. | kahlua
              Black R. | vodka
              Black R. | kahlua
              Nojito   | lime juice
              Nojito   | club soda
             (7 rows)

IG             ing      | alc
             ------------+-----
              vodka      |  40
              milk       |   0
              kahlua     |  20
              lime juice |   0
              club soda  |   0
             (5 rows)
```

returns Cocktails that
do not contain Vodka

```
select distinct Name from Cocktails
where Name NOT IN (
select Name from Cocktails
where ingr='Vodka' );
```

returns Cocktails that
contain Vodka

IN / NOT IN

returns Cocktails that **do not** contain Vodka

Consider this query:

Return all Cocktails that do **not** contain Vodka.

```
select distinct Name from Cocktails
where Name NOT IN (
select Name from Cocktails
where ingr='Vodka' );
```

returns Cocktails that contain Vodka

CT

name	ing
White R.	vodka
White R.	milk
White R.	kahlua
Black R.	vodka
Black R.	kahlua
Nojito	lime juice
Nojito	club soda

(7 rows)

IG

ing	alc
vodka	40
milk	0
kahlua	20
lime juice	0
club soda	0

(5 rows)

Explain why this query is non-monotonic.

Indicators for Non-Monotonic Behavior

Indicators for non-monotonic behavior (in natural language):

- "there is no", "does not exist", etc.
  $\rightarrow$ **existential quantification**
- "for all", "the minimum/maximum"
  $\rightarrow$ **universal quantification**
  $\rightarrow$ $\forall r \in R : C(r) \;\Leftrightarrow\; \nexists r' \in R : \neg C(r')$

In an equivalent SQL formulation of such queries, this ultimately leads to a test whether a certain **query yields a (non-)empty result**.

Indicators for Non-Monotonic Behavior

Indicators for non-monotonic behavior (in natural language):

- "there is no", "does not exist", etc.
  $\rightarrow$ **existential quantification**
- "for all", "the minimum/maximum"
  $\rightarrow$ **universal quantification**
  $\rightarrow$ $\forall r \in R : C(r) \;\Leftrightarrow\; \nexists r' \in R : \neg C(r')$

Poll:
Do you know
existential / universal quantifiers?
(first-order logic)

In an equivalent SQL formulation of such queries, this ultimately leads to a test whether a certain **query yields a (non-)empty result**.

IN / NOT IN

Such tests can be expressed with help of the IN ($\in$) and NOT IN ($\notin$) keywords in SQL:

```
SELECT c.Name
  FROM Cocktails AS c
 WHERE CocktailID NOT IN (SELECT co.CocktailID
                            FROM ConsistsOf AS co,
                                 Ingredients AS i
                           WHERE i.IngrID = co.IngrID
                             AND i.Alcohol <> 0 )
```

What does this query compute?

IN / NOT IN

Such tests can be expressed with help of the `IN` ($\in$) and `NOT IN` ($\notin$) keywords in SQL:

```
SELECT c.Name
  FROM Cocktails AS c
 WHERE CocktailID NOT IN (SELECT co.CocktailID
                            FROM ConsistsOf AS co,
                                 Ingredients AS i
                           WHERE i.IngrID = co.IngrID
                             AND i.Alcohol <> 0)
```

The `IN` (`NOT IN`) keyword tests whether an attribute value appears (does not appear) in a set of values computed by another SQL **subquery**.

→ At least conceptually, the subquery is evaluated before the main query starts.

IN vs. Join

Consider again the query for all alcoholic cocktails.

✎ **Do the following queries return the same result?**

```
SELECT Name
  FROM Cocktails
 WHERE CocktailID IN (SELECT DISTINCT CocktailID
                        FROM ConsistsOf AS co,
                             Ingredients AS i
                       WHERE i.IngrID = co.IngrID
                         AND i.Alcohol > 0 )
```

```
SELECT DISTINCT c.Name
  FROM Cocktails AS c, ConsistsOf AS co,
       Ingredients AS i
 WHERE c.CocktailID = co.CocktailID
   AND co.IngrID = i.IngrID AND i.Alcohol > 0
```

IN vs. Join

Consider again the query for all alcoholic cocktails.

✎ **Do the following queries return the same result?**

```
SELECT Name
  FROM Cocktails
 WHERE CocktailID IN (SELECT DISTINCT CocktailID
                        FROM ConsistsOf AS co,
                             Ingredients AS i
                       WHERE i.IngrID = co.IngrID
                         AND i.Alcohol > 0)
```

no!
Only if you add DISTINCT
to the upper query!

