Functional Programming
Close to the Data

PPDP
September 20, 2022 — 07:05am GMT

Torsten Grust
University of Tübingen
First off, release the data from its relational jail.

Our mission: heists in PL land!
“First off, release the data from its relational jail.”

(Dumb) Data Storage

(Indexes) Tables $T_i$

(Clever) Computation

```
*.py
...
data = [{...}, ..., {...}]
...
res = f(data) #
```
"Move your computation close to the data." [Stonebraker]
SQL, a Truly Declarative Programming Language

“SQL, Lisp, and Haskell\(^1\) are the only programming languages that I've seen where one spends more time thinking than typing.”

—Philip Greenspun

\(^1\) Let me add APL to that list.
Recursion in SQL
The addition of *recursion* to SQL:1999 changes everything:

**Expressiveness** SQL becomes a *Turing-complete language* and thus—in principle—a general-purpose PL (albeit with a particular flavor).

**Efficiency** ! No longer are queries guaranteed to terminate or to be evaluated with polynomial effort.

Like a pact with the 🙏 — but the payoff is plenty.
Shape of a Recursive SQL Query (Common Table Expression)

WITH RECURSIVE \( T \) AS ( 
\[ q_0 \]
\[ q_0 \]
\[ \text{UNION ALL} \] 
\[ q_0(T) \]

\) TABLE \( T; \)

- Semantics in a nutshell:

\[ T \equiv q_0 \cup q_0(q_0) \cup q_0(q_0(q_0)) \cup \ldots \]

iterate evaluation of \( q_0 \) (until \( q_0 = \phi \))
WITH RECURSIVE cc(n, s, e, w) AS (  
SELECT 0 AS n, s, e, edge.w  
FROM (SELECT s, e  
FROM generate_series(1, N) AS s,  
generate_series(1, N) AS e) AS _(s,e)  
LEFT OUTER JOIN  
edges AS edge  
ON (edge.here, edge.there) = (s, e)  
)
UNION ALL
(WITH cc(n, s, e, w) AS (  
TABLE cc  
)  
SELECT f0.n + 1 AS n, f0.s, f0.e, LEAST(f0.w, f1.w + f2.w) AS w  
FROM cc AS f0, cc AS f1, cc AS f2  
WHERE f1.s = f0.s AND f1.e = f0.n + 1  
AND f2.s = f0.n + 1 AND f2.e = f0.e  
AND f0.n <= N  
)
)  
TABLE cc  
ORDER BY n, s, e;
WITH RECURSIVE floyd(n,s,e,w) AS ( 
  SELECT 0 AS n, s, e, edge.w 
  FROM   (SELECT s, e 
           FROM   generate_series(1, N) AS s, 
                   generate_series(1, N) AS e) AS _(s,e) 
         LEFT OUTER JOIN 
           edges AS edge 
           ON (edge.here, edge.there) = (s, e) 

  UNION ALL 
  
  (WITH floyd(n,s,e,w) AS ( 
    TABLE floyd 
  ) 
    SELECT f0.n + 1 AS n, f0.s, f0.e, LEAST(f0.w, f1.w + f2.w) AS w 
    FROM   floyd AS f0, floyd AS f1, floyd AS f2 
    WHERE  f1.s = f0.s   AND  f1.e = f0.n + 1 
           AND  f2.s = f0.n + 1 AND  f2.e = f0.e 
           AND  f0.n <= N 
  ) 

  TABLE floyd 
ORDER BY n, s, e;
\[
\begin{array}{ll}
\text{if} & (\pi, i, j) = (0, 0, 0), \\
\infty & \text{otherwise}
\end{array}
\]

\[
(\pi, i, j) = \min((\pi, i, j-1), (\pi, i-1, j) + (\pi, i-1, j-1))
\]
Floyd–Warshall: Textbook Style, 3-fold Recursion

\[ floyd(0, s, e) = \begin{cases} 
  w & \text{if } s \xrightarrow{-w} e \\
  \infty & \text{otherwise}
\end{cases} \]

\[ floyd(n, s, e) = \min(floyd(n-1, s, e), floyd(n-1, s, n) + floyd(n-1, n, e)) \]

\[
\begin{array}{c|c|c}
\text{edges} & \text{here} & \text{there} \\
\hline
1 & 3 & -2 \\
2 & 3 & 4 \\
3 & 4 & 2 \\
4 & 2 & -1 \\
\end{array}
\]

