INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

A DIVERSION INTO SQL

- We will now temporarily shift focus from table **schema and state change** to **querying table states.**
- The ubiquitous `SELECT` construct forms the core of the SQL DML **query language.**
- `SELECT` embodies the principal data language operations we have studied before:

1. **Iteration** over rows of (multiple) tables, **filtering** based on predicates
2. **Computation** over column values (expression evaluation), **construction** of literal tables (`VALUES`)
3. **Grouping** of rows and **aggregation** of all (or groups of) values in a column
4. … *lots more* …

- We will use PostgreSQL's dialect of the SQL query language which implements a variant of SQL:2016, a recent language standard update (see ISO/IEC 9075 "Database Language SQL").

SQL: SELECT

SELECT

The SQL DML command SELECT retrieves rows from zero or more tables to construct one result table:

```
SELECT [ ALL | DISTINCT ] expression [ [ AS ] output_name ] [, …]
[ FROM   from_item [, …] ]
[ WHERE  condition ]
```

where *from_items* denote the source tables from which rows are drawn:

```
from_item:   (query) [ AS ] alias [ (column_name [, …]) ]
```

- Note that each source table itself is computed by a parenthesized SQL query expression *(query)*. These “queries in a query” are known as **nested queries** or **subqueries**.

SQL: SELECT

The semantics of SELECT can be quite precisely explained by a PyQL program. Consider these equivalent queries:

```
SELECT  e₁ AS c₁, …, eₙ AS cₙ
FROM    (q₁) AS v₁, (q₂) AS v₂, …, (qₜ) AS vₜ
WHERE   p
```

```
[ { c₁: e₁(v₁, v₂, …, vₜ), …, cₙ: eₙ(v₁, v₂, …, vₜ) }
    for v₁ in q₁
      for v₂ in q₂
        ⋱
          for vₜ in qₜ
            if p(v₁, v₂, …, vₜ) ]
```

1. In the PyQL query, q_i is the PyQL code for the SQL subquery q_i (*compositionality*)
2. $p(v_1, v_2, \dots, v_t)$ is a Boolean PyQL expression that refers to the variables v_i (i.e., the v_i *occur free* in expression p). Likewise for $e_i(v_1, v_2, \dots, v_t)$.

