Functional Programming Close to the Data

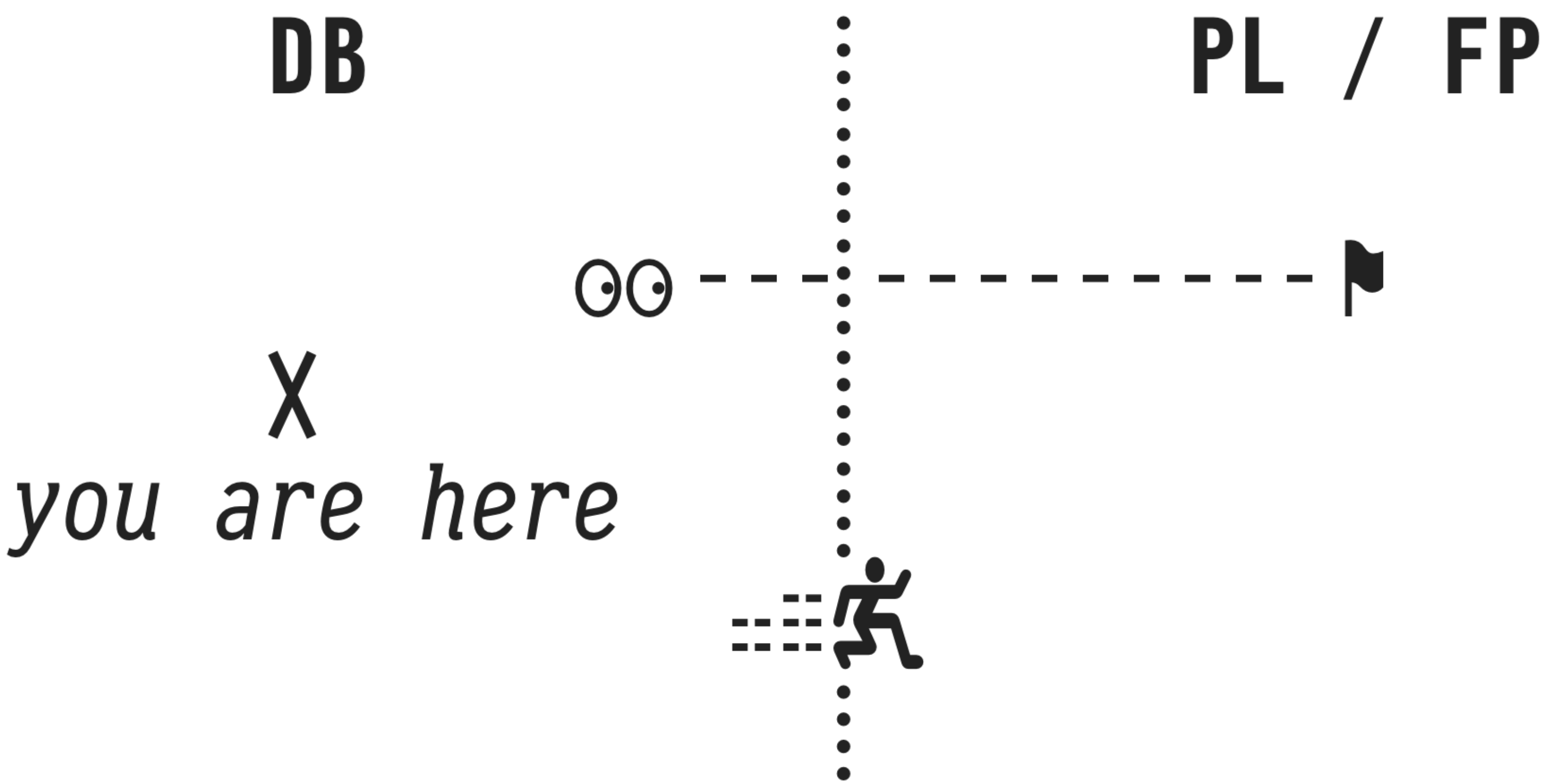
PPDP

September 20, 2022 — 07:05am GMT 😴

Torsten Grust

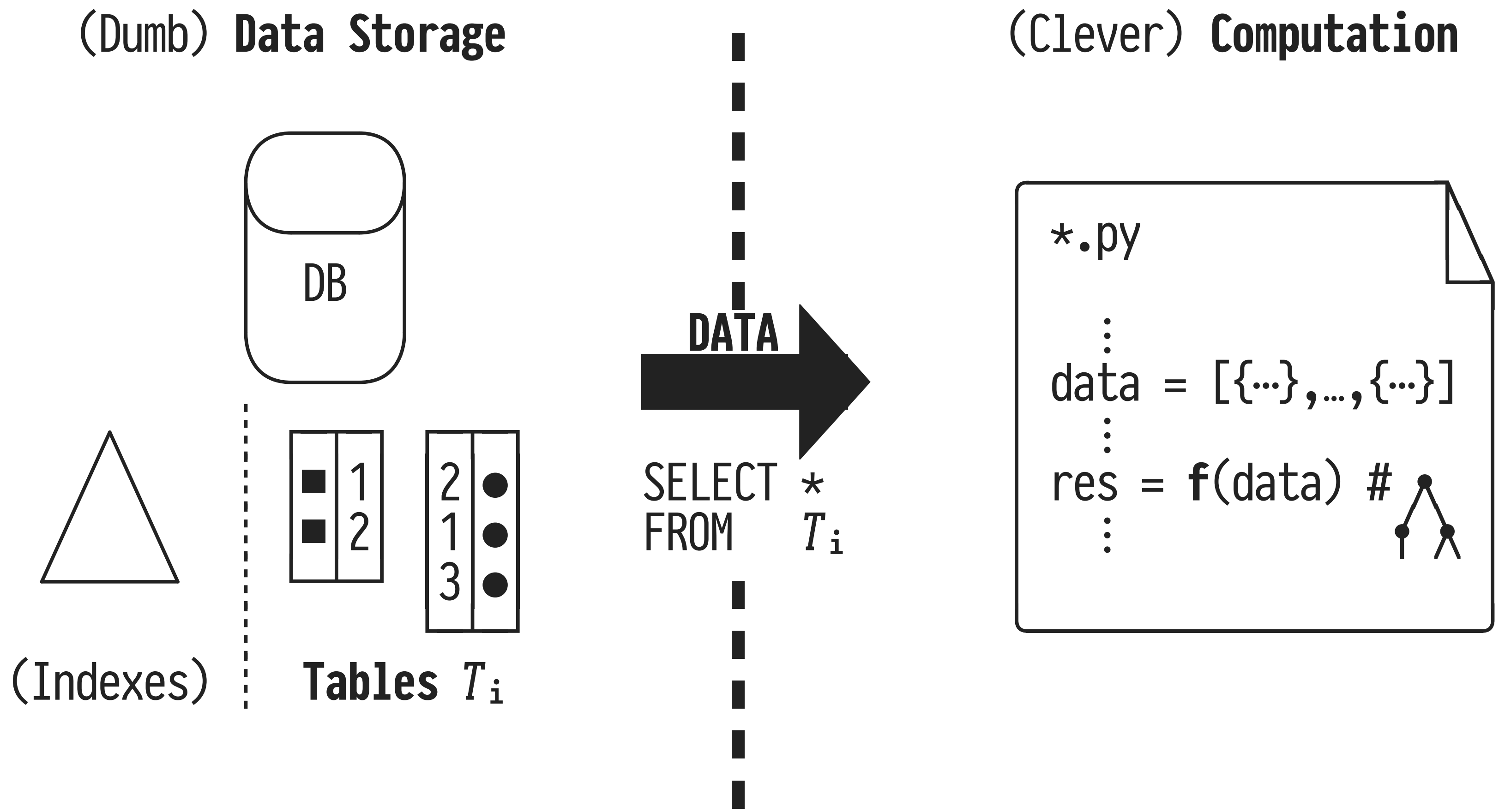
University of Tübingen

DB Research in Tübingen

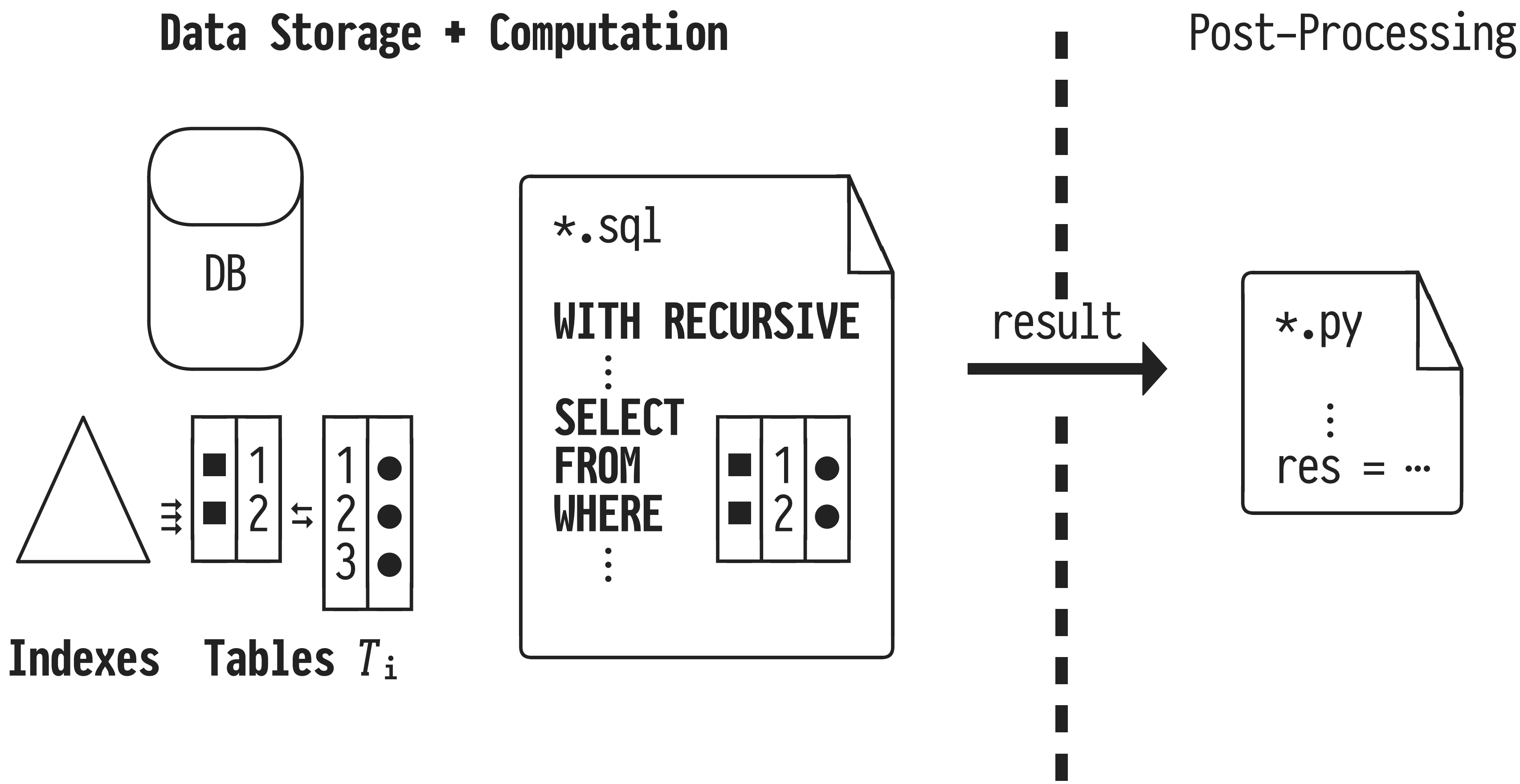


Our mission: heists in PL land!

“First off, release the data from its relational jail.”



"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

But Can I Do That Using SQL? — You Sure Can.

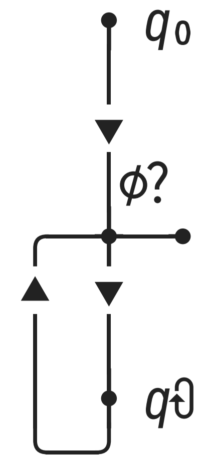
The addition of **recursion** to SQL:1999 changes everything:

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 (
q₀    [ ...
        UNION ALL
q↺(T) [ ...
      )
      TABLE T;
```