```
SELECT DISTINCT c.Name
  FROM Cocktails AS c, ConsistsOf AS co,
       Ingredients AS i
 WHERE c.CocktailID = co.CocktailID
   AND co.IngrID = i.IngrID AND i.Alcohol > 0
```

EXISTS / NOT EXISTS

The construct `NOT EXISTS` enables the main (or outer) query to check whether the **result of a subquery is empty**.[9]

- In the subquery, tuple variables declared in the `FROM` clause of the outer query may be referenced.

```
SELECT Name
  FROM Cocktails AS c
 WHERE NOT EXISTS (SELECT DISTINCT CocktailID
                    FROM ConsistsOf AS co,
                         Ingredients AS i
                   WHERE i.IngrID = co.IngrID
                     AND co.CocktailID = c.CocktailID
                     AND i.Alcohol > 0 )
```

What does this query compute?

[9] Likewise, `EXISTS` tests for non-emptiness.

Correlated Subqueries

The reference of an outer tuple makes the subquery **correlated**.

- The subquery is **parameterized** by the outer tuple variable.
- Conceptually, correlated subqueries have to be **re-evaluated** for every new binding of a tuple to the outer tuple variable.
  - $\rightarrow$ Again, the DBMS is free to choose a more efficient evaluation strategy that returns the same result ($\leadsto$ "query unnesting")

Correlation can be used with `IN`/`NOT IN`, too.

$\rightarrow$ Typically, this yields complicated query formulations (bad style).

Queries with `EXISTS`/`NOT EXISTS` can be non-correlated.

$\rightarrow$ The `WHERE` predicate then becomes **independent** of the outer tuple.

$\rightarrow$ This is rarely desired and almost always an indication of an **error**.

Correlated Subqueries

Subqueries may reference tuple variables from the **outer query**.

The converse (referencing a tuple variable of the subquery in the outer query) is **not** allowed:

```
SELECT c.Name, i.Alcohol  ← wrong!
  FROM Cocktails AS c
 WHERE EXISTS (SELECT DISTINCT CocktailID
                 FROM ConsistsOf AS co,
                      Ingredients AS i
                WHERE i.IngrID = co.IngrID
                  AND co.CocktailID = c.cocktailID
                  AND i.Alcohol > 0)
```

→ Compare this to **variable scoping** in block-structured programming languages (C, Java).

EXISTS / NOT EXISTS

- `EXISTS`/`NOT EXISTS` only tests for the **existence** of (at least) one row in the subquery result.
- The **actual tuple value** returned by the query is **immaterial** to the overall query result.
- It is good style to make this explicit in the subquery phrasing:
  - → `...EXISTS (SELECT * FROM ...)`
  - → `...EXISTS (SELECT NULL FROM ...)`
  - → `...EXISTS (SELECT 42 FROM ...)`
- It is legal SQL syntax, though, to specify arbitrarily complex result tuples in the subquery's `SELECT` clause.

3.) Set and Multi-Set Operations

UNION

The SQL keyword `UNION` allows to collect results from multiple queries into a single output relation ($\rightsquigarrow$ algebra operator $\cup$).

```
SELECT Name, Price
  FROM Ingredients
 WHERE Alcohol > 0
UNION
SELECT Name, Price
  FROM Cocktails
```

`UNION` is **strictly needed** (no other way in SQL to express such queries).

Typical use case:

→ Specializations of a general concept are stored in separate tables. They can be re-combined using `UNION`.

SQL Set Operators

- Combined relations must be **schema-compatible**.
  - But SQL is less strict than relational algebra.
  - Both operands must have the same number of columns; columns of compatible types must be listed in same order. Column names, however, do not matter (need not be identical).
- The **other set operators** are available in SQL, too:
  - `UNION` implements $\cup$
  - `EXCEPT` implements $-$ (`MINUS` is synonym)
  - `INTERSECT` implements $\cap$
- All three operators **remove duplicates**.
- To **keep duplicates**: combine with `ALL`

