Data is Data and Control Should be Data, Too

Compiling Iterative Table-valued PL/SQL UDFs into Recursive SQL Code

Denis Hirn

denis.hirn@uni-tuebingen.de

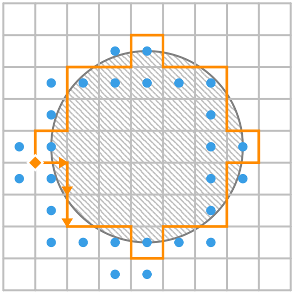
VLDB2023 vancouver

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Elements of PL/SQL

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
        dir := Embedded Query   Q1[·]




        RETURN NEXT cur;

        cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
        EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

Traces contour of 2-dimensional objects:



Elements of PL/SQL: Stateful Variables

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
      dir := [Q1[·] Embedded Query]

      RETURN NEXT cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

Statement sequencing
(straight-line control flow, mutable variables etc.)

Elements of PL/SQL: Complex and Iterative Control Flow

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
      dir := [ Embedded Query                Q1[·] ]



      RETURN NEXT cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

- Statement sequencing (straight-line control flow, mutable variables etc.)
- Arbitrarily complex control flow (via `IF THEN ELSIF`, `WHILE`, `FOR`, `EXIT`, …)

Elements of PL/SQL: Table-Valued Functions

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
      dir := [Embedded Query  Q1[·]]




      RETURN NEXT cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

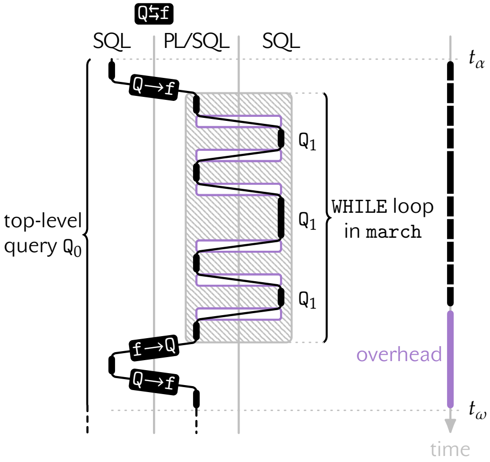
- **Statement sequencing** (straight-line control flow, mutable variables etc.)
- **Arbitrarily complex control flow** (via `IF THEN ELSIF`, `WHILE`, `FOR`, `EXIT`, …)
- **Result accumulation** (`RETURN NEXT`)

SQL ↔ PL/SQL

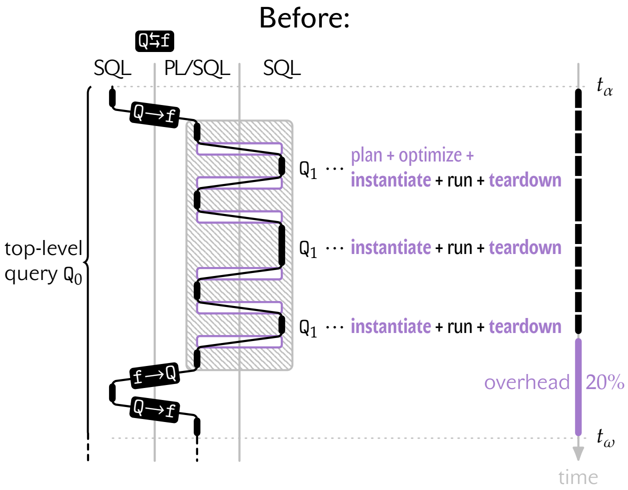
```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
      dir :=  [ Embedded Query ]                              Q1[·]



      RETURN NEXT cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

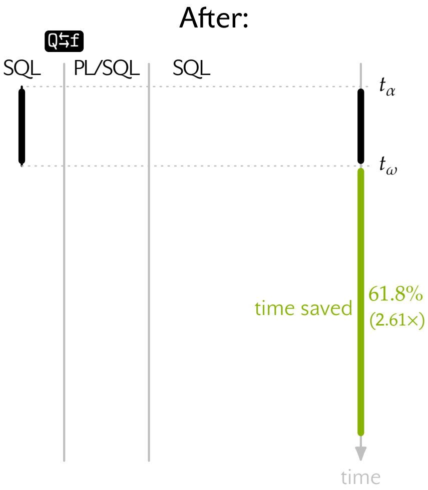
Invoking query $Q_0[\cdot]$:

```
SELECT [redacted]                                   Q0[·]
FROM   [redacted], march([redacted])
```

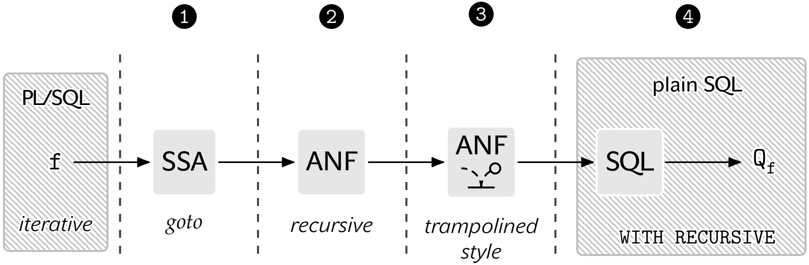


SQL ↔ PL/SQL Context Switches Are Costly





PL/SQL to Plain SQL: Compilation to the Rescue



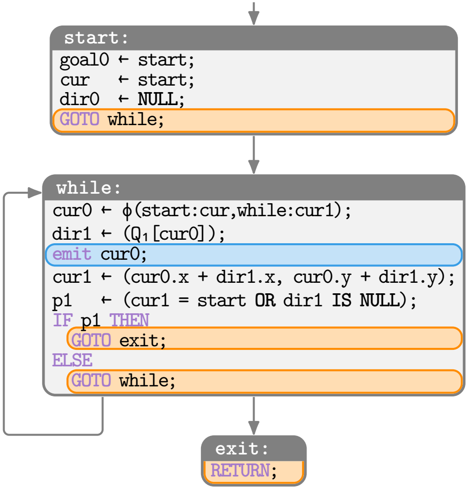
Until now: Only scalar PL/SQL UDFs.

Details: One `WITH RECURSIVE` is Worth Many `GOTO`s, SIGMOD 2021

❶ From PL/SQL to SSA

