New Compilation Methods for Complex User-Defined Functions

Compiling Iterative and Recursive UDFs into Recursive SQL Queries

Denis Hirn

✉️ denis.hirn@uni-tuebingen.de
Communication takes time.

- Overhead and context switching is incurred.
- Can slow down the application.
(a) Context switching from application code to the SQL database and back.
“Move your computation close to the data.”
—Michael Stonebraker & Lawrence A. Rowe (1987)
Different Execution Patterns of Database-Backed Applications

(a) Context switching from application code to the SQL database and back.

(b) Ideal program execution with a single context switch to SQL, and a single switch back.
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

```
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.)
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;
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;

Invoking query $Q_0[\cdot]$:
1. SELECT
2. FROM
SQL ↔ PL/SQL Context Switches Are Costly

Before:

- SQL query $Q_0$
- TIME: $t_\omega$
- Overhead: 20%
- Plan + Optimize + Instantiate + Run + Teardown

After:

- SQL query $Q_0$
- TIME: $t_\omega$
- Time saved: 61.8% (2.61x)
PL/SQL to Plain SQL: Compilation to the Rescue

Compile PL/SQL UDFs to plain SQL queries.

Reuse and adapt established off-the-shelf compilation techniques and intermediate representations.
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;
From SSA to ANF

Control Flow Graph (SSA):

- **go to while:**
  - 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;

ANF:

- **start(start):**
  - LET goal0 = start IN
  - LET cur = start IN
  - LET dir0 = NULL IN
  - while(cur)

- **while(cur0):**
  - LET dir1 = (Q1[cur0]) IN
  - EMIT cur0;
  - LET cur1 = (cur0.x + dir1.x, cur0.y + dir1.y) IN
  - LET p1 = (cur1 = start OR dir1 IS NULL) IN
  - IF p1 THEN
    - exit(NULL)
  - ELSE
    - while(cur1)

- **exit(cur0):**
  - RETURN;
From ANF to Trampolined Style ANF

ANF:

```
start(start)
LET goal0 = start IN
LET cur = start IN
LET dir0 = NULL IN
while(cur)
```