```
SELECT ··· FROM ··· WHERE ···
UNION ALL        (or: EXCEPT ALL, INTERSECT ALL)
SELECT ··· FROM ··· WHERE ···
```

Union has Set-Semantics (**no duplicates!**)

```
> SELECT * FROM T1;
+------+------+
| col1 | col2 |
+------+------+
|    1 |    2 |
|    2 |    1 |
|    1 |    1 |
|    2 |    3 |
+------+------+

> SELECT col1 FROM T1 UNION SELECT col2 FROM T1;
+------+
| col1 |
+------+
|    1 |
|    2 |
|    3 |
+------+
```

Union has Set-Semantics (**no duplicates!**)

— same holds for Intersect and for Except

3.) Set and Multi-Set Operations

If you want multi-set semantics, use

— Union ALL
— Intersect ALL
— Except ALL

```
> SELECT * FROM T1;
+------+------+
| col1 | col2 |
+------+------+
|    1 |    2 |
|    2 |    1 |
|    1 |    1 |
|    2 |    3 |
+------+------+

> SELECT col1 FROM T1 UNION ALL SELECT col2 FROM T1;
+------+
| col1 |
+------+
|    1 |
|    2 |
|    1 |
|    2 |
|    2 |
|    1 |
|    1 |
|    3 |
+------+
```

3.) Set and Multi-Set Operations

1. “UNION ALL” – adds multiplicities
2. “INTERSECT ALL” – keeps **minimum** (non-0) number of occ's of an element

```
select year from movies
                natural join actors2movies
                natural join persons
                where name='Jack Nicholson'
                GROUP BY year;
```

equivalent queries

```
select DISTINCT year from movies
                natural join actors2movies
                natural join persons
                where name='Jack Nicholson';
```

do you see yet another way to formulate this query?

```
select year from movies
                natural join actors2movies
                natural join persons
                where name='Jack Nicholson'
                GROUP BY year;
```

equivalent queries

```
select DISTINCT year from movies
                natural join actors2movies
                natural join persons
                where name='Jack Nicholson';
```

```
select year from movies
                natural join actors2movies
                natural join persons
                where name='Jack Nicholson'
                UNION
                select year from movies where year='-1';
```

union with the *emtpy set*

4.) ANY and ALL

- Find movies that are shorter than some currently playing movie:

```
SELECT M.Title
FROM Movies M
WHERE M.length < (SELECT MAX(M1.length)
                  FROM Movies M1, Schedule S
                  WHERE M1.title=S.title)
```

4.) ANY and ALL

- Find movies that are shorter than some currently playing movie:

```
SELECT M.Title
FROM Movies M
WHERE M.length < (SELECT MAX(M1.length)
                  FROM Movies M1, Schedule S
                  WHERE M1.title=S.title)
```

or

```
SELECT M.Title
FROM Movies M
WHERE M.length < ANY(SELECT M1.length
                     FROM Movies M1, Schedule S
                     WHERE M1.title=S.title)
```

4.) ANY and ALL

- `<value> <condition> ANY ( <query> )`
  is true if for some `<value1>` in the result of `<query>`,
  `<value> <condition> <value1>` is true.

4.) ANY and ALL

- `<value> <condition> ANY ( <query> )` is true if for some `<value1>` in the result of `<query>`, `<value> <condition> <value1>` is true.
- For example,

  $5 < \mathtt{ANY}(\emptyset)$ is false;

  $5 < \mathtt{ANY}(\{1, 2, 3, 4\}$ is false;

  $5 < \mathtt{ANY}(\{1, 2, 3, 4, 5, \underline{6}\}$ is true.

4.) ANY and ALL

- `<value>  <condition>  ALL ( <query> )` is true if either:
  - `<query>` evaluates to the empty set, or
  - for every `<value1>` in the result of `<query>`, `<value>  <condition>  <value1>` is true.

4.) ANY and ALL

- `<value> <condition> ALL ( <query> )` is true if either:
  - `<query>` evaluates to the empty set, or
  - for every `<value1>` in the result of `<query>`, `<value> <condition> <value1>` is true.

- For example,

  $5 > \mathtt{ALL}(\emptyset)$ is true;

  $5 > \mathtt{ALL}(\{1, 2, 3\})$ is true;

  $5 > \mathtt{ALL}(\{1, 2, 3, 4, 5, 6\})$ is false.

4.) ANY and ALL

fluss

fname	laenge
Weser	452
Elbe	1091

durch

fname	sname	laenge
Weser	Bremen	45
Elbe	Dresden	32
Elbe	Hamburg	36

4.) ANY and ALL

fluss

fname	laenge
Weser	452
Elbe	1091

durch

fname	sname	laenge
Weser	Bremen	45
Elbe	Dresden	32
Elbe	Hamburg	36

What is the **meaning** of this query?