\[ floyd(4, 2, 3) = 2 \]
Recursive UDFs
This function is a **1:1 transcription** from the textbook:

```haskell
/* Length of shortest path s→e (∞ if there is none) */
floyd : (int,int,int) → int
floyd(n,s,e) =
    case n = 0 of
        true:  weight(s,e)  /
             s → w→ e |
        false: min(floyd(n-1,s,e),
                     floyd(n-1,s,n) + floyd(n-1,n,e))
```
This UDF is a 1:1 transcription from textbook to SQL:

```
-- Length of shortest path s→e (NULL ≡ ∞ if there is none)
CREATE FUNCTION floyd(n int, s int, e int) RETURNS int
AS $$
SELECT CASE WHEN n = 0
  THEN (SELECT edge.w
        FROM edges AS edge
        WHERE (edge.here,edge.there) = (s,e)) -- s →w→ e
  ELSE LEAST(floyd(n-1,s,e),
              floyd(n-1,s,n) + floyd(n-1,n,e))
END;
$$ LANGUAGE SQL STABLE;
```
• On each invocation of recursive SQL UDF `floyd(·,·,·)`:  
  1. Analyze query in UDF body, generate query plan.  
  2. Instantiate that plan, evaluate, tear down.

```sql
=# SELECT floyd(10,6,7);
: 
=# SELECT * FROM pg_stat_user_functions;
```

<table>
<thead>
<tr>
<th></th>
<th>funcname</th>
<th>calls</th>
<th>total_time</th>
<th>self_time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>floyd</td>
<td>!88573</td>
<td>3172.024</td>
<td>3172.024</td>
</tr>
</tbody>
</table>

• PostgreSQL inlines simple SQL functions to depth 2 only.²

² No bashing intended here—some RDBMSs outlaw recursion in SQL UDFs in the first place.
• SQL supports user-defined functions—but it hardly encourages *programming* with these functions. 🙄
HERE: Treat UDFs Like Functions (not as a Piece of Plan)

<table>
<thead>
<tr>
<th>SQL</th>
<th>Functional Programming</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>f ↦</td>
<td>plain f</td>
<td>Qf</td>
</tr>
<tr>
<td>1</td>
<td>recursive</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>recursive first order</td>
<td>WITH RECURSIVE</td>
</tr>
<tr>
<td>2</td>
<td>f in CPS</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>defunctionalize</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>trampolined style</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mutually tail-recursive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>first order</td>
<td></td>
</tr>
<tr>
<td></td>
<td>single loop</td>
<td></td>
</tr>
</tbody>
</table>

Steps 1...4 bank on decades-old and proven FP techniques.
CREATE FUNCTION floyd(n int, s int, e int) RETURNS int AS $$
SELECT CASE WHEN \([v_0 = 0]|n\) \[1\]
    THEN (SELECT edge.w
          FROM edges AS edge
          WHERE (edge.here,edge.there) = (\([v_0,v_1]|s,e\) \[2\])
        END;
$$ LANGUAGE SQL STABLE;

- \[1,...,4\]: Need not peek inside \([\ldots]\). Unwrap at very end.
CREATE FUNCTION floyd(n int, s int, e int) RETURNS int
AS $$
SELECT CASE WHEN 1[n] 
  THEN 2[s,e]

  ELSE 3[floyd(4[n],s,e),
      floyd(4[n],s,n),
      floyd(4[n],n,e)]

END;
$$ LANGUAGE SQL STABLE;

• 1,...,4: Need not peek inside [...]. Unwrap at very end.
**Transformation into CPS: Tail Recursion Only!**

```haskell
floyd : (int, int, int) → int
floyd(n, s, e) =
    case \[n\] of
        case true: \[s, e\]
        false: \[floyd(\[n\], s, e),
               floyd(\[n\], s, n),
               floyd(\[n\], n, e)\]
```

- A SFW block hides inside \[2\]—but we don't open the box.
- This is the UDF's **backbone:**
  1. `case...of`: identify base/recursive cases,
  2. `floyd(...)`: recursive function invocations.
**Transformation into CPS: Tail Recursion Only!**

\[
floyd : (\text{int}, \text{int}, \text{int}, \text{int} \rightarrow \text{int}) \rightarrow \text{int}
\]

\[
floyd(n, s, e, k) = \text{case } \mathbf{1}[n] \text{ of }
\]

