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
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
DECLARE
  goal vec2 := start;
  cur  vec2 := start;
  dir  vec2;
BEGIN
  WHILE true LOOP
    dir := (''SELECT d.dir FROM directions AS d, squares AS s WHERE s.xy = cur AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur)'');
  RETURN NEXT cur;
    cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
    EXIT WHEN cur = goal OR dir IS NULL;
  END LOOP;
END;
$$ LANGUAGE PLPGSQL STRICT;

Traces contour of 2-dimensional objects:
Elements of PL/SQL: Stateful Variables

```sql
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
DECLARE
  goal vec2 := start;
  cur vec2 := start;
  dir vec2;
BEGIN
  WHILE true LOOP
    dir := (SELECT d.dir FROM directions AS d, squares AS s WHERE s.xy = cur AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur));
    RETURN NEXT cur;
    cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
    EXIT WHEN cur = goal OR dir IS NULL;
  END LOOP;
END;
$$
LANGUAGE PLPGSQL STRICT;
```

Statement sequencing
(straight-line control flow, mutable variables etc.)
Elements of PL/SQL: Complex and Iterative Control Flow

```
1 CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
2   DECLARE
3     goal vec2 := start;
4     cur vec2 := start;
5     dir vec2;
6   BEGIN
7     WHILE true LOOP
8       dir :=
9         (SELECT d.dir
10           FROM directions AS d, squares AS s
11           WHERE s.xy = cur
12           AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur));
13       RETURN NEXT cur;
14       cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
15       EXIT WHEN cur = goal OR dir IS NULL;
16     END LOOP;
17   END;
18 $$
19 LANGUAGE PLPGSQL STRICT;
```
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$

DECLARE
  goal vec2 := start;
  cur  vec2 := start;
  dir  vec2;
BEGIN
  WHILE true LOOP
    dir := (SELECT d.dir FROM directions AS d, squares AS s WHERE s.xy = cur AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur));
    RETURN NEXT cur;
    cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
    EXIT WHEN cur = goal OR dir IS NULL;
  END LOOP;
END;
$$ LANGUAGE PLPGSQL STRICT;
CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
DECLARE
    goal vec2 := start;
    cur vec2 := start;
    dir vec2;
BEGIN
    WHILE true LOOP
        dir := (SELECT d.dir FROM directions AS d, squares AS s WHERE s.xy = cur AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur)) 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;
SQL ↔ PL/SQL Context Switches Are Costly

Before:

- top-level query \( Q_0 \)
- plan + optimize + instantiate + run + teardown

After:

- time saved 61.8% (2.61×)
- overhead 20%
Until now: Only scalar PL/SQL UDFs.

Details: One WITH RECURSIVE is Worth Many GOTOs, SIGMOD 2021
From PL/SQL to SSA

CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
DECLARE
  goal vec2 := start;
  cur vec2 := start;
  dir vec2;
BEGIN
  WHILE true LOOP
    dir := (SELECT d.dir
             FROM directions AS d, squares AS s
             WHERE s.xy = cur
                   AND (s.ll, s.lr, s.ul, s.ur) = (d.ll, d.lr, d.ul, d.ur));
    RETURN NEXT cur;
    cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
    EXIT WHEN cur = goal OR dir IS NULL;
  END LOOP;
END;
$$
LANGUAGE PLPGSQL STRICT;

Control Flow Graph (SSA):

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;
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)
```
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 (Q[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?" IS NULL AND run."data?";
### Union Table with Data Rows and Control Rows

<table>
<thead>
<tr>
<th>rec?</th>
<th>data?</th>
<th>call</th>
<th>res</th>
<th>cur</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>while</td>
<td>NULL</td>
<td>(8,7)</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>NULL</td>
<td>(8,7)</td>
<td>NULL</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>while</td>
<td>NULL</td>
<td>(9,7)</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>NULL</td>
<td>(9,7)</td>
<td>NULL</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>NULL</td>
<td>(8,8)</td>
<td>NULL</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>exit</td>
<td>NULL</td>
<td>(8,8)</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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