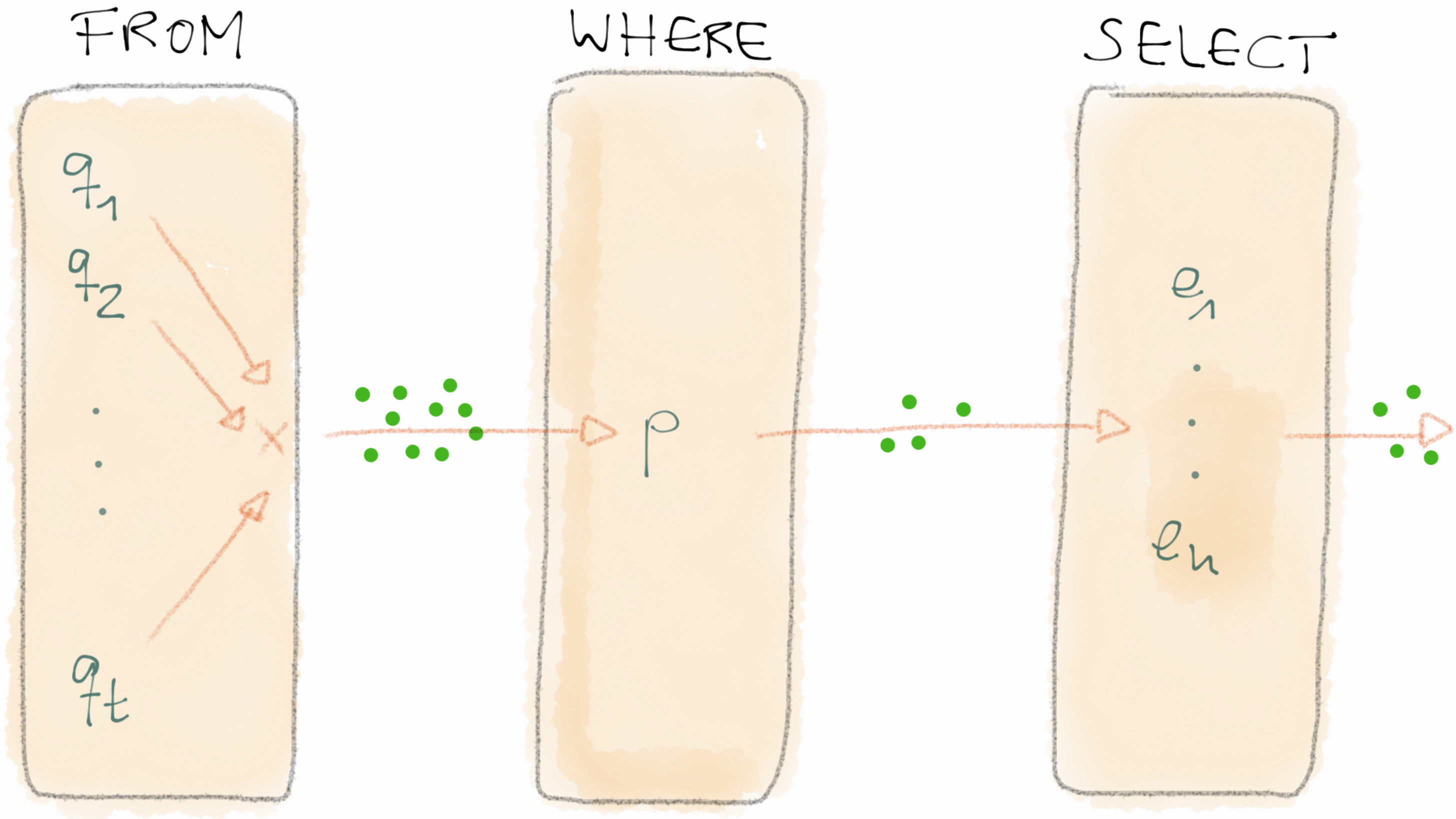
SQL: SELECT

```
SELECT  e₁ AS c₁, …, eₙ AS cₙ
FROM    (q₁) AS v₁, (q₂) AS v₂, …, (qₜ) AS vₜ
WHERE   p
```

- SQL refers to the v_i as **table aliases** but the PyQL equivalent makes it clear that **row variables** would be a better name: v_i is bound to each row of table q_i in turn (in *some* order).
- **Data flow** in a SQL SELECT query:

1. FROM clause: generate **all possible combinations** of row variable bindings. (Note that the order of the q_i under FROM is immaterial since SELECT returns an unordered table of rows anyway—in other words: the ‘,’ under FROM is commutative.)
2. WHERE clause: discard all row variable binding combinations that cannot satisfy predicate p.
3. SELECT clause: under all bindings that pass, evaluate the expressions e_i to construct a result table row of n columns named c_1, …, c_n.

SQL: SELECT



Data flow through a SQL SELECT-FROM-WHERE block

SQL: SELECT

- SQL implements a variety of syntactical sugar and abbreviations to aid the concise formulation of common query cases.
  - FROM clause: TABLE subqueries may be abbreviated:

```
SELECT   …                          ≡      SELECT …
FROM     …, (TABLE t) AS v, …              FROM   …, t AS v, …
```

  - WHERE clause: absence is interpreted as WHERE true.
  - SELECT clause: if row variable v iterates over the rows of $R(a_1,…,a_n)$, then v.* is interpreted as v.a_1 AS a_1, …, v.a_n AS a_n.
- Thus:

```
TABLE t        ≡       SELECT   v.*
                       FROM     t AS v
```

