Data Base Systems

VU 184.686, WS 2020

SQL Details

Anela Lolić

Institute of Logic and Computation, TU Wien

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Acknowledgements

The slides are based on the slides (in German) of Sebastian Skritek.

The Section on Views is based on Chapter 4 of
(Kemper, Eickler: Datenbanksysteme – Eine Einführung)

The remaining topics are based on the respective chapters in the PostgreSQL Online-Documentation:
`https://www.postgresql.org/docs/current/static/`

SQL Details

1. Overview
2. Decomposing Complex Queries and Recursive Queries
3. Views
4. Window Functions
5. Sequences
6. (Other) Data Types and Built-In Functions
7. Learning Objectives

Topics

1. Overview
2. Decomposing Complex Queries and Recursive Queries
3. Views
4. Window Functions
5. Sequences
6. (Other) Data Types and Built-In Functions
7. Learning Objectives

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

7. Learning Objectives

Decomposing Complex Queries

problem:

- nested SQL queries sometimes become confusing and unclear
- in many cases very similar sub queries

Decomposing Complex Queries

problem:

- nested SQL queries sometimes become confusing and unclear
- in many cases very similar sub queries

Example (nested queries)

find all those students, that have the best average grades

```
SELECT matrNr, avg(grade)
FROM examine
GROUP BY matrNr
HAVING avgGrade >= all (SELECT avg(grade)
                        FROM examine
                        GROUP BY matrNr)
```

Decomposing Complex Queries

Example (SQL WITH statement (slight "overkill"))

find all those students, that have the best average grades

```
WITH avgGrade AS (
  SELECT matrNr, avg(grade) AS dsN
  FROM examine
  GROUP BY matrNr ),
 bestavgGrade AS (
  SELECT matrNr, dsN
  FROM avgGrade
  WHERE dsn = (SELECT max(dsN)
               FROM avgGrade))
SELECT matrNr, dsN
FROM bestavgGrade
```

SQL: `SELECT` in `WITH`

`WITH` queries (also Common Table Expressions (CTE)):

- allow for the definition of temporary "tables"
- only visible and usable in the current query
- useful for structuring the query

SQL: SELECT in WITH

WITH queries (also Common Table Expressions (CTE)):

- allow for the definition of temporary "tables"
- only visible and usable in the current query
- useful for structuring the query

distinction: WITH and WITH RECURSIVE

WITH "syntactic sugar" — no additional expressive power

WITH RECURSIVE allows for "recursive" queries — was not possible before

Recursive Queries

problem: SQL queries we have had so far do not allow for querying transitive properties:

Example

- ancestors or descendants from the parent-child relationship
- all preconditions (and their preconditions, and . . .) of a lecture
- all answers or contributions to a discussion
- reachability in graphs
- . . .

Recursive Queries: General Structure

- query that is able to work on its own output

```
WITH RECURSIVE tabName(attrList) AS (
    non recursive part
  UNION [ ALL ]
    redursive part
    )
SELECT ...
```

- only the recursive part may link to its own output
- both parts contain sub queries (SELECT)
- output has to correspond to *attrList*
- *attrList* contains only names

Recursive Queries: Semantics (Idea)

1. non recursive part constructs a result set T_0
2. recursive part constructs a result set T_{i+1} from input T_i
3. step 2 is repeated until $T_{i+1} = \emptyset$
4. entire result set corresponds to `UNION` resp. `UNION ALL` of $T_0, \ldots, T_n$

remark:

- when using `UNION` (instead of `UNION ALL`): $T_i \cap \bigcup_{j=0}^{i-1} T_j = \emptyset$
- attention: in case the recursive part is generating always (new) tuples: endless loop

Recursive Queries: Semantics

- three tuple sets: result, *working table*, *intermediate table*

Recursive Queries: Semantics

- three tuple sets: result, *working table*, *intermediate table*

execution:

1. evaluate the non recursive part
   - in case of `UNION`: eliminate duplicates
   - copy remaining tuples both into the result set and in the *working table*

Recursive Queries: Semantics

- three tuple sets: result, *working table*, *intermediate table*

execution:

2. as long as the *working table* is not empty repeat:
   1. evaluate the recursive part:
      - *working table* is used as input for the self reference
      - store the result in the *intermediate table*
   2. in case of `UNION`: remove all duplicates from *intermediate table* and tuples that are already contained in the result set
   3. insert content of *intermediate table* into the result set
   4. replace content of the *working table* with the one of the *intermediate table*, then empty *intermediate table*

