Technische
Hochschule
Rosenheim

Chapter 3 – Introduction to SQL

Databases lectures

Prof. Dr Kai Höfig

Contents

- **3.1 SQL DDL data definition operations**
- 3.2 SQL DML change operations
- 3.3 SQL DML query operations

SQL

SQL = Structured Query Language

- Language with a relatively simple structure
- Based on English colloquial language or slang (a lot of “syntactic sugar”)

SQL DDL: Data Definition Language

Manipulation of table schemas

- Creation/modification/deletion of
  - Table schemas
  - Databases
  - Views
  - Indexes

SQL DML: Data Manipulation Language

Manipulation of tuples (data sets)

- Change operations: Tuple Insertion/modification/deletion
- Query operations:
  - Grouping & sorting
  - Querying data
  - Transactions

SQL versions

- History
  - SEQUEL (1974, IBM Research Labs San Jose)
  - SEQUEL2 (1976, IBM Research Labs San Jose)
  - SQL (1982, IBM)
  - ANSI-SQL (SQL-86; 1986)
  - ISO-SQL (SQL-89; 1989; three languages Level 1, Level 2, + IEF)
  - (ANSI / ISO) SQL2 (adopted as SQL-92)
  - (ANSI / ISO) SQL3 (adopted as SQL:1999)
  - (ANSI / ISO) SQL:2003
  - SQL/XML:2006 – adds XML handling
  - SQL:2008
  - SQL:2011 latest version (not yet widely used)

- Each DBMS implements SQL with different details and extensions

  ➔ It is always necessary to refer to the documentation (available online)

  ➔ In the lecture: SQL:2008 (current SQL standard)

  ➔ In the exercise: Transact-SQL (SQL variant of Microsoft SQL Server)

Practical significance of SQL

- Diverse uses of SQL in business information systems, among other areas
  - Database development (create and maintain tables, views, rights, etc.)
  - Application development (manipulate and present data).
  - Website creation (dynamic websites, mostly accessed using scripting languages such as JavaScript, ASP.NET, PHP, etc.)
  - Mobile applications (iOS, Android, etc.)
  - Data Warehouses / Business Intelligence Systems
  - Corporate Information Management, esp. in ERP systems like SAP
  - etc.

- However, there are often different, simultaneous access options to the DB
  - DB management tools (such as SQL Server Management Studio) for creating DBs, granting/revoking access rights, backup, optimisation, etc.
  - Access via SQL from the applications

SQL DDL – important instructions (statements)

- **`create table`**
  - Create a new (empty) relation
  - Specify the integrity constraints
  - Store the information in the data dictionary

- **`drop table`**
  - Delete a/an (empty) relation
  - Delete the information from the data dictionary

- **`alter table`**
  - Add/delete attributes/integrity constraints of an existing relation
  - Update the information in the data dictionary
  - See exercise!

Example of `create table`

- Creating the known PRODUCER relation:

```
create table PRODUCER(
   Vineyard varchar(50),
   Growing_area varchar(20) not null,
   Region varchar(10),
   primary key (Vineyard))
```

Possible value ranges in SQL

- `integer` (or also **`integer4`**, **`int`**),
- `smallint` (or also **`integer2`**),
- `float(p)` (or also in short **`float`**),
- `decimal(p,q)` and `numeric(p,q)` each with **`q`** decimal places,
- `character(n)` (in short `char(n)`, if **`n`** = 1 also `char`) for strings of fixed length **`n`**,
- `character varying(n)` (in short `varchar(n)`) for variable length strings up to the maximum length **`n`**,
- `bit(n)` or `bit varying(n)` similarly for bit sequences (bit strings), and
- `date`, `time` or `timestamp` for date, time and combined date/time information
- …plus typically quite a few more depending on the DBMS!