SQL: ROW TYPES AND ROW VALUES

- If row variable v iterates over the rows of table $R(a_1,\ldots,a_n)$ with $type(a_i) = t_i$, v itself has the **row type**

$$(a_1\ t_1,\ \ldots,\ a_n\ t_n)$$

- This row type is implicitly added to the set of all types $\mathbb{T}$ when the CREATE TABLE command for R is executed. CREATE TYPE can create such **composite types**, too:

```
CREATE TYPE R (a₁ t₁, …, aₙ tₙ)
```

- The **row values** of row type R can also be constructed via

```
ROW(e₁, …, eₙ) :: R
```

provided that the expressions are correctly typed, i.e., if e_i has type t_i. Keyword ROW is optional as long as $n > 1$.

SQL: JOINS

- In WHERE predicate p as well as in the SELECT expressions $e_1, \ldots, e_n$, all row variables introduced in the FROM clause are **in scope:**

```
SELECT  e₁(v₁, …, vₜ) AS c₁, …, eₙ(v₁, …, vₜ) AS cₙ
FROM    (q₁) AS v₁, (q₂) AS v₂, …, (qₜ) AS vₜ
WHERE   p(v₁, …, vₜ)
```

  - (Note: v_i is *not* in scope in $q_{i+1}, q_{i+2}, \ldots$ But see the SQL keyword LATERAL.)
- This permits the formulation of WHERE **predicates p that refer to multiple row variables and thus span tables**, the so-called **join predicates.**
- Join predicates may be used to reduce the arbitrary combinations of rows produced by the FROM clause. In particular, we may **bring related rows of separate tables together.**
- SELECT-FROM-WHERE blocks featuring such join predicates are also simply referred to as **joins.**

SQL: JOINS

Example: Who is busy at what times?

calendar

appointment	start	stop
meeting	11:30	12:00
lunch	12:00	13:00
biking	18:30	□
a_1	…	…

attendees

appointment	person
meeting	Alex
meeting	Bert
meeting	Cora
lunch	Bert
lunch	Drew
a_2	…

- Data from **both tables needed** to compute the result of the query. Two rows relate to each other if the **join condition** $a_1 = a_2$ is satisfied.

SQL: JOINS

- **Example:** Who is busy at what times?
- Result of the **equi-join:**

calendar			*attendees*	
appointment	**start**	**stop**	**appointment**	**person**
meeting	11:30:00	12:00:00	meeting	Alex
meeting	11:30:00	12:00:00	meeting	Bert
meeting	11:30:00	12:00:00	meeting	Cora
lunch	12:00:00	13:00:00	lunch	Bert
lunch	12:00:00	13:00:00	lunch	Drew
a	…	…	*a*	…

- In a join of tables R and S, a row of R may find between $0, \ldots, |S|$ **join partner** rows (e.g., the biking row in calendar found no join partner in attendees and thus does not contribute to the join result).
- In general: size of join result between $0, \ldots, |R| \times |S|$ rows.
  Note: omitted (forgotten) join conditions lead to join results that are too large.

SQL: JOINS

- Equality conditions are the common case in relational joins but the join conditions can be arbitrary, leading to the so-called **θ-joins** ("*theta-joins*").

calendar

appointment	start	stop
meeting	11:30	12:00
lunch	12:00	13:00
biking	18:30	□
…	s_1	e_1

