Tabular Database Systems

⑦

More SQL (Subqueries + Embedded SQL)

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 ¦ Compositionality (in Programming Languages)

“The meaning of a complex program is determined by the meanings of its constituent programs.”

—Principle of **Compositionality**

The following two (Python) programs are equivalent:

1. Uses literal 21:

```
print(2 * 21)
```

2. Computes value 21:[1]

```
def twenty_one():
  return (12 + 12*12 + 20) // (2*4) - 1
print(2 * twenty_one())
```

[1] Based on a variation of a popular limerick: “*A dozen, a gross, and a score / divided by two times the four / decreased just by one / it gives twenty-one / (which is three times seven, no more).*”

2 ┆ SQL: Subqueries (Queries Inside Queries)

SQL queries may contain nested **subqueries** enclosed in (⋯).

1. **Scalar subquery:** Where a query accepts a scalar x, x may be replaced by a subquery (q) that returns a single-cell table:

   #036

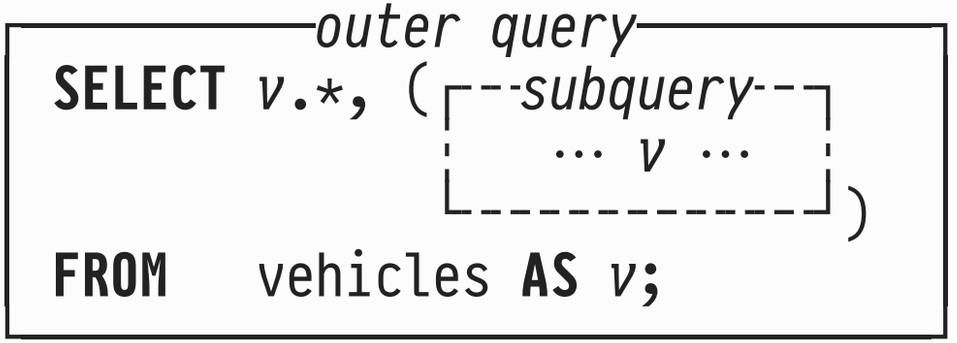
   q evaluates to $\boxed{\begin{array}{|c|}\hline\\\hline x\\\hline\end{array}} \equiv x$

   NB. A scalar subquery q needs to return…

   - … a **single-column** table (column name irrelevant). Otherwise: error at query *compile time*. 👎
   - … **at most one row.**
     0 rows ≡ NULL, ⩾ 2 rows: error at query *run time*. 👎 👎

Subqueries Can Relate to Outer Queries: Correlation

- Row variables bound inside a subquery do not “escape”: their scope is local to the subquery (see var v1 in #036).
- **But:** Subqueries may relate to row variables bound in the enclosing/outer query:



The subquery can access the current row bound to *v* ✓

  - The subquery in ⬚ *cannot* be evaluated in isolation: depends on outer query ☐ to provide a binding for row variable *v*.
  - DB jargon: “The subquery is **correlated** (since it uses *v*).”[2]

[2] PL jargon: “Variable *v occurs free* in the subquery (but bound in the outer query).”

Correlated Subqueries ≡ Nested Loops?

NB. Due to correlation, the subquery *q* in ⬚ below acts like a function of type int → text (*q*(pid) maps vehicle to driver name):

```
-- Query Q: Pair vehicles with their driver (if any)
                  ┌--------- q ----------┐
SELECT v.*, (     ¦ SELECT p.name        ¦
                  ¦ FROM   peeps AS p    ¦
                  ¦ WHERE  p.pid = v.pid ¦ ) AS driver
                  └----------------------┘
FROM   vehicles AS v;
```

- A "*nested loops*" evaluation strategy for *Q*:

```
for each v ∈ vehicles                                  ⎫
| driver ← q(v.pid)                                    ⎪  evaluated
⌊ result ← result ⊎ (v.*, driver)                      ⎭  |vehicles| times
```

