

Advanced SQL

07 — Procedural SQL

Summer 2020

Torsten Grust
Universität Tübingen, Germany

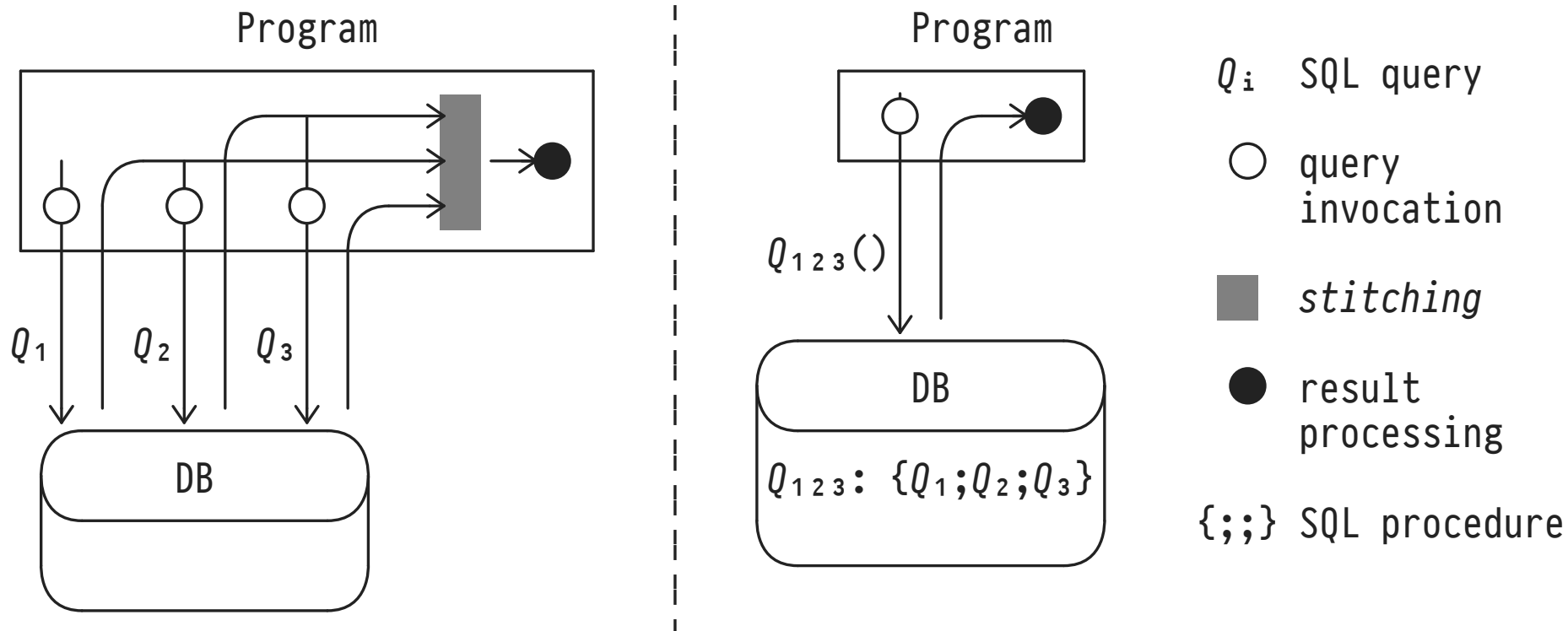
1 | Scripting Language + SQL = Procedural SQL

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  $f(x_1 \ \tau_1, \dots, x_n \ \tau_n)$  RETURNS  $\tau$  AS
 $\$ \$ \dots \langle block \rangle \dots \$ \$$ 
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 \rightsquigarrow —sets of (row) values with $\tau \equiv \text{SETOF } \bar{\tau}$.

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 Scope³

<i>in scope</i>		CREATE FUNCTION $f(x_1 \ \tau_1)$ RETURNS τ AS
		\$\$
$f.x_1$	[<< 0 >> -- outer block
$f.x_1, \ 0.v$		DECLARE $v \ \tau_v$; BEGIN
		\vdots
	[<< i >> -- inner (sub-)block
$f.x_1, \ 0.v, \ i.v$		DECLARE $v \ \tau_u$; BEGIN
		\vdots
		END;
		\vdots
		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 `v`.