Recursive Queries (Example)

Example (all numbers from 1 to 100)

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT n FROM t;
```

Recursive Queries (Example)

Example (all preconditions of lecture 5216)

```
WITH RECURSIVE pred(v) AS (
    SELECT predNr
    FROM presuppose
    WHERE succNr = 5216
  UNION
    SELECT presuppose.predNr
    FROM pred, presuppose
    WHERE pred.v=presuppose.succNr
)
SELECT v FROM pred;
```

Recursive Queries (Example)

remark: self reference is not permitted in nested queries

Example (all preconditions of lecture 5216)

```
WITH RECURSIVE pred(v) AS (
    SELECT predNr FROM presuppose
    WHERE succNr = 5216
  UNION
    SELECT predNr
    FROM presuppose
    WHERE succNr in (SELECT v FROM pred)
) SELECT v FROM pred;
```

results in: `ERROR: recursive reference to query "pred"must not appear within a subquery LINE 8: WHERE succNr in (SELECT v FROM pred)`

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views
3.1 Overview and Definition
3.2 Realization of the Generalization
3.3 Updates on Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views
3.1 Overview and Definition
3.2 Realization of the Generalization
3.3 Updates on Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

Views ...

- ... are stored queries, that are available as virtual tables
- ... originally used only for the reading access, but with certain constraints also for updates
- ... are constructed dynamically at each access.
- ... are not part of the physical schema

Views ...

use:

- concept for the adaptation to several users
- simplification of queries
- realization of the generalization

Views ...

use:
- concept for the adaptation to several users
- simplification of queries
- realization of the generalization

DBMS proprietary:
- materialized views (ORACLE):
  pre-compiled non virtual query
- indexed views (SQL Server):
  additionally stored index

used to speed-up frequently executed queries

Adjustment to Several Usergroups

Example

due to data security in general only the exam list but not the grade are readable:

```
create view examineView as
select matrNr, lecNr, persNr
from examine;
```

Simplification of Queries

Example

avoiding constant use of join

$$student \bowtie attend \bowtie lecture \bowtie professor$$

```
create view
  StudProf(Sname, semester, title, Pname) as
select s.name, s.semester, v.title, p.name
from student s, attend h,
     lecture v, professor p
where s.matrNr=h.matrNr and
      h.lecNr=v.lecNr and
      v.heldBy = p.persNr;
```

Simplification of Queries

Example

avoiding constant use of join

$$student \bowtie attend \bowtie lecture \bowtie professor$$

```
create view
  StudProf(Sname, semester, title, Pname) as
select s.name, s.semester, v.title, p.name
from student s, attend h,
     lecture v, professor p
where s.matrNr=h.matrNr and
      h.lecNr=v.lecNr and
      v.heldBy = p.persNr;
```

find name and semester of all students of Sokrates

```
select distinct name, semester from StudProf
where Pname='Sokrates';
```

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views
3.1 Overview and Definition
3.2 Realization of the Generalization
3.3 Updates on Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

Realization of Generalization

realization of inclusion and inheritance via views:

- tuple of sub type should belong automatically to the super type
- attributes of the super type: should be inherited automatically

Realization of Generalization

realization of inclusion and inheritance via views:

- tuple of sub type should belong automatically to the super type
- attributes of the super type: should be inherited automatically

$$\Rightarrow$$

define either sub type or super type as view
decision based on access frequency

Realization of Generalization

Example (sub types as view)

relations for storing the information
(super type + additional information for sub types):

```
create table employee(
                persNr integer not null,
                name varchar(30) not null);
create table profData (
                persNr integer not null,
                rank character(2),
                room integer);
create table assiData (
                persNr integer not null,
                field varchar(30),
                boss integer);
```

Realization of Generalization

Example (sub types as view)

views with entire information about the sub types:

```
create view professor as select *
from employee natural join profData;

create view assistant as select *
from employee natural join  assiData;
```

Realization of Generalization

Example (sub types as view)

views with entire information about the sub types:

```
create view professor as select *
from employee natural join profData;

create view assistant as select *
from employee natural join  assiData;
```

only the relation `employee` as well as views `professor` and `assistant` visible

Realization of Generalization

Example (super types as view)