Key constraints in SQL

- `primary key` identifies column(s) as primary key attribute
- `unique` identifies column(s) as (non-primary) key attribute
- If key consists of only one attribute: possible directly after attribute

```
create table PRODUCER(
Vineyard varchar(50) primary key,
Growing_area varchar(20) not null,
Region varchar(10))
```

Null values in SQL

- Special value **`null`**
  - represents the meaning "*value unknown*", "*value not applicable*" or "*value does not exist*", but does not belong to any value range
  - In SQL, null values are denoted by **`null`** or ⊥
- **`null`** can appear in all columns except
  - in primary key attributes and
  - those marked with **`not null`**
- **`create table`**
  - **`not null`** excludes null values as attribute values in specific columns
  - **`null`** allowed in null values column (rarely required)
  - note: also usable for `alter table`

More about data definition in SQL

- In addition to primary and foreign keys, SQL can specify:
  - with the `default` clause: default values for attributes,
  - with the `create domain` statement: user-defined value ranges and
  - with the `check` clause: additional local integrity constraints within the value ranges, attributes and relational schemas to be defined

➔ See SQL documentation for details

drop table

- Example: deleting the WINES relation

```
drop table WINES
```

- Condition: the relation cannot be referenced by referential integrity constraints

Peer Programming 1 – SQL DDL

- Introduced in
  - TSQL,
  - MS SQL Server 2016
  - MS SQL Server Management Studio

- Create table, drop table, alter table for an example scenario:

- Cocktails
  - What tables do we need?
  - What attributes do we need?
  - What are keys?

Contents

- 3.1 SQL DDL data definition operations
- **3.2 SQL DML change operations**
- 3.3 SQL DML query operations

Change operations in SQL

- `insert`
  Insert one or more tuples into a base relation or view

- `update`
  Change one or more tuples in a base relation or view

- `delete`
  Delete one or more tuples from a base relation or view

- Integrity constraints (key constraints, referential integrity, etc.) automatically checked by the DBMS
  - If violated by command: error message, command not executed

- With `select` also called CRUD operations
  (Create, Read, Update, Delete)

The **insert** statement

- Syntax

```
insert into relation [ (attribute₁, ..., attributeₙ) ]
values (constant₁, ..., constantₙ)
```

  - optional attribute list enables insertion of incomplete tuples

- Example

```
insert into PRODUCER
values ('Chateau Lafitte', 'Medoc', 'Bordeaux')
```

- Not all attributes specified → missing attributes become null/default

```
insert into PRODUCER (Vineyard, Region)
values ('Wairau Hills', 'Marlborough')
```

insert: inserting calculated data

- Syntax:

```
insert into relation [ (attribute_1, ..., attribute_n) ]
     SQL query
```

- Example:

```
insert into WINES
    select ProdID, ProdName, ’Red’, ProdYear, ’Chateau Lafitte’
    from SUPPLIER
    where SName = ’WineMerchant’
```

The delete statement

- Syntax:

```
delete from relation
[ where condition ]
```

- Example: deleting a tuple in the WINES relation:

```
delete from WINES
where WineID = 4711
```

More about **delete**

- The standard case is for deleting multiple tuples:

```
delete from WINES
where Colour = ’White’
```

- Deleting the entire relation:

```
delete from WINES
```

- Delete operations can result in violations of integrity constraints!
  - Example: violation of the foreign key property, if there are still wines from this producer:

```
delete from PRODUCER
where Region = ’Hesse’
```

The **`update`** statement

- Syntax

```
update relation
set    attribute_1 = expression_1
       ...
       attribute_n = expression_n
[ where condition ]
```

Example for **update**

WINES

WineID	Name	Colour	Vintage	Vineyard	Price
3456	Zinfandel	Red	2004	Helena	5.99
2171	Pinot Noir	Red	2001	Creek	10.99
3478	Pinot Noir	Red	1999	Helena	19.99
4711	Riesling Reserve	White	1999	Müller	14.99
4961	Chardonnay	White	2002	Bighorn	9.90

```
update  WINES
set     Price = Price * 1.10
where   Vintage < 2000
```

