Advanced SQL

07 — Procedural SQL

Summer 2022

Torsten Grust
Universität Tübingen, Germany

1 ┆ Scripting Language + SQL = Procedural SQL

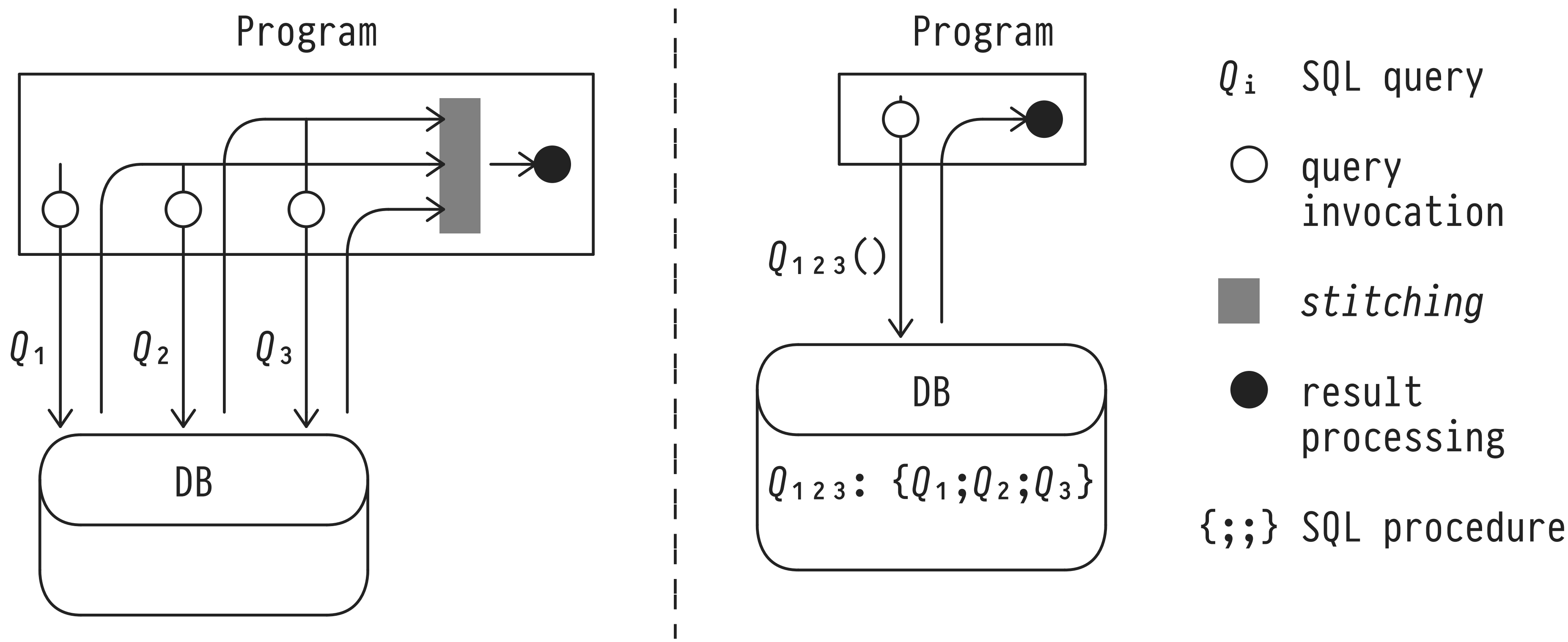
We started out in this course with the aim to **move more computation close to the data.** Admitting recursion in SQL is one way to declaratively express complex computation.

Procedural SQL follows an entirely different path towards this goal:

- Implement application logic *inside* the RDBMS, even if this **computation is inherently procedural** (≡ sequential, imperative).

- Use **SQL as a sub-language of a scripting language** whose types match those of the tabular data model.

Procedural SQL: Less Round-Trips, Less Stitching



- *Stitching*: On the PL heap, piece together the tabular results delivered by the individual SQL queries Q_i.

Procedural SQL aka *Stored Procedures*

Code in Procedural SQL is organized in **functions/procedures that are stored persistently** by the DBMS.[1]

These functions/procedures…

- may be used anywhere that SQL's built-ins could be used,
- inherit all user-defined types, functions, and operators,
- can define new operators, aggregate/window functions, and triggers.