```
create table professor(
                persNr integer not null,
                name varchar(30) not null,
                rank character (2),
                room integer);
create table assistant(
                persNr integer not null,
                name varchar(30) not null,
                field varchar(30),
                boss integer);
create table otherEmpl(
                persNr integer not null,
                name varchar(30) not null);
```

Realization of Generalization

Example (super types as view)

```
create view employee as
(select persNr, name from professor) union
(select persNr, name from assistant) union
(select * from otherEmpl);
```

idea: only relations `professor` and `assistant` and the view `employee` visible

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views
3.1 Overview and Definition
3.2 Realization of the Generalization
3.3 Updates on Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

Updates on Views

problematic: `insert`, `update`, `delete` on a view

Example (views incapable of update)

```
create view
  howDifficultAsExaminer(persNr, average) as
select persNr, avg(grade)
from examine
group by persNr;
```

Updates on Views

problematic: `insert`, `update`, `delete` on a view

Example (views incapable of update)

```
create view
  howDifficultAsExaminer(persNr, average) as
select persNr, avg(grade)
from examine
group by persNr;
```

```
create view lecturesView as
select title, SWS, name
from lectures, professor
where heldBy = persNr;
```

Updates on Views

restriction of views capable of update in SQL to (SQL-92):

- only one basic relation
- key has to exist
- no aggregate functions, grouping and duplicate elimination

Updates on Views

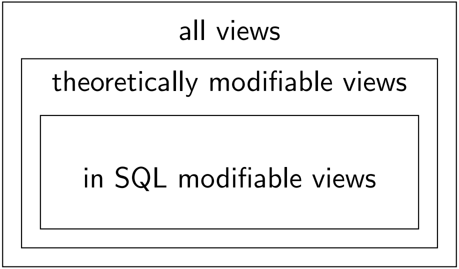
restriction of views capable of update in SQL to (SQL-92):

- only one basic relation
- key has to exist
- no aggregate functions, grouping and duplicate elimination

extension with SQL-99 (idea: consider key):

- a field of a view can be updated when
  - it can be assigned uniquely to a basic relation and
  - the corresponding key is part of the view
- distinction between views capable of update and views that allows for inserting new tuples

Updates on Views



Topics

1. Overview
2. Decomposing Complex Queries and Recursive Queries
3. Views
4. Window Functions
5. Sequences
6. (Other) Data Types and Built-In Functions
7. Learning Objectives

From an Exercise...

Example

```
SELECT floorNr, type
FROM   hive
WHERE (SELECT COUNT(*) OVER ()
      FROM workerBee NATURAL JOIN pollinate
      WHERE worksinType = type AND ...
      GROUP BY fieldID, town LIMIT 1)
        < 10 AND
      ...;
```

Window Functions: OVER ()

- allows for the calculation of (for instance aggregate-) functions over defined window
- grouping of rows not necessary

Window Functions: OVER ()

- allows for the calculation of (for instance aggregate-) functions over defined window
- grouping of rows not necessary

```
function_name ( args )
  [ FILTER ( WHERE filter_clause ) ]
  OVER ( window_definition )
```

- in SELECT or ORDER BY

here only basics, there are also other functionalities

Aggregate Functions without Grouping

Example

find for all students the number of attended lectures

```
SELECT DISTINCT matrnr,
  count(*) OVER (PARTITION BY matrnr)
  as number
FROM attend
ORDER BY number DESC;
```

Avoiding Nested SELECT

Example

find for all students all attended lectures and the number of attended lectures

```
SELECT matrnr, lecNr,
  count(*) OVER (PARTITION BY matrnr)
  as number
FROM attend
ORDER BY number DESC;
```

Evaluation Order of Window Functions

- evaluation logically after `WHERE, GROUP BY, HAVING`
- corresponding removed rows are not considered

Evaluation Order of Window Functions

- evaluation logically after WHERE, GROUP BY, HAVING
- corresponding removed rows are not considered

Example

find for all students the number of attended lectures with lecNr bigger than 5001

```
SELECT DISTINCT matrnr,
  count(*) OVER (PARTITION BY matrnr)
  as number
FROM attend
WHERE lecNr > 5001
ORDER BY number DESC;
```

Special Window Functions

- in addition to “normal” functions as window functions there are also extra functions
- have to be used via `OVER`
- allow for the computation of values within a partition
- for instance `row_number()`, `rank()`, ...

Example: rank()

- assigns a “ranking” to each entry

Example

find for each semester those students that attend the most lectures