WINES

WineID	Name	Colour	Vintage	Vineyard	Price
3456	Zinfandel	Red	2004	Helena	5.99
2171	Pinot Noir	Red	2001	Creek	10.99
3478	Pinot Noir	Red	1999	Helena	21.99
4711	Riesling Reserve	White	1999	Müller	16.49
4961	Chardonnay	White	2002	Bighorn	9.90

More about `update`

- Implementation of single tuple operation using (primary) key:

```
update   WINES
set      Price = 7.99
where    WineID = 3456
```

- Caution: without `where` condition: changes the entire relation:

```
update   WINES
set      Price = 11
```

Peer Programming 2 – SQL DML change operations

- DML change operations for our drinks
  - Which mixed drinks do you know?
  - What ingredients do we need?

SQL DML – practical implementation of relational algebra

- Queries to relational DBMS are not made directly in relational algebra
- They are implemented as part of the SQL DML instead
- SQL implements the relational algebra operations relatively directly
- But with substantial syntactic sugar
- SQL DML thus consists of
  - Change operations: insert/update/delete statements
  - Query operations of relational algebra: select/union/except/intersect statements

SQL core – the SFW block

`select`	projection list arithmetic operations & aggregate functions
`from`	relations to be used, possible renaming
`where`	selection conditions, join conditions nested queries (once again an SFW block)
`group by`	grouping for aggregate functions
`having`	selection conditions for groups
`order by`	output order

here

Chapter “Advanced SQL”

Projection π in SQL

- Expression in relational algebra

$$\pi_{\text{WineID, Vineyard}}(\text{WINES})$$

- Queries in SQL

```
select   WINES.WineID, WINES.Vineyard
from     WINES
```

```
select   WineID, Vineyard
from     WINES
```

Selection σ in SQL

- Expression in relational algebra

$$\sigma_{\text{Vintage}>2000}(\text{WINES})$$

- Query in SQL

```
select   *
from     WINES
where    WINES.Vintage > 2000
```

```
select   *
from     WINES
where    Vintage > 2000
```

Combination of selection and projection

Query to a single table with selection and projection

- Expression in relational algebra

$$\pi_{\text{Name, Colour}}(\sigma_{\text{Vintage=2002}}(\text{WINES}))$$

- Query in SQL

```
select   WINES.Name, WINES.Colour
from     WINES
where    WINES.Vintage = 2002
```

```
select   Name, Colour
from     WINES
where    Vintage = 2002
```

Multiset semantics of SQL (1)

Most important difference between SQL and relational algebra

Relational algebra always has **set semantics**
➔ Duplicates in the result are automatically removed!

By default, SQL has **multiset semantics**
➔ Duplicates in the result are **not** automatically removed!

- Reason: performance! Removing duplicates is expensive: O(*n* log *n*)
- Explicit set semantics in SQL through `distinct`

Multiset semantics of SQL - example

- Expression in relational algebra: $\pi_{\text{Region}}(\text{PRODUCER})$

- Query/queries in SQL

```
select Region
from   PRODUCER
```

Region
South Australia
California
Bordeaux
Bordeaux
Hesse
California

```
select distinct Region
from            PRODUCER
```

Region
South Australia
California
Bordeaux
Hesse

- Multiset semantics are not possible in relational algebra!

Peer Programming 3 – SQL DML query operations

- Projection

Cross product × in SQL

- Expression in relational algebra

WINES × BOTTLE

- Possible queries in SQL

```
select   *
from     WINES, BOTTLE
```

```
select   *
from     WINES cross join BOTTLE
```

Natural join ▷◁ in SQL

- Expression in relational algebra: WINES ▷◁ PRODUCER
- Possible queries for natural join in SQL