```
SELECT fname
FROM fluss
WHERE ‘Bremen’ = ALL
                  (SELECT sname
                   FROM durch
                   WHERE durch.fname=fluss.fname);
```

4.) ANY and ALL

X IN S is equivalent to X = ANY S

X NOT IN S is equivalent to X != ALL S

4.) Examples with IN

```
SELECT (‘a’) IN (‘a’,’b’);
?column?
_________
 t

SELECT (‘a’,’b’) IN ((‘a’,’b’),(‘a’,NULL));
?column?
_________
 t
```

4.) Examples with IN

```
SELECT ('a') IN ('a','b');
?column?
─────────
 t

SELECT ('a','b') IN (('a','b'),('a',NULL));
?column?
─────────
 t

SELECT ('a',NULL) IN (('a','b'),('a',NULL));

—> ??
```

4.) Examples with IN

```
SELECT (‘a’) IN (‘a’,’b’);
?column?
_________
 t

SELECT (‘a’,’b’) IN ((‘a’,’b’),(‘a’,NULL));
?column?
_________
 t

SELECT (‘a’,NULL) IN ((‘a’,’b’),(‘a’,NULL));
?column?
_________
 NULL
```

Functional Dependencies

5.) Functional Dependencies

Projection:
Let R be a relation with sch(R) = (A_1, …, A_k).
Let (i_1, …, i_m) be distinct numbers from { 1, …, k }.
Let t = (v_1,…, v_k) be a tuple from val(R).

The projection to (A_{i_1}, …, A_{i_m}) is defined as

$$\pi_{A_{i_1},\ldots,A_{i_m}}(t) = (v_{i_1}, \ldots, v_{i_m})$$

projection = “select only the given columns”

5.) Functional Dependencies

Let X, Y be *non-empty sets* of attributes of a given table T.
The functional dependency (FD) $X \rightarrow Y$ means that for any two tuples t_1, t_2 in val(T):

$$\text{if } \pi_X(t_1) = \pi_X(t_2) \text{ then } \pi_Y(t_1) = \pi_Y(t_2).$$

5.) Functional Dependencies

Let X, Y be *non-empty sets* of attributes of a given table T.
The functional dependency (FD) X → Y means that for any two tuples t_1, t_2 in val(T):

$$\text{if } \pi_X(t_1) = \pi_X(t_2) \text{ then } \pi_Y(t_1) = \pi_Y(t_2).$$

“if two tuples agree on their X-values, then they also agree on their Y-values”

“the X-values determine the Y-values”

$\pi_{X,Y}(\mathrm{val}(T))$ is a **function** from X to Y

5.) Functional Dependencies

Let X, Y be non-empty sets of attributes of a given table T.
The functional dependency (FD) X → Y means that for any two tuples t_1,t_2 in val(T):

$$\text{if } \pi_X(t_1) = \pi_X(t_2) \text{ then } \pi_Y(t_1) = \pi_Y(t_2).$$

"if two tuples agree on their X-values, then they also agree on their Y-values"

Important:

FDs are determined by the semantics of the table.
Not by any particular instance of a table!

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

What are the (interesting) functional dependencies?

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

What are the (interesting) functional dependencies?

Scenario-1: we assume that Titles are unique, i.e., there cannot be two different ISBN's with the same title. Then:

Title $\rightarrow$ ISBN	"Title gives me the ISBN"
ISBN $\rightarrow$ Title	"ISBN gives me the Title"

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

What are the (interesting) functional dependencies?

Scenario-1: we assume that Titles are unique, i.e.,
there cannot be two different ISBN's with the same title.
Then:

Title $\rightarrow$ ISBN "Title gives me the ISBN"
ISBN $\rightarrow$ Title "ISBN gives me the Title"

these are sets of attributes, but we often
omit the brackets '{' '}'.

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

What are the (interesting) functional dependencies?

Scenario-1: we assume that Titles are unique, i.e., there cannot be two different ISBN's with the same title. Then:

Title → ISBN   "Title gives me the ISBN"
ISBN → Title   "ISBN gives me the Title"

these are sets of attributes, but we often omit the brackets '{' '}'.

Question

What are the candidate keys of this table?

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