[1] This implies that we need to manage these procedures using familiar constructs like `CREATE PROCEDURE …`, `CREATE FUNCTION …`, `DROP PROCEDURE [IF EXISTS] …`, etc.

PL/SQL:[2] Scripting with SQL Types

```
CREATE FUNCTION f(x_1 τ_1, …, x_n τ_n) RETURNS τ AS
$$ … block … $$
LANGUAGE PLPGSQL;
```

- The τ_i, τ may be any scalar, array, or (named) row type.
- Limited polymorphism: functions may accept/return types anyelement, anyarray (recall our discussion of SQL UDFs).
  - Functions may return type record (then the caller must provide column names/types through explicit aliasing).
- Functions may return—but *not* accept ☹—sets of (row) values with $\tau \equiv$ SETOF $\bar{\tau}$.

[2] *PL/SQL* is the widely adopted abbreviation for *Procedural Language for SQL*, originating in the Oracle® RDBMS. Variants include *Transact-SQL* (Microsoft® SQL Server) and *PL/pgSQL* (PostgreSQL).

2 ¦ Block Structure

PL/SQL code is organized in (nested) **blocks** that group statements and define **variable scopes:**

```
block { [ DECLARE declarations ]
        BEGIN
          statement              -- ← any statement may be
        END;                     --    a (sub-)block again
```

- Declared variables are in scope in the block and its sub-blocks. Local names shadow outer names.
- Optionally introduce block with << *label* >>: variable *v* may then also be referred to as *label.v*.
- Outermost block of body for *f* has implicit << *f* >>.

Block Structure and Variable Scope[3]

in scope	**CREATE FUNCTION** $f(x_1\ \tau_1)$ **RETURNS** τ **AS**
	`$$`
$f.x_1$	`<<` o `>>` -- outer block
	DECLARE $v\ \tau_v$;
$f.x_1,\ o.v$	**BEGIN**
	⋮
	`<<` i `>>` -- inner (sub-)block
	DECLARE $v\ \tau_u$;
$f.x_1,\ o.v,\ i.v$	**BEGIN**
	⋮
	END;
	⋮
	END;
	`$$` **LANGUAGE** PLPGSQL;

[3] Additional special variables (like FOUND) are bound in the outermost f scope (see below).

3 ┆ Variable Declarations

The optional DECLARE *declarations* brings **typed variable(s)** v into scope. An initial binding expression e may be given:

```
DECLARE v [ CONSTANT ] τ [ NOT NULL ] [ := e ];
        ⋮
```

- If := e is omitted, v has initial value NULL.
- NOT NULL: any assignment of NULL yields a runtime error.
- CONSTANT: the initial binding may not be overwritten.

- Use c%TYPE for τ to declare v with the same type as variable or table column named c ("*the type of c*").

Variables With Row Types Have Row Values

Let T be a table with **row type** $(c_1\ \tau_1,\ \ldots,\ c_n\ \tau_n)$. Recall: this row type is also known as T. Thus:

```
--                              ↓ row type name
CREATE FUNCTION accessᵢ(t T) RETURNS T.cᵢ%TYPE AS
$$
--          ↓+↓ table + column name
DECLARE x T.cᵢ%TYPE;   -- x has type τᵢ
BEGIN
  x := t.cᵢ;           -- field access uses dot notation
  RETURN x;
END;
$$
LANGUAGE PLPGSQL;
```

4 ┆ PL/SQL Expressions

In PL/SQL, any expression *e* that could also occur in a `SELECT` clause, is a valid expression.

In fact, the execution of PL/pgSQL statements like

```
v := e
IF e THEN … ELSE … END IF
```

lead to the evaluation of `SELECT` *e* by the SQL interpreter.

- Interoperability between PL/pgSQL and SQL. 👍
- Performance impact: context switches PL/SQL↔SQL. 👎
- If $e \equiv e(x,y)$, compile SQL once with parameters x,y.