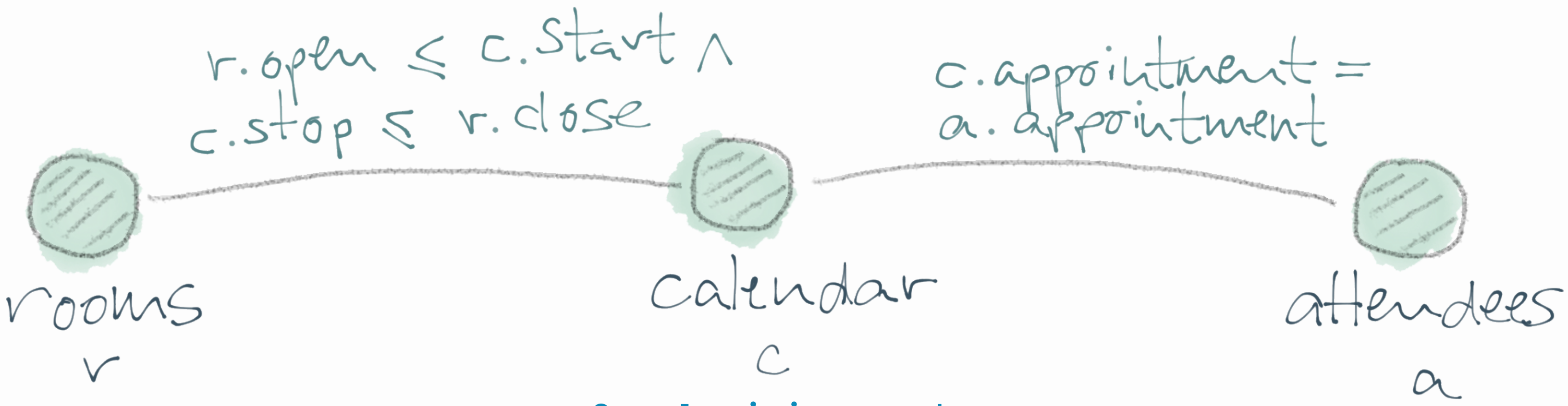
rooms

room	open	close
cafeteria	12:00	13:30
lobby	08:00	12:00
lobby	14:00	18:00
lecture hall	08:00	18:00
…	s_2	e_2

- **Example:** Which rooms are available for the scheduled appointments? Join condition: $[s_1,e_1] \subseteq [s_2,e_2]$ (here: θ ≡ interval inclusion).

SQL: JOIN

- In complex join queries, drawing the **join graph** may help to ensure that join conditions are not omitted and placed correctly between tables:
  - **Nodes** in join graph: tables participating in the query.
  - **Edge** $t_1 \overset{p}{\text{---}}\, t_2$: tables t_1, t_2 are joined under condition p.



Sample join graph.

SQL: EXPLICIT DUPLICATE REMOVAL

- Instances of SQL tables as well as SQL query results are **unordered lists** of rows and thus may contain duplicates.

calendar

no	appointment	start	stop
1	team meeting	2013-11-12 09:30	2013-11-12 10:30
2	lunch	2013-11-12 12:00	2013-11-12 13:00
3	team meeting	2013-11-12 14:00	2013-11-12 14:15
4	presentation	2013-11-12 18:00	2013-11-12 19:00

- The DISTINCT modifier to SELECT explicitly requests the **removal of duplicate rows** from a query result:

```
-- yields 4 rows                          -- yields 3 rows
SELECT c.appointment        vs.           SELECT DISTINCT c.appointment
FROM   calendar AS c                      FROM   calendar AS c
```

- Modifier ALL may be used to document that a query returns wanted duplicate rows.

SQL: EXPLICIT DUPLICATE REMOVAL

- Duplicate removal comes with potentially significant cost for the RDBMS if the result table is large.
- In some query scenarios, DISTINCT is superfluous and will perform wasted work:

1. If a declared **table has a primary key**, it cannot contain duplicate rows (cf. Codd's tuple set idea).
2. If the **columns of a SELECT clause form a (super-)key** for the query result, no duplicate rows can be returned.
3. If the WHERE clause contains **conjunctive equality conditions** $a_i = c_i$ (c_i literal) and the columns a_i together form a (super-)key of the query result, only one row will be returned.
4. [**Combination of 2. and 3.**] If the SELECTed columns and (constant) filtered columns together form a (super-)key of the result, no duplicate rows can be returned.