Variables With Row Types Have Row Values

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

```
--                                ▾ row type name
CREATE FUNCTION accessi(t T) RETURNS T.ci%TYPE AS
$$
--                ▾+▾ table + column name
DECLARE x T.ci%TYPE; -- x has type  $\tau_i$ 
BEGIN
    x := t.ci;          -- field access uses dot notation
    RETURN x;
END;
$$
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-row⁴ SQL query augmented with **INTO** is a valid PL/SQL assignment statement:

```
1 SELECT  $e_1, e_2, \dots, e_n$ 
      INTO  $v$ 
      FROM ...
```

```
2 SELECT  $e_1, e_2, \dots, e_n$ 
      INTO  $v_1, v_2, \dots, v_n$ 
      FROM ...
```

1. Evaluate SQL query, obtain a single row of n values.
 - 1** Assign row value to row-typed variable v , or
 - 2** assign value of e_i to v_i ($i \in \{1, \dots, n\}$).
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 scenes⁵. 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 $\tau \equiv \text{void}$, 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**)

1 RETURN NEXT e ;
 s

2 RETURN QUERY Q ;
 s

- Add (bag semantics: \cup) to the result table computed by the function. Execution resumes with following statement s — no return to the caller yet.
 - 1** Evaluate expression e , add scalar/row to result.
 - 2** 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.

9 | Iterated Statements

```

1                                LOOP sG END LOOP
2                                WHILE p LOOP sG END LOOP
3    FOR vi IN [ REVERSE ] e0..e1 [ BY e2 ] LOOP sG END LOOP
4                                FOR vr IN q LOOP sG END LOOP
5    FOREACH va IN [ SLICE n ] ARRAY ea LOOP sG END LOOP

```

- 1 Endless loop (see [EXIT](#) below).
- 2 $p :: \text{bool}$.
- 3 $e_{0,1,2} :: \text{int}$. No BY: $e_2 \equiv 1$. $v_i :: \text{int}$ (auto-declared)
bound to e_0 , $e_0 \pm 1 \times e_2$, $e_0 \pm 2 \times e_2$, ... (REVERSE: $\pm \equiv -$).
- 4 q SQL query. v_r successively bound to resulting rows.
- 5 $e_a :: \tau[]$. No SLICE: $v_a :: \tau$ bound to array elements.
SLICE n : $v_a :: \tau[]$ bound to sub-arrays in n th dimension.

Leaving/Short-Cutting Loops

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

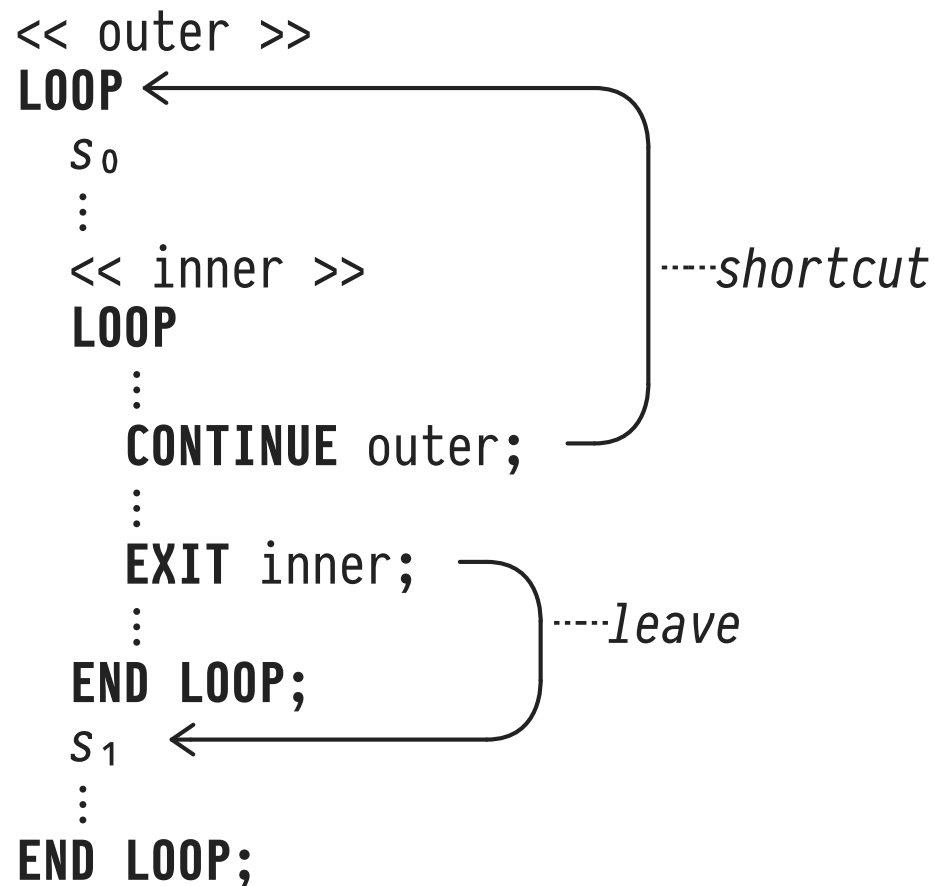
```
<< <label> >> ... LOOP sG END LOOP
```

We may alter the control flow inside a loop via:

```
1      EXIT [<label>] [ WHEN p ]  
2      CONTINUE [<label>] [ WHEN p ]
```

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

Leaving/Shortcutting Loops



- 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 exi1 [, exij]* THEN si ]+
END;
s1      -- next statement if no exception occurred

```

- On error or **RAISE**, search for first matching exception category/name *ex_{ij}*, execute *s_i*, then *s₁*.
- 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

```

1 RAISE [ <level> ] '... % ... % ...' [, e]*
2 RAISE [ <level> ] ex
3 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

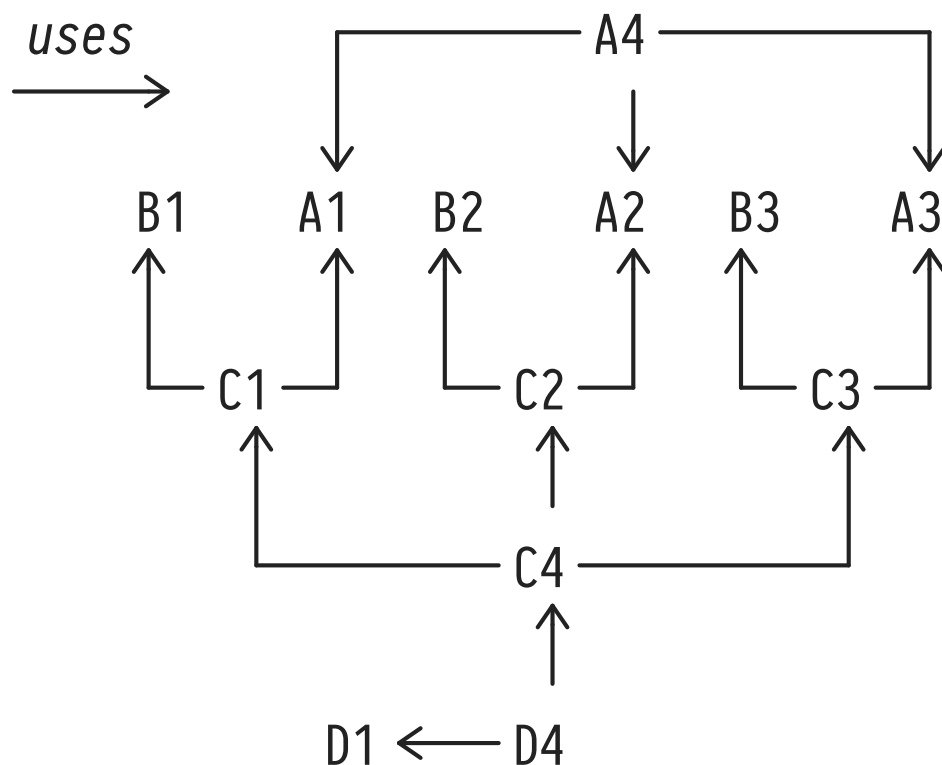
	A	B	C	D
1	1	3.50	3.50	€→£ 0.88
2	2	6.10	12.20	
3	2	0.98	1.96	
4	#items 5		total (€) 17.66	total (£) 15.54

After evaluation

- **A1×B1**: formulæ to be evaluated, total (€): static text.

🔧 A DAG of Cell Dependencies

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



- Formulae 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