1. Use of `natural join`

```
select *
from WINES natural join PRODUCER
```

2. Explicit join condition

```
select *
from WINES, PRODUCER
where WINES.Vineyard = PRODUCER.Vineyard
```

3. Use of `using`

```
select *
from WINES join PRODUCER using (Vineyard)
```

4. Use of `join on`

```
select *
from WINES join PRODUCER on WINES.Vineyard = PRODUCER.Vineyard
```

Combination of operations

- Expression in relational algebra

$$\pi_{\text{Name,Colour,Vineyard}}(\sigma_{\text{Vintage>2000}}(\text{WINES}) \bowtie \sigma_{\text{Region=“California”}}(\text{PRODUCER}))$$

- 2 possibilities for this query in SQL (there are many more!)

```
select   Name, Colour, WINES.Vineyard
from     WINES join PRODUCER on WINES.Vineyard = PRODUCER.Vineyard
where    Vintage > 2000 and
         Region = ’California’
```

```
select   Name, Colour, WINES.Vineyard
from     WINES, PRODUCER
where    Vintage > 2000 and
         Region = ’California’ and
         WINES.Vineyard = PRODUCER.Vineyard
```

Peer Programming 4 – SQL DML query operations

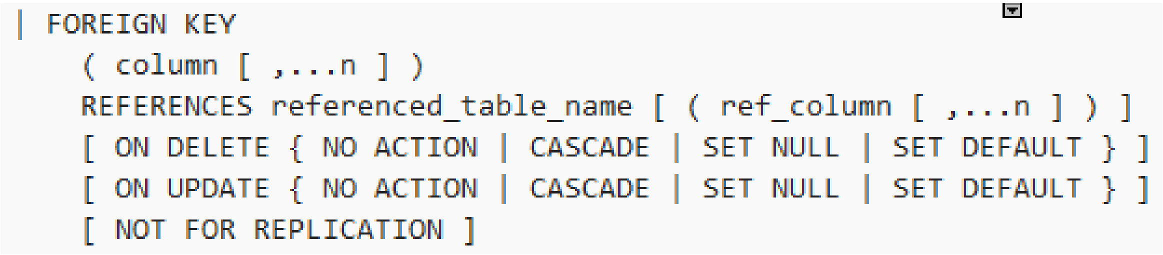
- Cross product and join

Revision

- DDL : creation of tables with attributes, keys and constraints
  - Recipes, ingredients, drinks
  - Subsequent addition of price
  - Null / Not null
  - Default value and check for alc.
- DML : change operations
  - Inserts for the tables
  - Inserts with select
  - Delete, update
  - Update multiple tuples
- DML: query operations
  - Simple selects
  - Cross product
  - Join

Chapter: Advanced SQL – Foreign Key Constraints

Included from another chapter



```
| FOREIGN KEY
    ( column [ ,...n ] )
    REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ NOT FOR REPLICATION ]
```

- NO ACTION
  - The database module generates an error, default value
- CASCADE
  - The corresponding cells are updated, or the row is deleted
- SET NULL
  - The corresponding cells are changed to null
- SET DEFAULT
  - The corresponding cells are set to the default value.

The from clause

The from clause

- Simplest form

select *

from relation$_1$, relation$_2$, …, relation$_n$

- Example:

```
select *
from   WINES
```

- If there is more than one relation, the Cartesian product is formed.
  - Example:

    ```
    select *
    from   WINES, PRODUCER
    ```

from : tuple variables for multiple access

- Introduction of tuple variables allows multiple access to a relation.
  - Behind each relation there can optionally be a tuple variable (and in between optionally **as**)

- Example:

```
select *
from   WINES as w1, WINES as w2
```

```
select *
from   WINES w1, WINES w2
```

➔ The columns are then called:

```
w1.WineID, w1.Name, w1.Colour, w1.Vintage, w1.Vineyard
w2.WineID, w2.Name, w2.Colour, w2.Vintage, w2.Vineyard
```

`from` : tuple variable for intermediate results

- “Intermediate relations” from SQL operations or an SFW block can be named using tuple variables
- Example:

```
select Result.Vineyard
from (WINES natural join PRODUCER) as Result
```

- `as` is once again optional

The **select** clause

The **select** clause

- Specifying the projection attributes

```
select [distinct] projection list
from ...
```