5 ┆ PL/SQL Statements — Assignment

```
v := e
```

1. Evaluate *e*, yields a single value (scalar, row, array, user-defined, including NULL). *e* may **not be table-valued.**

2. Cast value to type *τ* of *v*.
   - SQL casting rules apply (may fail at runtime).
   - *e* may use textual literal syntax (e.g., for user-defined enumerations, JSON, or geometric objects).

3. Bind variable *v* to value.

Assignment of Single-Row Query Results

A single-row[4] SQL query augmented with INTO is a valid PL/SQL assignment statement:

❶	❷
SELECT e_1, e_2, …, e_n INTO v FROM …	SELECT e_1, e_2, …, e_n INTO v_1, v_2, …, v_n FROM …

1. Evaluate SQL query, obtain a single row of n values.
   ❶ Assign row value to row-typed variable v, or
   ❷ assign value of e_i to v_i ($i \in \{1,\ldots,n\}$).

2. Variable FOUND :: boolean indicates if a row was found.

[4] Use INTO STRICT to enforce a single-row query result. Otherwise, the “first” row is picked… 🙈

Assignment of Scalar Query Results

RHS of assignment v := e is evaluated like a regular SQL query. In particular, e may be a *scalar subquery* in (…):

```
v := (Q)   --
           -- Q yields single row, single column: | c |
           --
```

- Evaluates SELECT (Q) behind the scenes[5]. Thus:
  - assigns cell value c cast to type τ of v, or
  - assigns NULL :: τ to v if Q returns no row, or
  - yields runtime error if Q returns more than one row or column (or if the cast fails). ⚠

[5] **NB.** Scalar assignment does not update variable FOUND (unlike the INTO construct, see above).

6 ┆ If All You Want Are the Side Effects…

1. Statement NULL does nothing (no side effects).

2. SQL **DML statements** (INSERT/DELETE/UPDATE) without RETURNING clauses are valid PL/SQL statements: no value is returned, the effect on the database is performed.

3. A SQL **query** SELECT *…query…* may be performed solely for its side effects (e.g., invocation of a side-effecting UDF) as well:

```
PERFORM …query… -- ⚠️ PERFORM replaces the SELECT keyword
```

Resulting rows are discarded (but variable FOUND is set).

7 ¦ Returning From a Non-Table Function (RETURNS τ)

```
RETURN e
```

1. Evaluate e, cast value to return type τ of the function.
    - If $\tau \equiv$ void, omit e. A void function whose control flow reaches the end of the top-level block, returns automatically.
2. Execution resumes in the calling function or query which receives the returned value.

To return multiple values, declare the function to return a row type.

"Returning" From a Table Function (RETURNS SETOF τ)

❶ **RETURN NEXT** *e*; *s*	❷ **RETURN QUERY** *Q*; *s*

- **Add (bag semantics: ⊎) to the result table** computed by the function. Execution resumes with following statement *s* — **no** return to the caller yet.
  ❶ Evaluate expression *e*, add scalar/row to result.
  ❷ Evaluate SQL query *Q*, append all rows to result.

- Use plain `RETURN;` to return the result table accumulated so far and resume execution in the caller.

8 ┆ Conditional Statements

$$\texttt{IF}\ p_0\ \texttt{THEN}\ s_0\ \underbrace{[\ \texttt{ELSIF}\ p_i\ \texttt{THEN}\ s_i\ ]^*}_{\text{optional, repeatable}}\ \underbrace{[\ \texttt{ELSE}\ s_e\ ]}_{\text{optional}}\ \texttt{END IF}$$

- Semantics as expected; p_i :: bool, s_i statements.

$$\texttt{CASE}\ e\ \underbrace{[\ \texttt{WHEN}\ e_{i1}\ [,\ e_{ij}]^*\ \texttt{THEN}\ s_i\ ]^+}_{\text{mandatory, repeatable}}\ [\ \texttt{ELSE}\ s_e\ ]\ \texttt{END CASE}$$

- Execute first branch s_i with $\exists_j: e = e_{ij}$.
- Raise CASE_NOT_FOUND exception (see below) if no branch was found and ELSE s_e is missing.