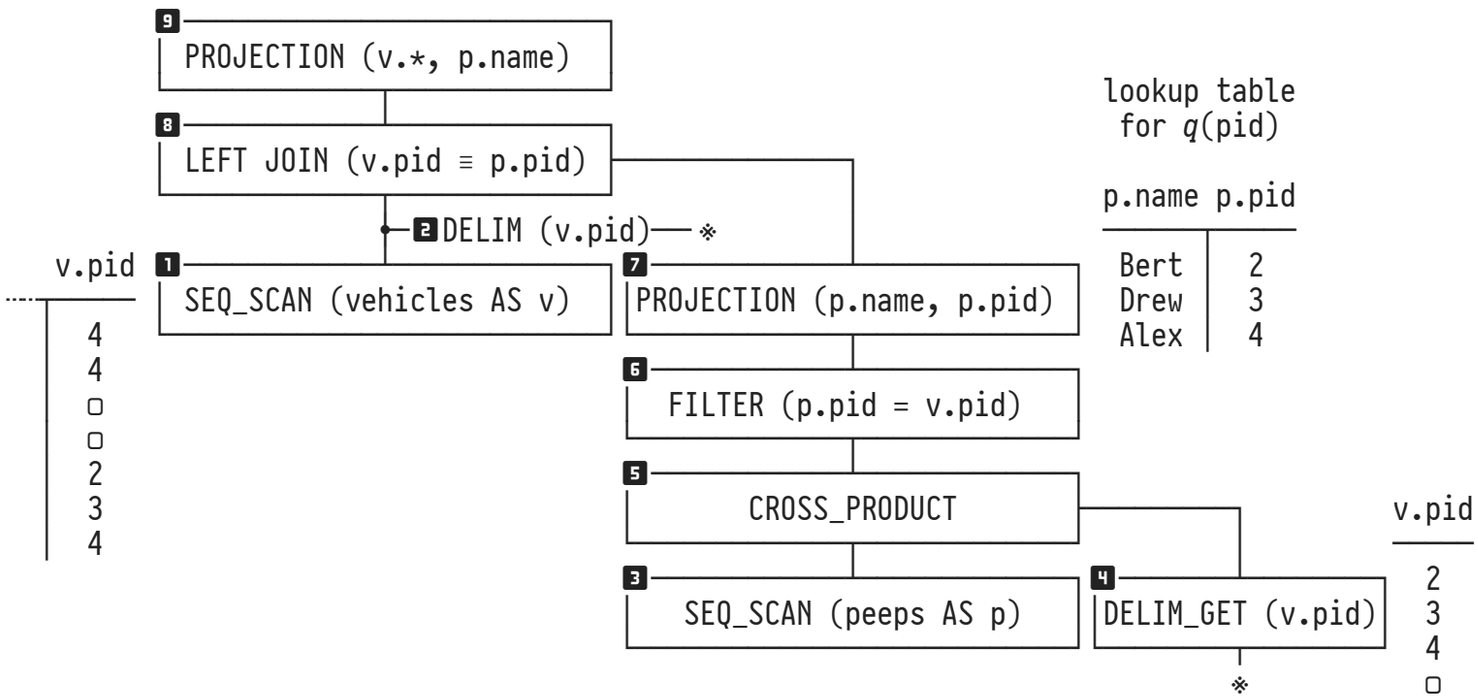
  - If vehicles.pid is not unique, this will evaluate subquery *q*(pid) repeatedly with identical arguments. 👎

DuckDB's Query Optimizer Removes Correlation

Query optimization **decorrelates** the subquery (plan for *Q*):[3]

1. 2,4: Compute the set of *distinct arguments* for subquery *q*.
2. 3–7: Build a lookup table for "function" *q*(pid).
3. 8: For each vehicle v, perform a lookup to evaluate *q*(v.pid).

#038

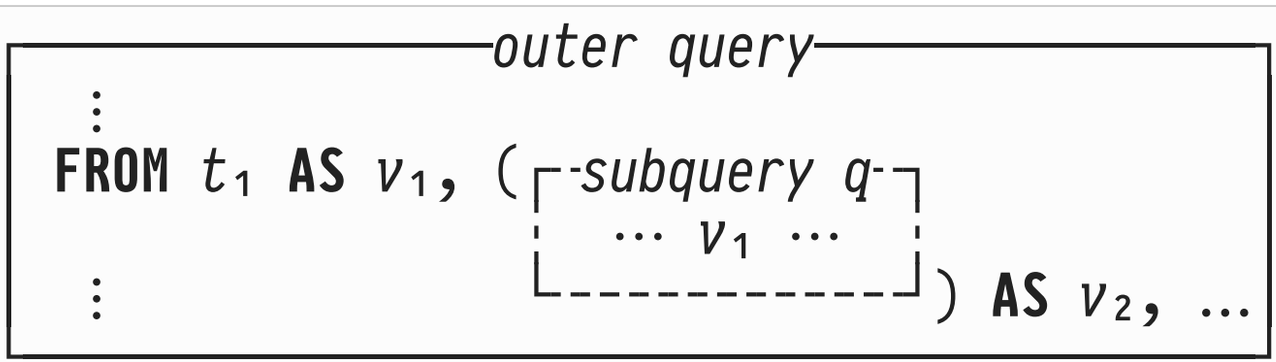


[3] Plan has been simplified. Use `PRAGMA explain_output = 'all';` to see the above Unoptimized Logical Plan. In these plans, the LEFT (OUTER) JOIN/DELIM pair is represented as DELIM_JOIN.