- **true:** \(k(\mathbf{2}[s, e])\)
- **false:** \(floyd(\mathbf{4}[n], s, e, \lambda s_1.\ floyd(\mathbf{4}[n], s, n, \ldots \text{ A}))\)
- \(\lambda s_2.\ floyd(\mathbf{4}[n], n, e, \ldots \text{ B})\)
- \(\lambda s_3.\ k(\mathbf{3}[s_1, s_2, s_3])))\)

- Uses continuation \(k\) to pass intermediate results \(s_i\) on.
- \(\text{floyd}\) is tail-recursive ☑, but higher-order ☹.
Defunctionalization: Functions are Data, Too

\[ floyd : (\text{int}, \text{int}, \text{int}, \text{stack}) \rightarrow \text{int} \]
\[ floyd(n, s, e, ks) = \]
  \[
  \text{case } 1[n] \text{ of } \]
  \[
  \text{true: } \text{apply}(2[s, e], ks) \]
  \[
  \text{false: } floyd(4[n], s, e, \text{PUSH}(\langle A, n, s, e, \varnothing, \varnothing \rangle, ks))
  \]

\[ apply : (\text{int}, \text{stack}) \rightarrow \text{int} \]
\[ apply(x, ks) = \text{let } \langle k, n, s, e, s_1, s_2 \rangle = \text{TOP}(ks) \text{ in } \]
  \[
  \text{case } k \text{ of } \]
  \[
  2: x \]
  \[
  A: floyd(4[n], s, n, \text{PUSH}(\langle B, n, \varnothing, e, x, \varnothing \rangle, \text{POP}(ks)))) \]
  \[
  B: floyd(4[n], n, e, \text{PUSH}(\langle C, \varnothing, \varnothing, \varnothing, s_1, x \rangle, \text{POP}(ks)))) \]
  \[
  C: apply(3[s_1, s_2, x], \text{POP}(ks))
  \]

- \textit{floyd} and \textit{apply}: first-order and tail-recursive \(\bigcirc\), ...
- ... but mutually invoke each other \(\bigcirc\).
Function \textit{trampoline} embodies a \textit{single-loop} computation, just like SQL's \textsc{WITH RECURSIVE}.

- **Trampolined Style**: Single Loop
An Iterative “Interpreter” for the Recursive UDF

```sql
trampoline(fn,n,s,e,x,ks,res) =
case fn of
  ⑩:  res
else: trampoline(interpret(fn,n,s,e,x,ks,res))

interpret(fn,n,s,e,x,ks,res) =
case fn of
  ⑩: case ⑩[n] of
true: (⑩,□,□,□,⑩[s,e],ks)
falses: (⑩,⑩[n],s,e,□),PUSH(⟨⑩,n,s,e,□,□,ks⟩),□
⑩: let ⟨k,n,s,e,s₁,s₂⟩ = TOP(ks) in
case k of
  ⑩: (⑩,□,□,□,□,□)
  ⑩: (⑩,⑩[n],s,n,□),PUSH(⟨⑩,n,□,e,x,□⟩,POP(ks)),□
  ⑩: (⑩,⑩[n],n,e,□),PUSH(⟨⑩,□,□,□,s₁,x⟩,POP(ks)),□
  ⑩: (⑩,□,□,□,⑩[s₁,s₂,x],POP(ks)),□

fn n s e x ks res
```

- Think of tuples (fn,n,s,e,x,ks,res) as “instructions”.

---

4 An Iterative “Interpreter” for the Recursive UDF
Plain SQL: A WITH RECURSIVE-based Interpreter for floyd

WITH RECURSIVE trampoline(fn,n,s,e,x,ks,res) AS (
  SELECT ♂, n, s, e, □, PUSH((②,□,□,□,□,□),EMPTY), □
  floyd(n,s,e)

  UNION ALL -- recursive UNION

  SELECT interpret.*
  FROM trampoline AS t,
  LATERAL (SELECT (TOP(t.ks)).* AS k(k,n,s,e,s₁,s₂),
  LATERAL (  
  SELECT ♂, □, □, □, ⚁[k.s₁,k.s₂,t.x], POP(t.ks), □
  WHERE t.fn = ♂ AND k.k = ⚁