9 ¦ Iterated Statements

❶ **LOOP** $s^{\circlearrowright}$ **END LOOP**

❷ **WHILE** p **LOOP** $s^{\circlearrowright}$ **END LOOP**

❸ **FOR** v_i **IN** [**REVERSE**] e_0..e_1 [**BY** e_2] **LOOP** $s^{\circlearrowright}$ **END LOOP**

❹ **FOR** v_r **IN** q **LOOP** $s^{\circlearrowright}$ **END LOOP**

❺ **FOREACH** v_a **IN** [**SLICE** n] **ARRAY** e_a **LOOP** $s^{\circlearrowright}$ **END LOOP**

❶ Endless loop (see EXIT below).

❷ p :: bool.

❸ $e_{0,1,2}$ :: int. No BY: $e_2 \equiv 1$. v_i :: int (auto-declared) bound to e_0, $e_0 \pm 1 \times e_2$, $e_0 \pm 2 \times e_2$, … (REVERSE: $\pm \equiv -$).

❹ q SQL query. v_r successively bound to resulting rows.

❺ e_a :: τ[]. No SLICE: v_a :: τ bound to array elements. SLICE n: v_a :: τ[] bound to sub-arrays in nth dimension.

Leaving/Short-Cutting Loops

All five LOOP forms support optional << *label* >> prefixes:

<< *label* >> … **LOOP** *s*↺ **END LOOP**

We may alter the control flow inside a loop via:

❶ **EXIT** [*label*] [**WHEN** *p*]
❷ **CONTINUE** [*label*] [**WHEN** *p*]

- No *label*: refer to innermost enclosing loop.
- WHEN *p*: leave/shortcut loop only if *p* ≡ true.
- EXIT *label* may also be used to leave a statement block.

Leaving/Shortcutting Loops

```
<< outer >>
LOOP
  s0
  ⋮
  << inner >>                         shortcut
  LOOP
    ⋮
    CONTINUE outer;
    ⋮
    EXIT inner;
    ⋮                                 leave
  END LOOP;
  s1
  ⋮
END LOOP;
```

- Shortcutting a WHILE p loop leads to re-evaluation of p.

10 ┆ Trapping Exceptions in Blocks

```
BEGIN
  ⋮   -- ⎫ errors or RAISE ex statements transfer control
  sₓ  -- ⎬ to the EXCEPTION clause – if sₓ changed the
  ⋮   -- ⎭ database, also performs a rollback
EXCEPTION
  [ WHEN ex_i1 [, ex_ij]* THEN s_i ]+
END;
s_1     -- next statement if no exception occurred
```

- On error or RAISE, search for first matching exception category/name ex_{ij}, execute s_i, then s_1.
- If no match is found (or s_i fails), propagate exception to enclosing block. Abort function if in outermost block.

Raising Exceptions

```
                    one expression per '%' in message
                                      ⏞
❶  RAISE [ level ] '… % … % …' [, e]*
❷  RAISE [ level ] ex
❸  ASSERT p [, e]
```

- *level* ∈ {DEBUG,LOG,INFO,NOTICE,WARNING}. Only the default *level* ≡ EXCEPTION raises an exception of name RAISE_EXCEPTION (or *ex*[6], if provided).

- ASSERT *p* (*p* :: bool) raises exception ASSERT_FAILURE—with optional message *e* :: text—if *p* ≡ false.

[6] See https://www.postgresql.org/docs/current/errcodes-appendix.html for a catalog of exception categories/names.

11 ⁞ The Core of a Spreadsheet

╲	A	B	C	D
1	1	3.50	A1×B1	€→£ 0.88
2	2	6.10	A2×B2	
3	2	0.98	A3×B3	
4	#items SUM(A1:A3)		total (€) SUM(C1:C3)	total (£) D1×C4