- With
  projection list := {*attribute* | *arithmetic expression* | *aggregate function* | * } [, …]

- And
  - *Attributes* of the relations behind **from**, optionally with a prefix, which specifies relation names or tuple variable names
  - *Arithmetic expressions* via attributes of these relations and matching constants
  - *Aggregate functions* via attributes of these relations

select: projection list *

- Special case of the projection list: *
  - returns all attributes of the relation(s) from the **from** part

- Example

```
select *
from WINES
```

select: **distinct** eliminates duplicates

- **distinct** eliminates duplicates
- Example:

```
select Name                      select distinct Name
from WINES                       from WINES
```

→ returns multiset

Name
La Rose GrandCru
Creek Shiraz
Zinfandel
Pinot Noir
Pinot Noir
Riesling Reserve
Chardonnay

→ returns set

Name
La Rose GrandCru
Creek Shiraz
Zinfandel
Pinot Noir
Riesling Reserve
Chardonnay

select: tuple variables and relation names

- Query

```
select Name
from   WINES
```

- is equivalent to

```
select WINES.Name
from   WINES
```

- and

```
select W.Name
from   WINES W
```

`select`: prefixes for uniqueness

- **Incorrect** example:

```
select Name, Vintage, Vineyard
from   WINES natural join PRODUCER
```

- Attribute `Vineyard` exists in both the table `WINES` as well as in `PRODUCER`!

- Correct example with prefix:

```
select Name, Vintage, PRODUCER.Vineyard
from   WINES natural join PRODUCER
```

select: tuple variables for uniqueness

- When using tuple variables, the name of a tuple variable can be used to qualify an attribute.

- Example:

```
select w1.Name, w2.Vineyard
from   WINES w1, WINES w2
```

`select/where`: scalar expressions

Scalar operations

- Numerical value ranges: such as +, -, * and /
- Strings: typical string operations such as
  - `char_length(str)`: current length of a string
  - `str1 || str2`: concatenation of strings str1 and str2
  - `substring(str, start, len)`: substring
  - `position(str1 in str2)`: position of the first occurrence of str1 in str2 (>=1; 0 if not contained)
- Date types & time intervals: operations such as +, -,*; functions such as
  - `current_date`: today's date
  - `current_time`: the current time
  - `year(d), month(d), day(d)`: year, month, day of a date
- Note:
  - Expressions can contain multiple attributes
  - Application is tuple by tuple: one result tuple is created for each input tuple

select/where: scalar expressions - examples

- Output of the names of all Grand Cru wines

```
select substring(Name from 1 for
    position(’GrandCru’ in Name) - 2)
from   WINES where Name like ’%GrandCru’
```

- Assumption: additional attribute `ProdDate` in WINES

```
alter table WINES add column ProdDate date

update WINES set ProdDate = date ’2004-08-13’
where  Name = ’Zinfandel’
```

- Query

```
select Name, year(current_date - ProdDate) as   Age
from   WINES
```

select/where: conditional expressions

- **case** statement: output of a value depending on the evaluation of a predicate

```
case
    when predicate$_1$ then expression$_1$
    ...
    when predicate$_{n-1}$ then expression$_{n-1}$
    [ else expression$_n$ ]
end
```

- Use in **select** and **where** clauses

```
select case
    when Colour = 'Red' then 'Red wine'
    when Colour = 'White' then 'White wine'
    else 'Other'
  end as Type_of_wine, Name from WINES
```

select/where: type conversion

- Explicit conversion of the expression type with cast

```
   cast(expression as type name)
```

- Example: int values as a string for the concatenation operator

```
select cast(Vintage as varchar) || ’er ’ ||
       Name as Designation
from   WINES
```

Peer Programming 5 – SQL DML query operations

- SELECT and FROM

The **`where`** clause

- The **`where`** clause

```
select ...from ...
where  condition
```

- Forms of the condition:
  - Comparison of an attribute with a constant:

    attribute θ *constant*

    possible comparison symbols θ depending on the value range, e.g. =, <>, >, <, >= and <=.
  - Comparison between two attributes with compatible value ranges:

    attribute1 θ *attribute2*
  - Logical connectors **`or`**, **`and`** and **`not`**