  UNION ALL
  :
  ) AS interpret(fn,n,s,e,x,ks,res)
)

TABLE trampoline;

• Single query, planned once, no (recursive) UDF calls. 🌈
### Functional Programming On Top of SQL Engines

#### Recursive SQL UDF Overhead

<table>
<thead>
<tr>
<th>Speedup via</th>
<th>[2345]</th>
<th>[6789]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTW</td>
<td>[97.59]%</td>
<td>[15.6] ×</td>
</tr>
<tr>
<td>Connected components</td>
<td>[90.64]%</td>
<td>[8.1] ×</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>[96.74]%</td>
<td>[14.7] ×</td>
</tr>
<tr>
<td>2D Marching Squares</td>
<td>[89.37]%</td>
<td>[6.8] ×</td>
</tr>
<tr>
<td>Virtual machine simulation</td>
<td>[98.17]%</td>
<td>[183.1] ×</td>
</tr>
<tr>
<td>Expression tree evaluation</td>
<td>[96.00]%</td>
<td>[21.5] ×</td>
</tr>
<tr>
<td>⋮</td>
<td>[≈ 95.00]%</td>
<td>[≈ 10] ×</td>
</tr>
</tbody>
</table>

---

Treat UDFs for what they are: functions.

No invasion of RDBMS kernel: SQL transformation.

---

#### Table Trampoline = Instruction Trace + Memoization

<table>
<thead>
<tr>
<th>fn</th>
<th>n</th>
<th>s</th>
<th>e</th>
<th>x</th>
<th>ks</th>
<th>res</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td>ack</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>ks</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Row \(\leftarrow\): Result of top-level UDF call (floyd(2,2,3) = 2).

#### Memoization

- Rows with fn = a (apply continuation) pass on intermediate result x.

- Rows \(\leftarrow\): Save (args, x) in table memo. Lookup on subsequent invocations.

- floyd: Avoids \(O(3^n)\) recursive calls. Dynamic programming “for free.”
Functional Programming On Top of SQL Engines

<table>
<thead>
<tr>
<th>Recursive SQL UDF</th>
<th>Overhead</th>
<th>Speedup via WITH RECURSIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Time Warping (DTW)</td>
<td>97.59%</td>
<td>15.6×</td>
</tr>
<tr>
<td>Connected components</td>
<td>90.64%</td>
<td>8.1×</td>
</tr>
<tr>
<td>Floyd–Warshall</td>
<td>96.74%</td>
<td>14.7×</td>
</tr>
<tr>
<td>2D Marching Squares</td>
<td>89.37%</td>
<td>6.8×</td>
</tr>
<tr>
<td>Virtual machine simulation</td>
<td>98.17%</td>
<td>183.1×</td>
</tr>
<tr>
<td>Expression tree evaluation</td>
<td>96.00%</td>
<td>21.5×</td>
</tr>
<tr>
<td></td>
<td>≈ 95.00%</td>
<td>≈ 10×</td>
</tr>
</tbody>
</table>

- Treat UDFs for what they are: **functions**.
- No invasion of RDBMS kernel: SQL→SQL transformation.
“Move your computation close to the data.”

—Mike Stonebraker

<table>
<thead>
<tr>
<th>More Application/Algorithms Expressed in SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes-Hut $n$-body simulation</td>
</tr>
<tr>
<td>CASH algorithm (robust clustering)</td>
</tr>
<tr>
<td>Cellular automata (<em>Game-of-Life</em>-style)</td>
</tr>
<tr>
<td>CYK parsing</td>
</tr>
<tr>
<td>Distance vector routing</td>
</tr>
<tr>
<td>Graph algorithms (shortest paths, connected components, ...)</td>
</tr>
<tr>
<td>Handwriting recognition</td>
</tr>
<tr>
<td>Liquid/heat flow simulations, water percolation</td>
</tr>
<tr>
<td>Loose index scans</td>
</tr>
<tr>
<td>Markov decision processes (robot control)</td>
</tr>
<tr>
<td>Spreadsheet-style formula evaluation</td>
</tr>
<tr>
<td>Traffic simulation</td>
</tr>
<tr>
<td>Turing machine simulation</td>
</tr>
<tr>
<td>Sessionization, bin fitting</td>
</tr>
<tr>
<td>Z-order image processing</td>
</tr>
</tbody>
</table>