```
while(cur0)
LET dir1 = (Q1[cur0]) IN
EMIT cur0;
LET cur1 = (cur0.x + dir1.x, cur0.y + dir1.y) IN
LET p1 = (cur1 = start OR dir1 IS NULL) IN
IF p1 THEN
  exit(NULL)
ELSE
  while(curl1)
exit(cur0)
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)
```
Fixpoint Semantics of Recursive CTEs

(a) SQL syntax for recursive CTEs.

(b) Semi-naive evaluation.

(c) Recursion in a CTE.

Recursive CTEs provide a single loop.
WITH RECURSIVE run("rec?", "data?", call, res, cur) AS (  
  -- trampoline(true, false, 'while', NULL, cur)
  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(dir1),
      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 + dir1.x, (run.cur).y + dir1.y) :: vec2) AS let_cur(cur),
        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?";

SELECT run.res FROM run 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>
## Compiling a Collection of UDFs

| UDF                  | \(|Q_i|\) | Loop constructs | Q→f+f→Q_i | Runtime (speedup) after compilation |
|----------------------|----------|----------------|-----------|------------------------------------|
| bbox                 | 2        |                | 33%       | 61% (1.7x)                         |
| force                | 3        |                | 53%       | 52% (1.9x)                         |
| global               | 1        |                | 33%       | 70% (1.4x)                         |
| items                | 2        |                | 56%       | 33% (3.0x)                         |
| late                 | 1        |                | 34%       | 86% (1.2x)                         |
| march                | 2        |                | 31%       | 72% (1.4x)                         |
| margin               | 3        |                | 16%       | 84% (1.2x)                         |
| markov               | 3        |                | 16%       | 87% (1.2x)                         |
| packing              | 3        |                | 46%       | 73% (1.4x)                         |
| savings              | 6        |                | 42%       | 46% (2.2x)                         |
| sched                | 5        |                | 32%       | 78% (1.3x)                         |
| service              | 1        | Ø              | 28%       | 62% (1.6x)                         |
| ship                 | 3        | Ø              | 17%       | 77% (1.3x)                         |
| sight                | 3        | Ø              | 70%       | 69% (1.5x)                         |
| visible              | 2        | Ø              | 43%       | 57% (1.7x)                         |
| vm                   | 1        | Ø              | 40%       | 77% (1.3x)                         |
| ray                  | 5        |                | 23%       | 829% (0.1x)                        |
| sheet                | 9        |                | 27%       | 186% (0.5x)                        |

Average speedup: ≈ 51%

Measured on PostgreSQL 15
Interim Summary

The Good:  Imperative UDFs leverage reusability and customized logic.
The Bad:    Naive PL/SQL UDF execution may drastically slow down query execution.
The Solution: Compile UDFs to pure (recursive) SQL.
Recursive UDFs
Floyd-Warshall: Textbook Style

1 -- length of shortest path (via nodes 1...n)
2 -- from node s to e
3 CREATE FUNCTION floyd(n int, s int, e int)
4 RETURNS int AS
5 $$
6 \begin{align*}
7 & \text{CASE WHEN } n = 0 \\
8 & \quad \text{THEN } \left( \text{SELECT } p.w \text{ FROM edges AS } p \text{ WHERE } (p.here, p.there) = (s, e) \right) \\
9 & \quad \text{ELSE LEAST } \\
10 & \quad \quad \text{floyd(n-1,s,e),} \\
11 & \quad \quad \text{floyd(n-1,s,n) +} \\
12 & \quad \quad \text{floyd(n-1,n,e))} \\
13 & \quad \text{END;}
14 $$
15 $$\text{LANGUAGE SQL STABLE;}$$
Recursive SQL UDFs Lead to Deep Stacks of Plans

SQL supports UDFs—but hardly encourages programming with them.

UDF `floyd` encounters 96% overhead.
Compilation Overview

Part 1: Iterative UDFs
- PL/SQL
- $f$
- SSA
- GOTO

Part 2: Recursive UDFs
- UDF
- $f$
- SSA$_{REC}$
- ANF$_{REC}$
- CPS
- GOTO
- recursive
- higher-order

Intermediate Steps:
- ANF
- $\text{ANF} + \gamma$
- SQL
- $Q_\ell$
- single loop
- WITH RECURSIVE
Translation from SQL UDF to SSA

1 -- length of shortest path (via nodes 1...n)
2 -- from node s to e
3 CREATE FUNCTION floyd(n int, s int, e int)
4 RETURNS int AS
5 $$
6 SELECT CASE WHEN (v0 = 0)[n]
7 THEN ($$Q_1$$)
8 ELSE LEAST (v0, v1 + v2)[floyd(n-1,s,e),
9 floyd(n-1,s,n),
10 floyd(n-1,n,e)]
11 END;
12 $$
13 LANGUAGESQLSTABLE;

floyd is in SSA, but contains recursive calls.
Translation from SSA to ANF

fun floyd(n int, s int, e int) : int {
    start:
    v0 ← q[n];
    IF v0
    THEN RETURN q[s,e];
    ELSE GOTO κthen;
    κthen:
    n ← ϕ(start:n);
    s ← ϕ(start:s);
    e ← ϕ(start:e);
    REC v1 = floyd(n-1, s, e);
    REC v2 = floyd(n-1, s, n);
    REC v3 = floyd(n-1, n, e);
    RETURN q[v1,v2,v3];
}

fun floyd(n int, s int, e int) : int {
    LET v0 = q[n] IN
    IF v0
    THEN
        q[s,e]
    ELSE
        REC v1 = floyd(n-1, s, e) IN
        REC v2 = floyd(n-1, s, n) IN
        REC v3 = floyd(n-1, n, e) IN
        q[v1,v2,v3];
    }

Transformation into CPS: Tail Recursion Only!

```
fun floyd(n int, s int, e int) : int {
  LET v0 = q[n] IN
  IF v0 THEN
    q[s,e]
  ELSE
    REC v1 = floyd(n-1, s, e) IN
    REC v2 = floyd(n-1, s, n) IN
    REC v3 = floyd(n-1, n, e) IN
    q[v1,v2,v3];
}
```

```
def floyd(n int, s int, e int | 𝜅0 : stack) : int {
  LET v0 = q[n] IN
  IF v0 THEN
    𝜅0(q[s,e])
  ELSE
    cnt 𝜅1(v1) {
      cnt 𝜅2(v2) {
        cnt 𝜅3(v3) {
          𝜅0(q[v1,v2,v3])
        }
      }
    }
    floyd(n-1, n, e | 𝜅3)
  }
  floyd(n-1, s, n | 𝜅2)
  floyd(n-1, s, e | 𝜅1)
}
```

floyd is tail-recursive, but higher-order.
Defunctionalization: Functions are Data, Too

```ml
fun start(n int, s int, e int) : int {
  floyd(n, s, e, NULL, EMPTY_STACK())
}

fun floyd(n int, s int, e int, x int, ks stack) : int {
  LET v0 = q1[n] IN
  IF v0 THEN apply(n, s, e, q2[s,e], ks)
  ELSE floyd(n-1, s, e, NULL, PUSH([('k1',n,s,e,NULL,NULL)], ks))
}