`where`: join condition

- Join condition has the form:

```
   relation1.attribute = relation2.attribute
```

- Example:

```
select Name, Vintage, PRODUCER.Vineyard
from   WINES, PRODUCER
where  WINES.Vineyard = PRODUCER.Vineyard
```

`where`: range selection

- Range selection

```
    attrib between constant1 and constant2
```

- is an abbreviation for

```
    attrib ≥ constant1 and
    attrib ≤ constant2
```

- thereby restricts attribute values to the closed interval [`constant1, constant2`]

- Example:

```
select * from WINES
where  Vintage between 2000 and 2005
```

where: wildcard selection (1)

- Notation

```
   attribute like special constant
```

- Pattern recognition in strings (search for multiple substrings)
- Special constant can contain the special symbols ‘%’ and ‘_’
  - ‘%’ stands for no character or any number of characters
  - ‘_’ stands for exactly one character

`where`: wildcard selection (2)

- Example

```
select *
from   WINES
where  Name like ’La Rose%’
```

- is an abbreviation for

```
select *
from   WINES
where  Name = ’La Rose’
       or Name = ’La RoseA’ or Name = ’La RoseAA’ ...
       or Name = ’La RoseB’ or Name = ’La RoseBB’ ...
       ...
       or Name = ’La Rose Grand Cru’ ...
       or Name = ’La Rose Grand Cru Classe’ ...
       or Name = ’La RoseZZZZZZZZZZZZZ’ ...
```

where: quantifiers and set comparisons

- Quantifiers: **all**, **any**, **some** and **exists**

- Notation

```
attribute θ { all | any | some } (
                                 select attribute
                                 from ...
                                 where ...)
```

- **all**: condition met if for **all** tuples of the inner SFW block, the θcomparison with attribute is **true**

- **any** or **some**: condition met if the θ comparison with at least one tuple of the inner SFW block is **true**
  any is the same as **all**

where: quantifiers: examples

- Determining the oldest wine

```
select *
from    WINES
where   Vintage <= all (
        select Vintage
        from   WINES)
```

- All vineyards that produce red wines

```
select *
from    PRODUCER
where   Vineyard = any (
        select Vineyard
        from   WINES
        where  Colour = ’Red’)
```

where: comparison of value sets

- Testing for equality of two sets with quantifiers alone is not possible
- Example: “Output out all producers that produce both red and white wines.”
- **Wrong** query

```
select Vineyard
from WINES
where Colour = ’Red’ and Colour = ’White’
```

- Correct formulation

```
select w1.Vineyard
from WINES w1, WINES w2
where w1.Vineyard = w2.Vineyard
and w1.Colour = ’Red’ and w2.Colour = ’White’
```

Peer Programming 6 – SQL DML query operations

- WHERE clause

Renaming β in SQL

- Expression in relational algebra

$$\beta_{\text{Name} \leftarrow \text{Wine}}(\text{RECOMMENDATION})$$

- Possible queries in SQL

```
select   RECOMMENDATION.Wine as Name
from     RECOMMENDATION
```

```
select   Wine as Name
from     RECOMMENDATION
```

Note: **`as`** is optional but is generally used

Set operations ∪, -, ∩ in SQL

- Union = **union**, difference = **except**, intersection = **intersect**

```
select   Surname
from     WINEMAKER
  union
select   Surname
from     CRITIC
```

```
select   Surname
from     WINEMAKER
  except
select   Surname
from     CRITIC
```

```
select   Surname
from     WINEMAKER
  intersect
select   Surname
from     CRITIC
```

Set operations

- Set operations are performed according to the position of the attributes, not according to the names of the attributes, and require compatible value ranges
  - both value ranges are equal or
  - both are value ranges based on `character` (regardless of the length of the strings) or
  - both are numerical value ranges (regardless of the exact type) such as `integer` or `float`