Before evaluation

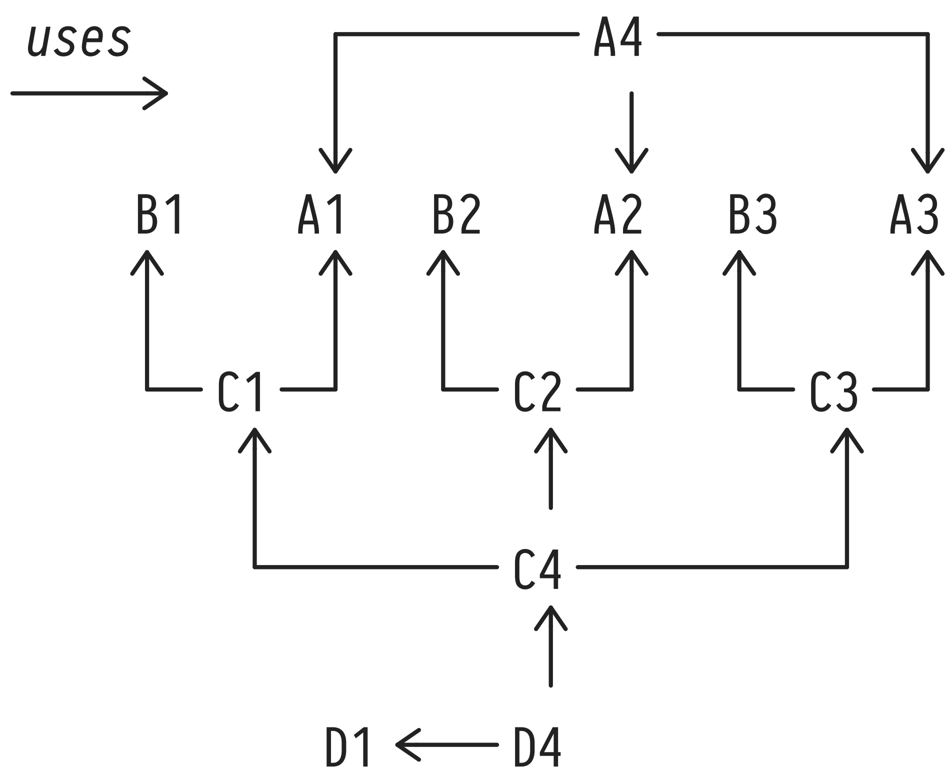
╲	A	B	C	D
1	1	3.50	3.50	€→£ 0.88
2	2	6.10	12.20	
3	2	0.98	1.96	
4	#items 5		total (€) 17.66	total (£) 15.54

After evaluation

- A1×B1: formulæ to be evaluated, <u>total (€)</u>: static text.

A DAG of Cell Dependencies

- Spreadsheet formulæ induce a directed **dependency graph:**



- Formulæ in A1–A3, B1–B3, D1 may be evaluated first (and in parallel).
- Formula in cell D4 needs to be evaluated last.
- **Topologically sort** the graph's cells to derive an **evaluation order.**

Topologically Sort a DAG dependencies(cell, uses)

```
WITH RECURSIVE
earliest(pos, cell) AS (
  SELECT DISTINCT 0 AS pos, d.uses AS cell
  FROM   dependencies AS d
  WHERE  d.uses NOT IN (SELECT d1.cell                    -- cells d with no
                        FROM   dependencies AS d1)        -- dependencies

    UNION

  SELECT e.pos + 1 AS pos, d.cell                  -- if cell d uses cell e, d
  FROM   earliest AS e, dependencies AS d          -- can be evaluated directly
  WHERE  d.uses = e.cell                           -- after e at the earliest
),
topo_sort(pos, cell) AS (
  SELECT MAX(e.pos) AS pos, e.cell               -- for each cell e, pick the
  FROM   earliest AS e                           -- evaluation time at which all
  GROUP BY e.cell                                -- dependencies are available
)
```