```

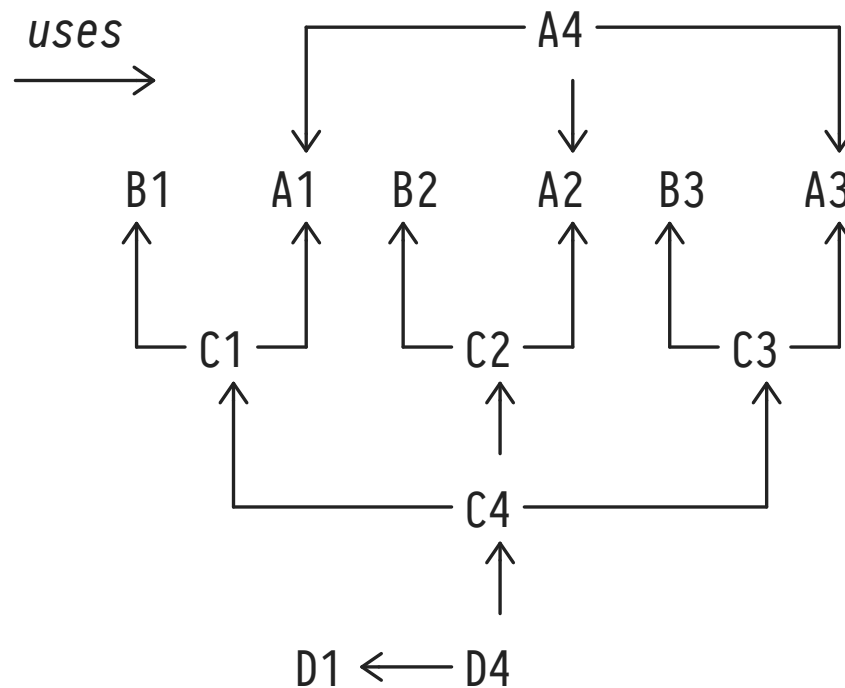
earliest(pos, cell) AS (
  SELECT DISTINCT 0 AS pos, d.uses AS cell
  FROM dependencies AS d
  WHERE d.uses NOT IN (SELECT d1.cell
                        FROM dependencies AS d1) -- } cells d with no
                                                    dependencies

  UNION

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

```

🔧 Dependencies, Topologically Sorted



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

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)

🔧 Formula Representation

- We need a representation of formulae 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 formulae:

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)

```

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;
$$

```

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`:
 - 1 Read formula `e` for `c` off table `sheet`.
 - 2 `v := eval(e)` to find float value `v` of formula `e`.
 - 3 **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
        1 [ SELECT s.formula
           INTO   e
           FROM   sheet AS s
           WHERE  s.cell = c;

        2   v := eval(e);

        3 [ UPDATE sheet AS s
           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  $f(\dots)$  RETURNS  $\tau$  AS
$$
```

```
=====
v1 := (  ); ] embedded
                                SQL Q1
```

```
=====
=====
```

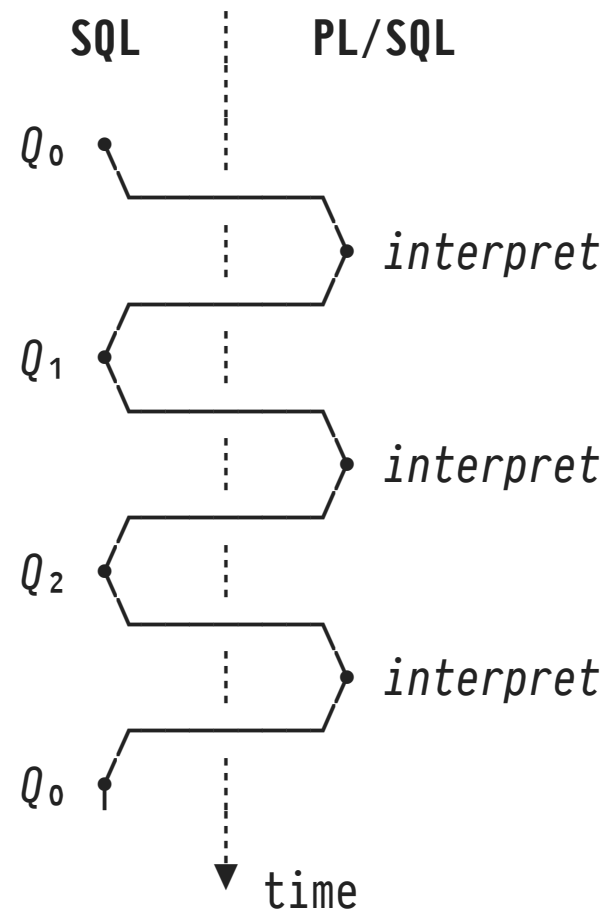
```
=====
=====
```

```
=====
v2 := (  ); ] embedded
                                SQL Q2
```