Scenario-2: Titles are **not** unique, there may be two different ISBNs with the **same title**.

What are now the (interesting) functional dependencies?

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

Scenario-2: Titles are **not** unique, there may be two different ISBNs with the **same title**.

What are now the (interesting) functional dependencies?

ISBN $\rightarrow$ Title — this one has not changed

any other one?

Functional Dependencies

sch(Book) = (ISBN, Title, Author)

Scenario-2: Titles are **not** unique, there may be two different ISBNs with the **same title**.

What are now the (interesting) functional dependencies?

ISBN → Title — this one has not changed

any other one?

Question

What are the candidate keys of this table?

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10
cs477	Smith	Mo, 9-11

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10
cs477	Smith	Mo, 9-11

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Course —> Prof

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10
cs477	Smith	Mo, 9-11

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Course —> Prof

any other one?

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10
cs477	Smith	Mo, 9-11

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Course —> Prof

Prof, Time —> Course

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10
cs477	Smith	Mo, 9-11

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Course —> Prof

Prof, Time —> Course

Question

What are the candidate keys of this table?

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Does this table contain redundancy?

Functional Dependencies

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	**Knuth**	Mo, 9-11
cs101	**Knuth**	Fr, 14-16
cs311	Knuth	Th, 8-10

Important assumption: each course is taught by exaclty one Prof.

What are the (interesting) functional dependencies?

Does this table contain redundancy? — **YES!**

Superkeys and Keys

If X → Y holds and $X \cup Y = \mathrm{Sch}(T)$
then X is a superkey of T.

sch(T), but seen as a **set**

Superkeys and Keys

If X → Y holds and $X \cup Y = \mathrm{Sch}(T)$
then X is a superkey of T.

sch(T), but seen as a **set**

A key (aka "candidate key") is a **minimal superkey**.

if we remove any element, then it
ceases to be a superkey

There can be *several keys* (minimal superkeys).
That is why we often say "candidate key" instead of "key."

Superkeys and Keys

sch(Teach) = (Course, Prof, Time)

Course	Prof	Time
cs101	Knuth	Mo, 9-11
cs101	Knuth	Fr, 14-16
cs311	Knuth	Th, 8-10

Important assumption: each course is taught by exaclty one Prof.

What are the candidate keys of this table?

Course —> Prof

Prof, Time —> Course

Prof, Time
Course, Time

Functional Dependencies $\leftrightarrow$ Keys

Functional dependencies are a generalization of **key constraints**:

$$A_1, \ldots, A_n \text{ is a set of identifying attributes}^{11} \text{ (“superkey”)}$$
$$\text{in relation } R(A_1, \ldots, A_n, B_1, \ldots, B_m).$$
$$\Leftrightarrow$$
$$A_1 \ldots A_n \rightarrow B_1 \ldots B_m \text{ holds.}$$

Conversely, functional dependencies can be explained with keys.

$$A_1 \ldots A_n \rightarrow B_1 \ldots B_m \text{ holds for } R.$$
$$\Leftrightarrow$$
$$A_1, \ldots, A_n \text{ is a set of identifying attributes in } \pi_{A_1,\ldots,A_n,B_1,\ldots B_m}(R).$$

→ Functional dependencies are **“partial keys”**.

→ A goal of this chapter is to turn FDs into **real keys**, because key constraints can easily be enforced by a DBMS.

[11] If the set is also minimal, $A_1, \ldots, A_n$ is a key (↗ slide 53).

Functional Dependencies

What does it mean "interesting" functional dependency?