```
SELECT * FROM (
SELECT matrnr, count(*) as numberVO, sem,
  rank() OVER
   (PARTITION BY sem ORDER BY count(*) DESC)
   as pos
FROM attend NATURAL JOIN student
GROUP BY matrnr, sem
ORDER BY numberVo DESC) as subq WHERE pos=1;
```

Window Frame

- in addition to the entire partition there is a window frame for each row
- subset of partition, depending on sorting:
  - without `ORDER BY`: entire partition
  - with `ORDER BY`: all rows from the beginning until current row, additionally all rows with same `ORDER BY` value

Example

```
SELECT matrnr, lecNr, sws,
  sum(sws) OVER (PARTITION BY matrnr
    ORDER BY lecNr)
FROM attend NATURAL JOIN lectures
ORDER BY matrnr;
```

Topics

1. Overview
2. Decomposing Complex Queries and Recursive Queries
3. Views
4. Window Functions
5. Sequences
6. (Other) Data Types and Built-In Functions
7. Learning Objectives

Idea

problem:

- in some cases unique (integer) values are needed, whose specific value is not relevant
- example: invoice numbers, visitor numbers, numbering of entries, . . .

Idea

problem:

- in some cases unique (integer) values are needed, whose specific value is not relevant
- example: invoice numbers, visitor numbers, numbering of entries, . . .

solution (in SQL): `SEQUENCE`

- constructs numerical sequence based on given properties (domain, step size, ascending/descending)
- no value is assigned twice
- in some circumstances numbers cannot be assigned, or cannot be assigned in chronological order
- modifications are kept also with `ROLLBACK`

Constructing and Using Sequences

```
CREATE SEQUENCE name
[INCREMENT [ BY ] increment]
[MINVALUE minvalue | NO MINVALUE]
[MAXVALUE maxvalue | NO MAXVALUE]
[START [WITH] start] [CACHE cache]
[[NO] CYCLE]
```

Constructing and Using Sequences

```
CREATE SEQUENCE name
[INCREMENT [ BY ] increment]
[MINVALUE minvalue | NO MINVALUE]
[MAXVALUE maxvalue | NO MAXVALUE]
[START [WITH] start] [CACHE cache]
[[NO] CYCLE]
```

Zugriff:

`nextval`(*sname*) increases the sequence and returns current value

`currval`(*sname*) returns the last value which was read with `nextval` in the session

`setval`(*sname*, *val*) sets the counter to the given value

SEQUENCE (Example)

Example

```
CREATE SEQUENCE persnr_seq
 START WITH 2200 INCREMENT BY 5
 MINVALUE 2200 MAXVALUE 15000
 NO CYCLE;

INSERT INTO professor VALUES(
 nextval('persnr_seq'), 'Nietzsche','C3',205);
INSERT INTO professor VALUES(
 nextval('persnr_seq'), 'Sun Tzu','C3',206);
SELECT curval('persnr_seq');
```

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

7. Learning Objectives

(Postgre)SQL Data Types

- numeric data types: `integer`, `int`, `numeric(p,s)`, ...
- strings: `char(n)`, `varchar(n)`, text, ...
- date and time data types: `date`, `timestamp`, ...
- enumerating types: `ENUM`
- geometric data types: `point`, `line`, `circle`, ...
- XML and JSON data types
- arrays
- ...

(Postgre)SQL Data Types

- numeric data types: `integer`, `int`, `numeric(p,s)`, ...
- strings: `char(n)`, `varchar(n)`, text, ...
- date and time data types: `date`, `timestamp`, ...
- enumerating types: `ENUM`
- geometric data types: `point`, `line`, `circle`, ...
- XML and JSON data types
- arrays
- ...

Example (`ENUM` data type)