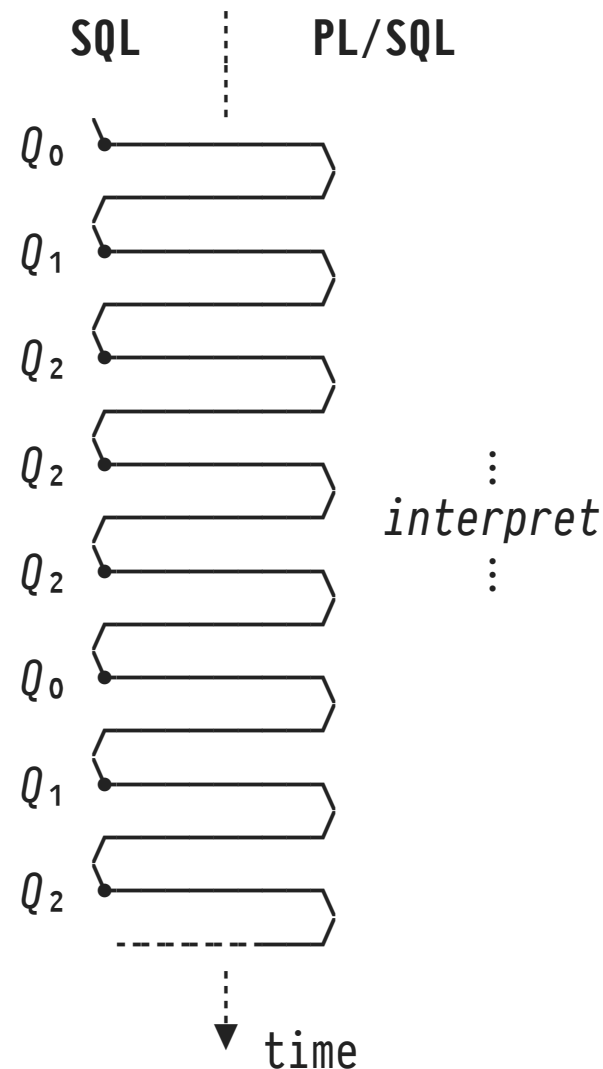
```
=====
```

```
=====
$$ LANGUAGE PLPGSQL;
```

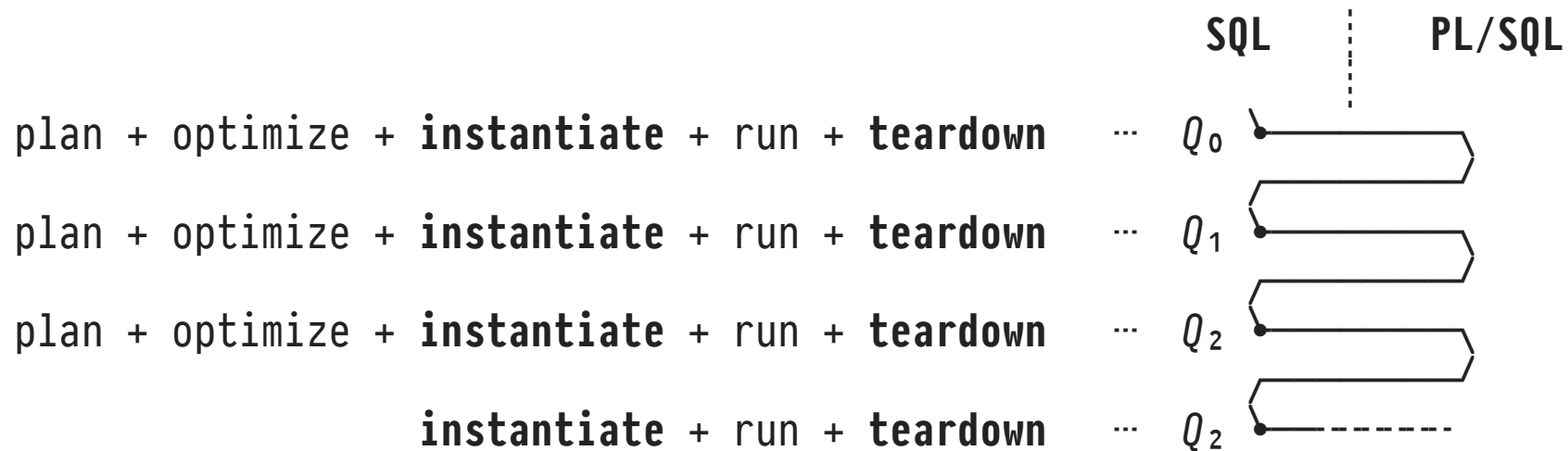
```
SELECT  $f(\dots)$ ; ] top-level
                   SQL Q0
```



SELECT	$f(\dots, t, \dots);$	top-level
FROM	$T \text{ AS } t$	SQL θ_0



SQL↔PL/SQL Context Switches Are Costly



PostgreSQL Runtime Profile 🕒



That's All, Folks

Keep on querying and until next time.