- Result schema = schema of the “left” relation

```
select A, B, C from R1
  union
select A, C, D from R2
```

- with union
  - Default case is elimination of duplicates (`union distinct`)
  - Without elimination of duplicates by `union all`

in predicate and nested queries

- Notation:

```
   attribute [not] in ( SFWblock )
```

- Example:

```
select Name
from   WINES
where  Vineyard in (
     select  Vineyard
     from    PRODUCER
     where   Region='Bordeaux')
```

`in`: (internal) evaluation of nested queries

- Evaluation of the inner query about the vineyards in Bordeaux
- Insertion of the result as a set of constants in the outer query after `in`
- Evaluation of the modified query

```
select Name
from   WINES
where  Vineyard in (
    ’Château La Rose’, ’Château La Point’)
```

Name
La Rose Grand Cru

`in`: negation of the `in` predicate

- “Simulation” of the difference operator

$$\pi_{\text{Vineyard}}(\text{PRODUCER}) - \pi_{\text{Vineyard}}(\text{WINES})$$

by SQL query

```
select Vineyard
from   PRODUCER
where  Vineyard not in (
    select Vineyard
    from WINES )
```

The **exists/not exists** predicate

- Simple form of nesting using **(not) exists**

```
    exists ( SFWblock )
```

- Returns **true** if the result of the inner query is not **empty**
(of course, **not exists** returns **true** if it is empty)

- Example: vineyards in Bordeaux without stored wines:

```
select *
from   PRODUCER e
where  Region = ’Bordeaux’ and not exists (
    select *
    from   WINES
    where  Vineyard = e.Vineyard)
```

Correlated subqueries

- `in/exists` often with **correlated subqueries** (synchronised subqueries)
  - i.e. in the inner query, the relation name or tuple variable name from the **`from`** part of the outer query is used

- Example: vineyards with 1999 red wines

```
select *
from PRODUCER
where  1999 in (
    select Vintage
    from WINES
    where Colour=’Red’ and WINES.Vineyard=PRODUCER.Vineyard)
```

  - Conceptual evaluation
    - Investigation of the first PRODUCER tuple in the outer query and insertion into the inner query
    - Evaluation of the inner query
    - Continue with second tuple . .

Correlated subqueries – transformation into join

- Correlated subqueries can be replaced by joins
  - Therefore, avoid correlated subqueries in general, as they are unclear!

- Example: vineyards with 1999 red wines

```
select *
from PRODUCER
where  1999 in (
   select Vintage
   from WINES
   where Colour=’Red’ and WINES.Vineyard=PRODUCER.Vineyard)
```

- Alternative formulation with join

```
select PRODUCER.*
from PRODUCER natural join WINES
where Vintage=1999 and Colour=’Red’
```

Peer Programming 7 – SQL DML query operations

- Projection
- Set operations
- IN predicate

Power of the SQL core

Relational algebra	SQL
Projection	**`select distinct`**
Selection	**`where`** without nesting
Cross product	**`cross join`** or “comma”

Join	**`from`**, **`where`**
	`from` with **`join`** or **`natural join`**

Renaming	**`from`** with tuple variable; **`as`**
Difference	**`where`** with nesting **`except`**
Intersection	**`where`** with nesting **`intersect`**
Union	**`union`**

Equivalent queries and query optimisation

- Many queries in SQL or relational algebra have equivalent queries, i.e. queries that return the same result
  - In the exercises, you will learn many examples (or find them yourself)
- Some queries can be evaluated much more efficiently than (equivalent) others
- Advantage of relational algebra: expressions can be transformed (by means of mathematical rules), whereby the equivalence is guaranteed
- Query optimiser (part of the query processor)
  - Part of every DBMS
  - Task: transform every SQL query (usually after prior transformation into relational algebra) into an equivalent expression that can be executed as efficiently as possible
  - Are among the most complex software modules in existence!