ISBN → Title

also implies that

ISBN, Author → Title "ISBN and Author give me the Title"
(in fact, we do not need the Author)

Functional Dependencies

What does it mean "interesting" functional dependency?

ISBN → Title

also implies that

ISBN, Author → Title "ISBN and Author give me the Title"
(in fact, we do not need the Author)

ISBN, Author, Title → Title

Functional Dependencies

What does it mean "interesting" functional dependency?

ISBN → Title

also implies that

ISBN, Author → Title "ISBN and Author give me the Title"
(in fact, we do not need the Author)

ISBN, Author, Title → Title

ISBN, Author, Title → Title, Author

Functional Dependencies

What does it mean “interesting” functional dependency?

ISBN → Title

also implies that

ISBN, Author → Title “ISBN and Author give me the Title”
(in fact, we do not need the Author)

ISBN, Author, Title → Title

ISBN, Author, Title → Title, Author

ISBN, Author, Title → Title, Author, ISBN

Functional Dependencies

What does it mean "interesting" functional dependency?

ISBN → Title

"interesting"

also implies that

ISBN, Author → Title

"ISBN and Author give me the Title"
(in fact, we do not need the Author)

ISBN, Author, Title → Title

ISBN, Author, Title → Title, Author

"not so interesting"

ISBN, Author, Title → Title, Author, ISBN

Functional Dependencies

Consider a table T with three columns:
sch(T) = (A, B, C)

How many different functional dependencies exist for such a table T?

5.) Functional Dependencies

Consider a table T with three columns:
sch(T) = (A, B, C)

How many different functional dependencies exist for such a table T?

a.) 8
b.) 27
c.) 49
d.) 64

5.) Functional Dependencies

Consider a table T with three columns:
sch(T) = (A, B, C)

How many different functional dependencies exist for such a table T?

a.) 8
b.) 27
c.) 49
d.) 64?

correct

$$\underline{(2^3 - 1)} * (2^3 - 1) = 7*7 = 49$$

number of **non-empty** sets
with at most 3 elements

5.) Functional Dependencies

A functional dependency $X \rightarrow Y$ is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?

5.) Functional Dependencies

A functional dependency $X \rightarrow Y$ is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?
30

5.) Functional Dependencies

A functional dependency X → Y is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?
30

A functional dependency X → Y is completely non-rivial,

if $X \cap Y = \emptyset$.

How many completely non-trivial FD for a table w. 3 columns?

5.) Functional Dependencies

A functional dependency X → Y is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?
30

A functional dependency X → Y is completely non-rivial,

if $X \cap Y = \emptyset$.

How many completely non-trivial FD for a table w. 3 columns?
12

ab → c	b → a
ac → b	b → c
bc → a	b → ac
a → b	c → b
a → c	c → a
a → bc	c → ab

5.) Functional Dependencies

A functional dependency X → Y is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?
30

A functional dependency X → Y is completely non-rivial,

if $X \cap Y = \emptyset$.

How many completely non-trivial FD for a table w. 3 columns?
12

ab → c	b → a
ac → b	b → c
bc → a	b → ac
a → b	c → b
a → c	c → a
a → bc	c → ab

Still: if a → b holds, then we **do not care** that also ac → b holds.

5.) Functional Dependencies

A functional dependency X → Y is trivial, if Y is a subset of X.

How many non-trivial FDs are there for a table with 3 columns?
30

A functional dependency X → Y is completely non-rivial,

if $X \cap Y = \emptyset$.

How many completely non-trivial FD for a table w. 3 columns?
12

ab → c	b → a
ac → b	b → c
bc → a	b → ac
a → b	c → b
a → c	c → a
a → bc	c → ab

Still: if a → b holds, then we **do not care** that also ac → b holds.