```
CREATE TYPE rank AS ENUM ('C2', 'C3', 'C4');
```

Functions

we have already learned about some built-in functions

- aggregate functions
  (compute a value over several rows)
- occasionally also single row functions

other single row functions:

- character functions
- numeric functions
- converting functions
- date functions

Character Functions

- concatenation: of strings, for instance: `SELECT 'Postgre' || 'SQL'`
- `lower()` and `upper()`:
  converting to lower- and upper-case characters
- `length()`:
  returns the length of a strings.
- `substring(string from for)`:
  `substring('Thomas' from 2 for 3)` – returns 'hom'
- `replace (string from to)`:
  `replace('abcdefcdg', 'cd', 'XX')` – returns 'abXXefXXg'

Numeric Functions

- „everything that can be done with a calculator“, for instance
  - `sqrt`(x), `power`(x,y)
  - `exp`(x), `ln`(x), `log`(b, x)
  - `cos`(x), `sin`(x), `tan`(x), `atan`(x), etc.
  - `abs`(x), `sign`(x)
- `round()`:
  rounding (optional: to a certain number of decimal places)
  - `round(105.75)` – returns 106
  - `round(105.75, 1)` – returns 105.8
  - `round(105.75, -1)` – returns 110

Converting Functions

- purpose:
  transforming values between different data types
- for instance: `to_char()`, `to_number()`
  - `to_char(17)` – returns '17'
  - `to_char(12345.678, '99,999.99')` – returns '12,345.68'
  - `to_number('17')` – returns 17
  - `to_number('-12,345.67', '99,999.99')` – returns -12345.67
  - `to_number('123.45', '99.99')` – returns error!
  - `to_number('123.49', '999.9')` – returns 123.4
- many other functions

Data Type DATE

- used for storing dates (day, month, year)
- no time (SQL standard)
- CURRENT_DATE returns current date
  (according to SQL standard: begin of a transaction)
- to_char and to_date offer several formatting possibilities

to_char() Function

cast of DATE to a string

Example (to_char())

```
SELECT to_char(birthDate) FROM professor;
SELECT to_char(sent) FROM order;
```

to_char() Function

cast of DATE to a string

Example (to_char())

```
SELECT to_char(birthDate) FROM professor;
SELECT to_char(sent) FROM order;
```

Example (formatting via to_char())

```
to_char(birthDate, 'MONTH DD, YYYY')
to_char(birthDate, 'DD-MON-YYYY')
to_char(birthDate, 'Day, DD.MM.YY')
to_char(CURRENT_DATE, 'DD-MON-YYYY')
```

TO_DATE() Function

cast of a string to DATE.

Example (to_char())

```
UPDATE clients SET
  born = TO_DATE('12-JUN-1976')
  WHERE clientNr = 1001;
```

TO_DATE() Function

cast of a string to DATE.

Example (to_char())

```
UPDATE clients SET
  born = TO_DATE('12-JUN-1976')
  WHERE clientNr = 1001;
```

Example (formatting via to_char())

```
TO_DATE('12-JUN-1976')
TO_DATE('12.06.1976', 'DD.MM.YYYY')
TO_DATE('October 3, 1974', 'Month DD, YYYY')
```

AGE() Function

difference between two dates given as years/months/days

Example

```
SELECT AGE(CURRENT_DATE,
    TO_DATE('01 Sep 2012', 'DD Mon YYYY'));
-> "5 years 2 mons 14 days"
   (entered 15.11.2017)

SELECT AGE(
    TO_DATE('01 Nov 2012', 'DD Mon YYYY'),
    TO_DATE('05 Dec 2013', 'DD Mon YYYY'));
-> "-1 years -1 mons -4 days"
```

Data Type TIME and TIMESTAMP

TIME for storing the time

TIMESTAMP for storing date and time

CURRENT_TIME resp. CURRENT_TIMESTAMP returns current time resp. time and date (according to SQL standard: begin of transaction)

TO_CHAR resp. TO_TIMESTAMP functions offer several formatting possibilities

EXTRACT() Function

enables reading parts (day, month, ...) of a date

Example

```
SELECT EXTRACT (YEAR FROM Datum) AS year,
  EXTRACT (MONTH FROM Datum) AS month,
  EXTRACT (DAY FROM Datum) AS day
  FROM orders WHERE clientNr = 1003;
```

Topics

1. Overview

2. Decomposing Complex Queries and Recursive Queries

3. Views

4. Window Functions

5. Sequences

6. (Other) Data Types and Built-In Functions

7. Learning Objectives

Learning Objectives

- How can a (longer) SQL query be structured?
- How do recursive SQL `SELECT` statements work? (understanding and composing)
- What are views?
- What are views used for?
- How are they constructed?
- How do `UPDATE`s behave on views?
- What is a `SEQUENCE`, and how do we construct and use it?