Scalar vs. Table-Valued Subqueries

SQL interprets subqueries (q) based on their *usage context*:

1. **Scalar:** q returns single-cell table ▢ that holds one scalar.
2. **Table-valued** (e.g., in **FROM** clause): q returns any table ▦:



#039

Subquery q in table-valued context

- Correlation: row variable v_1 may occur free[4] in subquery q. In this case, ⬚ acts like a table-valued function $q(v_1)$. (DBMS will decorrelate to avoid $|t_1|$ evaluations of q).
- If q is uncorrelated: ⬚ acts like a (computed) table.

[4] Some SQL implementations require the keyword **LATERAL** (“sideways”) to allow q to refer to v_1 (and thus depend on the evaluation of t_1): **FROM** t_1 AS v_1, **LATERAL** (q) AS v_2, …. (DuckDB infers whether **LATERAL** is required.)

3 ¦ SQL: Existential and Universal Quantification

SQL uses table-valued subqueries (*q*) to compactly formulate **existentially or universally quantified comparisons:**[5]

```
    EXISTS (q)  does q return ⩾ 1 rows  (is q non-empty)?
NOT EXISTS (q)  does q return no row at all (is q empty)?
```

```
expr = ANY (q)  does expr equal any value in ▨?
expr = ALL (q)  does expr equal all values in ▨?
```

also: <> < <= >= >

q evaluates to a single-column table:

[5] The SQL keywords **ANY** and **SOME** are synonyms. Syntactic sugar: *expr* = **ANY** (*q*) is equivalent to *expr* **IN** (*q*) (think of ∈ or "*is element* ***in***").

#040 + #041

4 Embedding SQL in Python Programs

- The DuckDB CLI enables interactive experimentation and the execution of ad-hoc/one-short querying. Definitely valuable.

- **Database-supported applications** embed SQL statements directly in the program source instead:
  - Programs can connect to/disconnect from selected databases.
  - Program flow controls which/how often SQL queries execute.
    - Queries may be constructed/parameterized on the fly.
  - Query results may be consumed by the program:
    - Map SQL data types to programming language's type system.
    - Receive *all* rows at once? *Iterate* over result row-by-row?

  Q: Which parts of the app logic are performed by the DBMS?
  Which parts are implemented by program code? ¯_(ツ)_/¯

Here ↔ **:** Use DuckDB's API to **embed SQL queries into Python.**

Embdedding SQL in Python: General Setup

Application code mixes SQL query strings 🛢 with program code 🐍:

#042

```
import duckdb   # requires: pip install duckdb
 ⋮
 # 1 connect to database (in 🔲 or on 💽)
 with duckdb.connect(database) as con:
    # 2 construct SQL query, submit to DuckDB
    rel = con.sql("""
      ┌-----------🛢   ┐  • embed SQL query as literal multi-line
      ¦  SQL query  ¦   }    string between """..."""
      └-------------┘   ┘  • sent to DuckDB (not executed yet)
      """)
    # 3 DuckDB executes query, retrieve all result rows
    result = rel.fetchall()
    # 4 iterate over list of rows
    for row in result:
      ┌-----------🐍   ┐  • Python code operates on a result row
      ¦    code     ¦   }    represented as a tuple (...,...,...)
      └-------------┘   ┘  • DuckDB not involved
```

DuckDB's Python DB API (Overview[6])

DuckDB Python API Call	Python Result
❶ `con = duckdb.connect(":memory:")`	DuckDB connection object
`con = duckdb.connect(`*`database file`*`)`	
❷ `rel = con.sql(`*`sql`*`)`	effect on DB or DuckDB relation object[7]
`rel = con.sql(`*`sql`*`, params = [...])`	*sql* may contain parameters `$1`, `$2`, ...
❸ `rel.fetchall()`	list of all result tuples in table `rel`
`rel.fetchmany(`*`n`*`)`	list of next *n* result tuples
`rel.fetchone()`	next result tuple or `None`
`rel.columns`	list of column names for table `rel`
`rel.types`	list of column types
`rel.show()`	`None` + printed table output (⊞)

[6] DuckDB's Python DB API (documentation)

[7] If *sql* is a **SELECT** query, returns a DuckDB relation object `rel`. Otherwise, applies the effect of the DDL or DML statement to the database represented by DuckDB connection object `con`.