Dependencies, Topologically Sorted

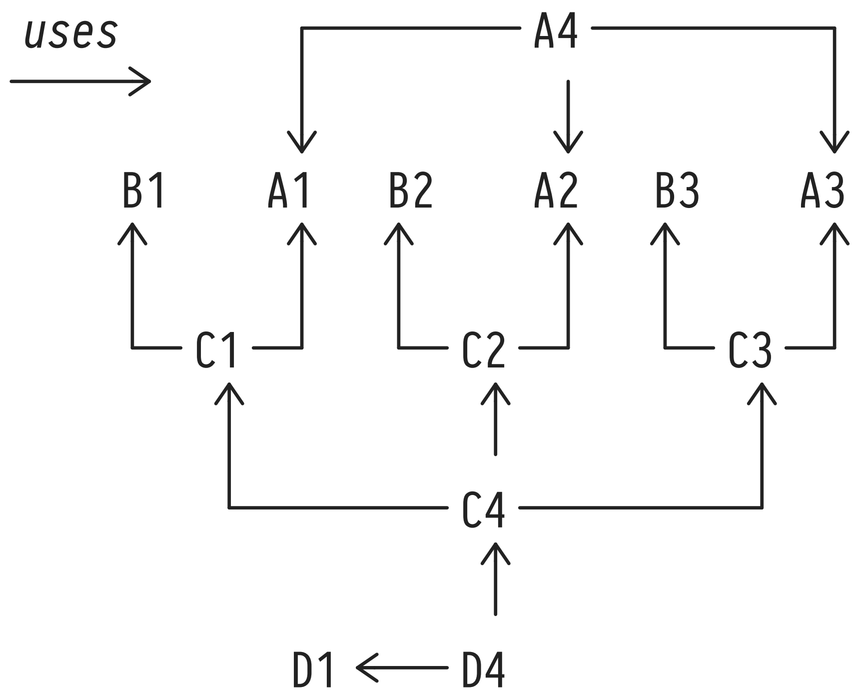


Table **topo_sort**

pos	cell
0	(A,1)
0	(A,2)
0	(A,3)
0	(B,1)
0	(B,2)
0	(B,3)
0	(D,1)
1	(A,4)
1	(C,1)
1	(C,2)
1	(C,3)
2	(C,4)
3	(D,4)

- Column **pos** describes parallel evaluation order.
- Use **DENSE_RANK**() to obtain a sequential order.

Formula Representation

- We need a representation of formulæ that supports
  1. the **extraction of references** to other cells and
  2. the **evaluation** of (arithmetic) expressions.

- One option: use **nested JSON objects** to reflect the hierarchical structure of formulæ:

❶ literal ¦ ❷ cell ref ¦ ❸ *n*-ary op ¦ ❹ agg over cell range

```
❶ {"entry":"num",  "num":4.2}
❷ {"entry":"cell", "cell":"(A,3)"}
❸ {"entry":"op",   "op":"+", args:[‹formula›,‹formula›]}
❹ {"entry":"agg",  "agg":"sum", "from":"(A,2)", "to":"(D,5)"}
```

formula kind — formula details (“payload”)

Extracing Cell References in a Formula (PL/SQL)[7]

```
CREATE FUNCTION refs(e jsonb) RETURNS SETOF cell AS
$$
BEGIN
  CASE e['entry']
    WHEN '"op"' THEN
        -- recursively collect references found in operator arguments
        RETURN QUERY SELECT c.*
                     FROM   jsonb_array_elements(e['args']) AS arg,
                            LATERAL refs(arg) AS c; -- ← recursive call
    WHEN 'agg'  THEN
        -- all cells in rectangular area are referenced (SQL UDF cells())
        RETURN QUERY SELECT c.*
                     FROM   cells(e->>'from', e->>'to') AS c;
    WHEN '"cell"' THEN RETURN NEXT e['cell'];   -- reference to single cell
    WHEN '"num"'  THEN NULL;                     -- NULL: do nothing (≡ NOP)
    ELSE RAISE EXCEPTION 'refs: unknown cell entry %', e['entry'];
  END CASE;
  RETURN;
END;
$$
```

[7] *e->>f* extracts field *f* from JSON value *e*, then casts to type text.