The FD a → b **implies** ac → b
(and it is strictly smaller).

Functional Dependencies

A functional dependency with m attributes on the right-hand side

$$A_1 \ldots A_n \rightarrow B_1 \ldots B_m$$

is **equivalent** to the m functional dependencies

$$\begin{array}{ccc} A_1 \ldots A_n & \rightarrow & B_1 \\ \vdots & & \vdots \\ A_1 \ldots A_n & \rightarrow & B_m \end{array}$$

Often, functional dependencies **imply** one another.

→ We say that a set of FDs $\mathcal{F}$ **entails** another FD f if the FDs in $\mathcal{F}$ guarantee that f holds as well.

Reasoning over Functional Dependencies

Intuitively, we want to (re-)write relational schemas such that

- **redundancy is minimized** (and thus also update anomalies) **and**
- the system can still guarantee the **same integrity constraints**.

Functional dependencies allow us to **reason** over the latter.

Closure of a Set of Functional Dependencies

Given a set of functional dependencies $\mathcal{F}$, the set of all functional dependencies entailed by $\mathcal{F}$ is called the **closure of** $\mathcal{F}$, denoted $\mathcal{F}^+$:[12]

$$\mathcal{F}^+ := \left\{ \alpha \rightarrow \beta \mid \alpha \rightarrow \beta \text{ entailed by } \mathcal{F} \right\} \ .$$

Closures can be used to express **equivalence** of sets of FDs:

$$\mathcal{F}_1 \equiv \mathcal{F}_2 \ \Leftrightarrow \ \mathcal{F}_1^+ = \mathcal{F}_2^+ \ .$$

If there is a way to **compute** $\mathcal{F}^+$ for a given $\mathcal{F}$, we can test

- whether a given FD $\alpha \rightarrow \beta$ is entailed by $\mathcal{F}$ ($\leadsto \alpha \rightarrow \beta \overset{?}{\in} \mathcal{F}^+$)
- whether two sets of FDs, $\mathcal{F}_1$ and $\mathcal{F}_2$, are equivalent.

[12] Let α, β, ... denote sets of attributes.

6.) Armstrong Axioms

Armstrong Axioms

$\mathcal{F}^+$ can be computed from $\mathcal{F}$ by repeatedly applying the so-called **Armstrong axioms** to the FDs in $\mathcal{F}$:

- **Reflexivity:** ("trivial functional dependencies")
  $$\text{If } \beta \subseteq \alpha \text{ then } \alpha \rightarrow \beta.$$
- **Augmentation:**
  $$\text{If } \alpha \rightarrow \beta \text{ then } \alpha\gamma \rightarrow \beta\gamma.$$
- **Transitivity:**
  $$\text{If } \alpha \rightarrow \beta \text{ and } \beta \rightarrow \gamma \text{ then } \alpha \rightarrow \gamma.$$

It can be shown that the three Amstrong axioms are **sound** and **complete**: exactly the FDs in $\mathcal{F}^+$ can be generated from those in $\mathcal{F}$.

Armstrong Axioms

$\mathcal{F}^+$ can be computed from $\mathcal{F}$ by repeatedly applying the so-called **Armstrong axioms** to the FDs in $\mathcal{F}$:

- **Reflexivity:** ("trivial functional dependencies")

$$\text{If } \beta \subseteq \alpha \text{ then } \alpha \rightarrow \beta .$$

- **Augmentation:**

$$\text{If } \alpha \rightarrow \beta \text{ then } \alpha\gamma \rightarrow \beta\gamma .$$

- **Transitivity:**

$$\text{If } \alpha \rightarrow \beta \text{ and } \beta \rightarrow \gamma \text{ then } \alpha \rightarrow \gamma .$$

It can be shown that the three Amstrong axioms are **sound** and **complete**: exactly the FDs in $\mathcal{F}^+$ can be generated from those in $\mathcal{F}$.

Exercise: Show that $A_1 \ldots A_n \rightarrow B_1 \ldots B_m$

is **equivalent** to the m functional dependencies

$$\begin{array}{ccc} A_1 \ldots A_n & \rightarrow & B_1 \\ \vdots & & \vdots \\ A_1 \ldots A_n & \rightarrow & B_m \end{array}$$

using the Armstrong Axioms.

Kurs

Datenbankgrund
und Model

End of this Lecture

Sebast rsität Bremen

remen.de

ersemester 2023

22.5.2023

Vorlesung 5: SQL Wrap Up &
Functional Dependencies