How SQL Data Maps to Python Values/Objects

The type systems of SQL and Python—indeed most PLs—differ.[8] SQL data (tables, rows, cell values) needs to be **mapped** to Python values and objects:

a	b	c
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

SQL	Python
table	≡ list
row	≡ tuple
cell a_i	≡ val/obj A_1

```
[
  (A₁, B₁, C₁),
  (A₂, B₂, C₂),
  (A₃, B₃, C₃)
]
```

Data and Type Mapping in DuckDB's Python DB API

- DuckDB implements a best-effort mapping from cell values a_i (of types int, text, ...) to Python's value and objects A_i. #043

[8] This tension—not only regarding types—between SQL as a query language and programming languages (PLs) is known as the **impedance mismatch**.

Constructing SQL Queries at Program Run Time

Embedded SQL queries are regular strings: **programs can construct queries at run time** by interpolation 👍 or concatenation 👎.

1. **Interpolation** (Python values replace parameterized SQL values $1, $2, … in a template query):

   "**SELECT** $1 || p.name **FROM** peeps **AS** p **AS WHERE** p.born < $2"

   🐍 [1:"Driver: ", 2:2000]

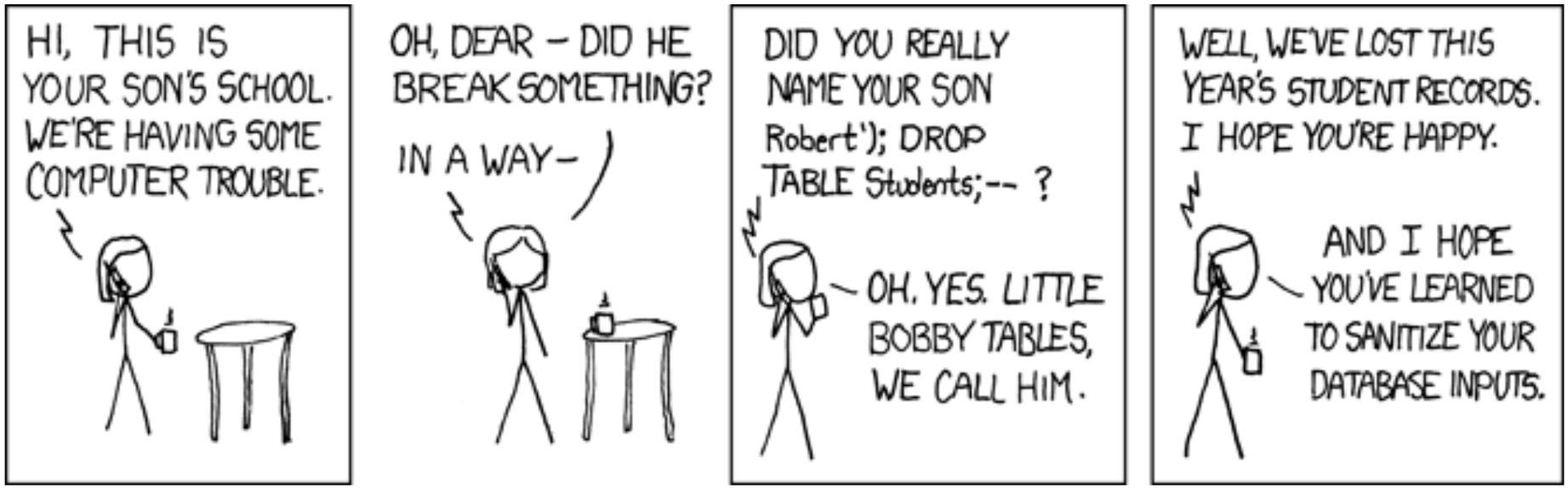
2. String concatenation (⚠️ Risk of SQL injection):

   "**FROM** peeps **AS** p **WHERE** p.name = '" + *N* + "' **AND** p.born < 2000"

   🐍 string concatentation

   If an attacker controls the value of Python string variable *N*, the DBMS may be tricked into executing arbitrary queries.

Parameterized Queries Protect Against Little Bobby's Mom



"Exploits of a Mom" , © xkcd

#044 + #045

Move Your Computation Close to the Data!

A rule of thumb 👍: *If you can, express data-related computation using SQL.*[9] *Do not demote the DBMS to a dumb table storage.*

- Filter/aggregate tables to **reduce result set sizes** before you transport data from the DBMS to the program.
- Common anti-pattern (the "***n+1 query problem***"):

```
outer = con.sql("SELECT ...")     # yields n rows
for row in outer.fetchall():
    inner = con.sql("SELECT ... $1 ...", params = [... row ...])
    ⋮
```

#046

  - The above issues *n+1 separate queries* all of which need to be interpolated, parsed, optimized, executed, and fetched. 👎
  - Reformulate using a join or a correlated subquery (which the query optimizer will decorrelate). Issues a *single query*. 👍

[9] Indeed, with SQL:1999 and the introduction of *recursive common table expressions*, the query language has become Turing-complete. We explore the consequences of this jump in expressivness in the course *Advanced SQL*.