Evaluate a Formula (PL/SQL)

```
CREATE FUNCTION eval(e jsonb) RETURNS float AS
$$                             --
DECLARE v float;               --
BEGIN                          --
  CASE e['entry']              --
    WHEN '"op"' THEN           --
      CASE e['op']
        WHEN '"+"' THEN v := eval(e['args'][0]) + eval(e['args'][1]);
        ⋮
      END CASE;
    WHEN '"agg"' THEN v:= (SELECT CASE e['agg']
                                    WHEN '"sum"' THEN SUM(value(c))
                                    ⋮
                                  END
                           FROM   cells(e->>'from', e->>'to') AS c);
    WHEN '"cell"' THEN v := value(e->>'cell');
    WHEN '"num"'  THEN v := e->>'num';
  END CASE;
  RETURN v;
END;
$$
```

PL/SQL UDF **value**(*c*) may assume that cell *c* contains a float literal: if we refer to *c*, the topological sort ensures that *c* has already been evaluated

Spreadsheet Evaluation (Query Plan)

1. Store the cells in table `sheet(cell, formula :: jsonb)`.

2. Extract dependencies of each cell's formula (`refs()`), use to build topologically sorted array *cs* of cells.

3. PL/SQL UDF `eval_sheet()`:
   For each cell *c* in *cs*:
   ❶ Read formula *e* for *c* off table `sheet`.
   ❷ *v* := `eval(`*e*`)` to find float value *v* of formula *e*.
   ❸ **Update** cell *c* in `sheet` to `{"entry":"num", "num":`*v*`}`.

4. All cells in `sheet` will contain `{"entry":"num", …}`.

Spreadsheet Evaluation (PL/SQL)

```
  CREATE FUNCTION eval_sheet(cs cell[]) RETURNS boolean AS
  $$
  DECLARE c cell; v float; e jsonb;
  BEGIN
    FOREACH c IN ARRAY cs LOOP
      SELECT s.formula
1     INTO   e
      FROM   sheet AS s
      WHERE  s.cell = c;

2     v := eval(e);

      UPDATE sheet AS s
3     SET    formula = jsonb_build_object('entry', 'num', 'num', v)
      WHERE  s.cell = c;
    END LOOP;
  RETURN true;
  END;
  $$
```

PL/SQL—The Best Thing Since Sliced 🍞?

- PL/SQL…

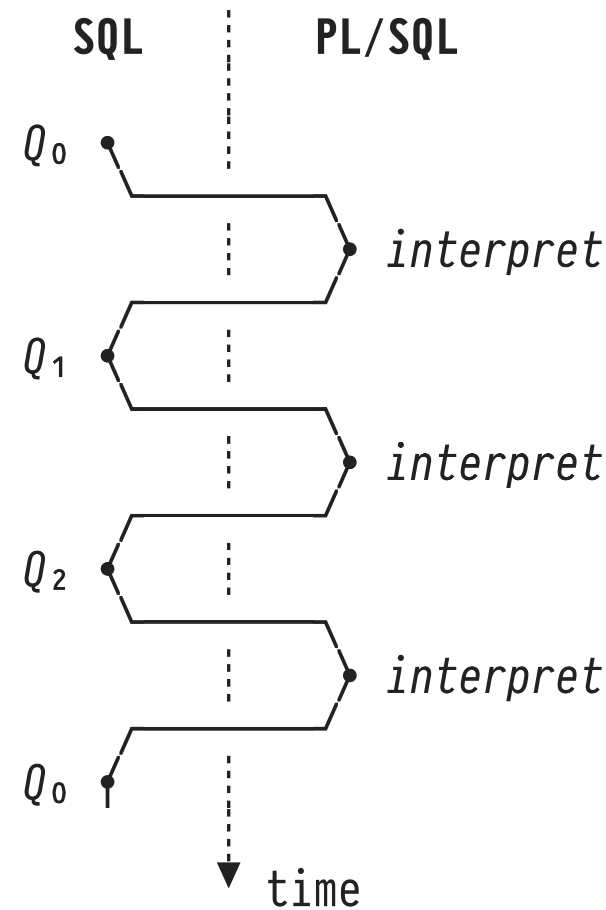
  1. … follows the established **imperative programming paradigm** in which so many developers are well-versed,
  2. … is **tightly coupled with the RDBMS**'s data model, type system, and built-in operators/functions:

  Why would we ever consider to express complex computation in terms of pure SQL (e.g., recursive CTEs) again?