```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
  DECLARE
    goal vec2 := start;
    cur  vec2 := start;
    dir  vec2;
  BEGIN
    WHILE true LOOP
      dir := [Embedded Query]  Q1[·]

      RETURN NEXT cur;

      cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
      EXIT WHEN cur = goal OR dir IS NULL;
    END LOOP;
  END;
$$ LANGUAGE PLPGSQL STRICT;
```

Control Flow Graph (SSA):



❷ + ❸ From SSA to Trampolined Style ANF

Control Flow Graph (SSA):

```
start:
goal0 ← start;
cur   ← start;
dir0  ← NULL;
GOTO while;
```

```
while:
cur0 ← ϕ(start:cur,while:cur1);
dir1 ← (Q₁[cur0]);
emit cur0;
cur1 ← (cur0.x + dir1.x, cur0.y + dir1.y);
p1   ← (cur1 = start OR dir1 IS NULL);
IF p1 THEN
  GOTO exit;
ELSE
  GOTO while;
```

```
exit:
RETURN;
```

Trampolined Style ANF:

```
start(start)
LET goal0 = start IN
LET cur = start IN
LET dir0 = NULL IN
trampoline(true, false, 'while', NULL, cur)
```

```
trampoline("rec?","data?",call,res,cur)
IF NOT "rec?" THEN res ELSE
CASE call OF
  'while':
    LET dir1 = (Q₁[cur]) IN
    emit cur;
    LET cur1 = (cur0.x + dir1.x, cur0.y + dir1.y) IN
    LET p1   = (cur1 = start OR dir1 IS NULL) IN
    IF p1 THEN
      trampoline(true,false,'exit',NULL,NULL)
    ELSE
      trampoline(true,false,'while',NULL,cur1)
  'exit':
    trampoline(false,false,NULL,NULL,NULL)
```

❹ Trampolined Style WITH RECURSIVE

```
WITH RECURSIVE run("rec?","data?",call,res,cur) AS (
  SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL::vec2 AS res, start AS cur
    UNION ALL -- recursive UNION ALL
  SELECT result.*
  FROM   run,
  LATERAL (SELECT if_p1.*
           FROM     ( Q1[run.cur] ) AS let_dir(dir),
           LATERAL (SELECT NULL AS "rec?", true AS "data?", NULL AS call, run.cur AS res, NULL AS cur
                      UNION ALL
                    SELECT if_p2.*
                    FROM     (SELECT ((run.cur).x + dir.x, (run.cur).y + dir.y) :: vec2) AS let_cur(cur),
                    LATERAL  (SELECT let_cur.cur = start OR dir IS NULL) AS let_p1(p1),
                    LATERAL  (SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL AS res, let_cur.cur AS cur
                               WHERE  NOT p1
                                 UNION ALL
                               SELECT true AS "rec?", false AS "data?", 'exit' AS call, NULL AS res, NULL AS cur
                               WHERE  p1) AS if_p2
                   ) AS if_p1
          WHERE run.call = 'while'
            UNION ALL
          SELECT false AS "rec?", false AS "data?", NULL AS call, NULL AS res, NULL AS cur
          WHERE run.call = 'exit') AS result
  WHERE run."rec?")
SELECT run.res FROM run WHERE run."data?" IS NULL AND run."data?";
```

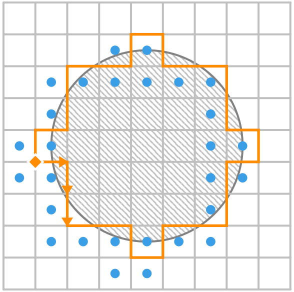
WHILE

EXIT

Union Table with Data Rows and Control Rows

run

rec?	data?	call	res	cur
true	false	while	NULL	(8,7)
false	true	NULL	(8,7)	NULL
true	false	while	NULL	(9,7)
false	true	NULL	(9,7)	NULL
false	true	NULL	(8,8)	NULL
true	false	exit	NULL	(8,8)
false	false	NULL	NULL	NULL



Takeaway Messages

The Good:	UDFs leverage reusability and customized logic.
The Bad:	Naive UDF execution may drastically slow down query execution.
The Solution:	Compile UDFs to pure (recursive) SQL.

Data is Data and Control Should be Data, Too

Compiling Iterative Table-valued PL/SQL UDFs into Recursive SQL Code

Denis Hirn

denis.hirn@uni-tuebingen.de

VLDB2023 vancouver

EBERHARD KARLS UNIVERSITÄT TÜBINGEN