fun apply(n int, s int, e int, x int, ks stack) : int {
  IF EMPTY(ks) THEN x
  ELSE LET ⟨l, n, s, e, v1, v2, v3⟩ = TOP(ks) IN
  LET ks = TAIL(ks) IN
  CASE l OF
    'k1': floyd(n-1, s, n, NULL, PUSH([('k2',n,s,e,x,NULL)], ks))
    'k2': floyd(n-1, n, e, NULL, PUSH([('k3',n,s,e,v1,x)], ks))
    'k3': apply(n, n, e, q3[v1,v2,x], ks)
}
```

floyd and apply: first-order and (mutually) tail-recursive.
### Experiments

<table>
<thead>
<tr>
<th>Recursive SQL UDF</th>
<th>Recursion</th>
<th>Overhead [%] UDF</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected components</td>
<td>2-way</td>
<td>90.64</td>
<td>8.2×</td>
</tr>
<tr>
<td>Dynamic Time Warping (DTW)</td>
<td>3-way</td>
<td>97.59</td>
<td>15.7×</td>
</tr>
<tr>
<td>Expression tree evaluation</td>
<td>2-way</td>
<td>96.00</td>
<td>21.6×</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>3-way</td>
<td>96.74</td>
<td>14.7×</td>
</tr>
<tr>
<td>Parse with a finite state machine</td>
<td>linear</td>
<td>94.08</td>
<td>9.2×</td>
</tr>
<tr>
<td>Find longest common substring</td>
<td>2-way</td>
<td>98.43</td>
<td>12.8×</td>
</tr>
<tr>
<td>Compute Mandelbrot set</td>
<td>tail</td>
<td>97.43</td>
<td>19.2×</td>
</tr>
<tr>
<td>Trace border of 2D object</td>
<td>linear</td>
<td>89.37</td>
<td>6.8×</td>
</tr>
<tr>
<td>Construct file system path names</td>
<td>tail</td>
<td>92.27</td>
<td>10.0×</td>
</tr>
<tr>
<td>Run program on a virtual machine</td>
<td>tail</td>
<td>98.17</td>
<td>183.1×</td>
</tr>
</tbody>
</table>

Before compilation ≈ 95% overhead

Speedup through compilation 📈 6.8×–183×
Conclusion and Future Work

Compilation…
- “Move(s) your computation close to the data.” 🌟
- Enables UDFs to run on DBMSs without native UDF support. 🌟
- Improves performance on legacy systems. 🌟
- May slow down UDFs with large intermediate results. 🍀

Future Work

LATERAL Join Free Translation. Investigate compilation for different database systems.
Batched Evaluation. Encode multiple UDF calls into the working table.
WITH TRAMPOLINE. Research specialized database support for trampolined-style queries.
New Compilation Methods for Complex User-Defined Functions

Compiling Iterative and Recursive UDFs into Recursive SQL Queries

Denis Hirn

✉️ denis.hirn@uni-tuebingen.de
## Related Work

<table>
<thead>
<tr>
<th>Approach</th>
<th>Linear Control Flow</th>
<th>Looping Control Flow</th>
<th>Recursion</th>
<th>SQL-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>This Work</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Froid</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Aggify</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Functional-Style UDFs</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fun SQL</td>
<td>✓</td>
<td>×</td>
<td>only tail-recursion</td>
<td>×</td>
</tr>
</tbody>
</table>
Cursor Loops

```sql
CREATE FUNCTION f(...) RETURNS ... AS $$
DECLARE
    cur RECORD;
BEGIN
    FOR cur IN (SELECT * FROM Q LIMIT 1 OFFSET i) LOOP
        ...
        END LOOP;
    RETURN ...;
END;
$$ LANGUAGE PLPGSQL STRICT;
```

```sql
CREATE FUNCTION f(...) RETURNS ... AS $$
DECLARE
    curs RECORD[];
    cur RECORD;
BEGIN
    curs := (SELECT ARRAY_AGG(q) FROM Q AS q);
    FOREACH cur IN ARRAY curs LOOP
        ...
        END LOOP;
    RETURN ...;
END;
$$ LANGUAGE PLPGSQL STRICT;
```
### Table run

<table>
<thead>
<tr>
<th>rec?</th>
<th>call</th>
<th>res</th>
<th>n</th>
<th>s</th>
<th>e</th>
<th>x</th>
<th>$\kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>floyd</td>
<td>NULL</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>floyd</td>
<td>NULL</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>floyd</td>
<td>NULL</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>3</td>
<td>$[k_1, \ldots]$</td>
</tr>
<tr>
<td>3</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>floyd</td>
<td>NULL</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td>$[k_2, \ldots]$</td>
</tr>
<tr>
<td>12</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>apply</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>NULL</td>
<td>2</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

- Table run contains result of $floyd(2, 2, 3) = 2$.

### Memoization

- Rows contain *intermediate* results.
- Save $(\text{args}, x)$ in table $\text{memo}$.
  - Lookup on subsequent invocations.
- $floyd$: Avoids $O(3^n)$ recursive calls.
  - Dynamic programming “for free”.

---

3/3