- ⚠️ Many PL/SQL implementations incur a serious performance penalty due to **PL/SQL↔SQL context switches.**

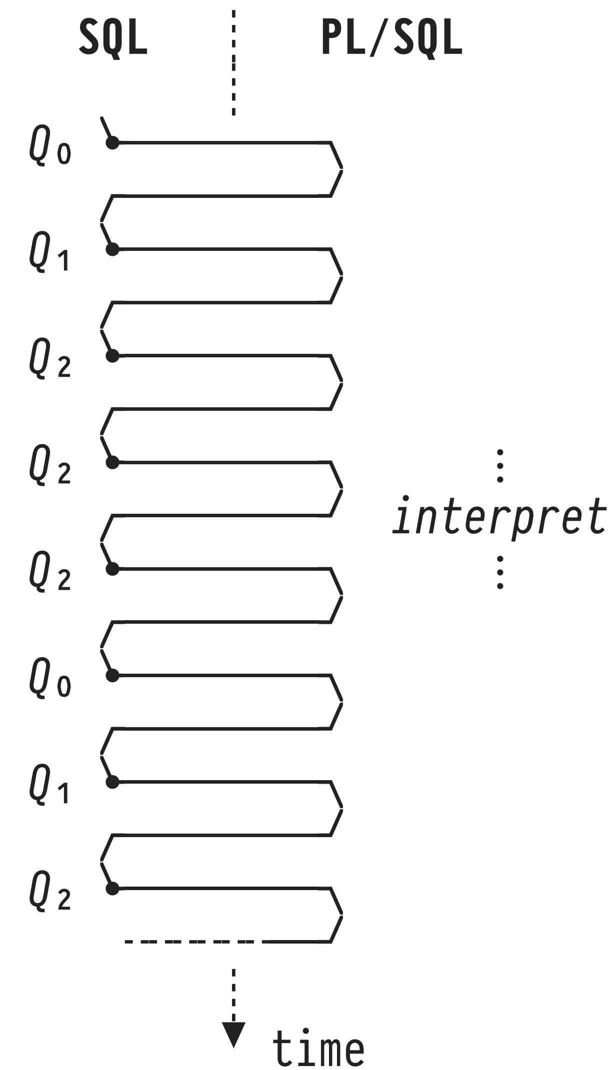
From SQL to PL/SQL And Back Again

```
CREATE FUNCTION f(…) RETURNS τ AS
$$
   v₁ := (
         );                    ] embedded
                               ] SQL Q₁
     v₂ := (
           );                  ] embedded
                               ] SQL Q₂
$$ LANGUAGE PLPGSQL;

SELECT f(…);                   ] top-level
                               ] SQL Q₀
```

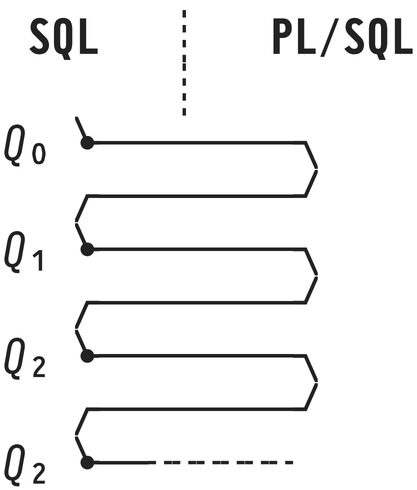


SQL↔PL/SQL (Iteration Inside and Outside the UDF)

```
CREATE FUNCTION f(⋯) RETURNS τ AS
$$
   ====
   v1 := (                               ]  embedded
                  );                     ]  SQL Q1
   ==========
   WHILE === LOOP
   |  ============
   |     ============
   |  ===
   |     =====
   |     v2 := (                         ]  embedded
   |                       );            ]  SQL Q2
   LOOP END
   ==========
$$ LANGUAGE PLPGSQL;

SELECT f(⋯,t,⋯);                         ]  top-level
FROM   T AS t                            ]  SQL Q0
```



SQL↔PL/SQL Context Switches Are Costly



plan + optimize + **instantiate** + run + **teardown** ··· Q_0

plan + optimize + **instantiate** + run + **teardown** ··· Q_1

plan + optimize + **instantiate** + run + **teardown** ··· Q_2

instantiate + run + **teardown** ··· Q_2

PostgreSQL Runtime Profile ⏰

That's All, Folks

Keep on querying and until next time.