- Unfortunately, even such basic (incomplete, even) result **key inference** is not implemented in most RDBMSs.

SQL: COMPOSITIONALITY

> The **Principle of Compositionality** is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them.
>
> (Gottlob Frege)

- **Fully compositional (query) languages** admit the use of an expression—provided it has type *t*—wherever a value of type *t* is expected.
- Admits the assembly of complex queries from simpler constituent queries (or: subqueries) that can be tested separately.
- SQL has become more and more compositional in the course of its development (in particular with the SQL-92 standard) but has not reached full compositionality yet.

SQL: COMPOSITIONALITY

$$\begin{array}{ll}
\texttt{SELECT} & e_1(v_1, \ldots, v_t) \texttt{ AS } c_1, \ldots, e_n(v_1, \ldots, v_t) \texttt{ AS } c_n \\
\texttt{FROM} & (q_1) \texttt{ AS } v_1, (q_2) \texttt{ AS } v_2, \ldots, (q_t) \texttt{ AS } v_t \\
\texttt{WHERE} & p(v_1, \ldots, v_t)
\end{array}$$

- SQL admits **query nesting in the FROM, WHERE, and SELECT clauses:**
  - The q under the FROM clause are subqueries that yield tables. ✓
  - Predicate expression p may contain subqueries provided the overall expression yields a Boolean value.
  - Column expression e_i may contain subqueries provided the overall expression yields an atomic value.
- In an expression, SQL regards the **atomic value x and a subquery (q) (parentheses!) that yields the following single-row, single-column table** as equivalent:

…
x

SQL: COMPOSITIONALITY

- SQL further supports a modular, step-by-step formulation of complex queries through **common table expressions** (think “let … in for SQL”):

WITH (Common Table Expression)

A **common table expression** (CTE) binds the result of a *query* (i.e., a table) to a user-specified *query_name*. Subsequent bindings and the primary query in the same WITH statement can refer to this name like any other table:

```
WITH
 query_name [ ( column_name [, …] ) ] AS ( query ) [, …] -- bindings
 query                      -- primary query evaluated under all bindings
```

The *column_name* list is optional and can be inferred from the queries itself.

SQL: CORRELATION

- Recall row variable scoping: the row variables v_i are accessible ("*may occur free*") in predicate p as well as in the column expressions:

```
SELECT  e₁(v₁, …, vₜ) AS c₁, …, eₙ(v₁, …, vₜ) AS cₙ
FROM    (q₁) AS v₁, (q₂) AS v₂, …, (qₜ) AS vₜ
WHERE   p(v₁, …, vₜ)
```

  - This also applies to subqueries embedded in p or the e: **subqueries may refer to row variables that have been bound in the enclosing SELECT-FROM-WHERE block.**
  - These particular subqueries are referred to as **correlated subqueries.**
- Note that subquery correlation coincides with the variable scoping rules of most programming languages:
  - ⚠️ The enclosing (or: outer) query block *cannot* refer to the row variables bound in inner blocks.

SQL: CORRELATION

- **Example:** Use a correlated subquery in the SELECT clause to translate user ratings held in table users(name, rating) into ***⋯ markers:

```
SELECT  u.name, (SELECT s.stars
                 FROM   (VALUES (1, '*'    ),
                                (2, '**'   ),
                                (3, '***'  ),
                                (4, '****' ),
                                (5, '*****')) AS s(rating, stars)
                 WHERE  s.rating = u.rating) AS stars
FROM    users AS u;                ⚑
```

(Note how the query uses the row variables and column aliases s(rating, stars) to name the rows and columns of the literal table.)

- The flagged (⚑) occurrence of u.rating makes this a correlated query.
- Given the semantics of SELECT-FROM-WHERE blocks, the inner subquery will be evaluated repeatedly for different bindings of outer row variable u.

[End of SQL Diversion.]