- Semantics in a nutshell:

$$T \equiv q_0 \uplus \underbrace{q^{\circlearrowleft}(q_0) \uplus q^{\circlearrowleft}(q^{\circlearrowleft}(q_0)) \uplus \cdots}$$

iterate evaluation of $q^{\circlearrowleft}$ (until $q^{\circlearrowleft} = \phi$)

Recursive SQL CTE (↓ *Connected Graph Components*)

```
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;
```

edges

here	there	w
1	3	-2
2	1	4
2	3	3
3	4	2
4	2	-1

Oops… This ↓ was *All Pairs Shortest Path* (Floyd-Warshall)

```
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;
```

edges

here	there	w
1	3	-2
2	1	4
2	3	3
3	4	2
4	2	-1

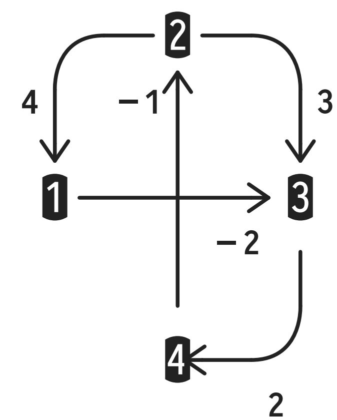
Floyd-Warshall: Textbook Style, 3-fold Recursion

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

$$floyd(n,s,e) = \mathbf{min}(floyd(n-1,s,e), floyd(n-1,s,n) + floyd(n-1,n,e))$$

edges

here	there	w
1	3	−2
2	1	4
2	3	3
3	4	2
4	2	−1



$floyd(4,2,3) = 2$

Recursive UDFs

Floyd–Warshall: Textbook Code (FP Style)

This function is a **1:1 transcription** from the textbook:

```
/* 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))
```

Can't We Have This? A Recursive SQL UDF

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;
```

SQL UDF Invocation? An Opportunity to Plan Afresh…

- On **each invocation** of recursive SQL UDF floyd(•,•,•):
  1. Analyze query in UDF body, generate query plan.
  2. ❶ Instantiate that plan, ❷ evaluate, ❸ tear down.

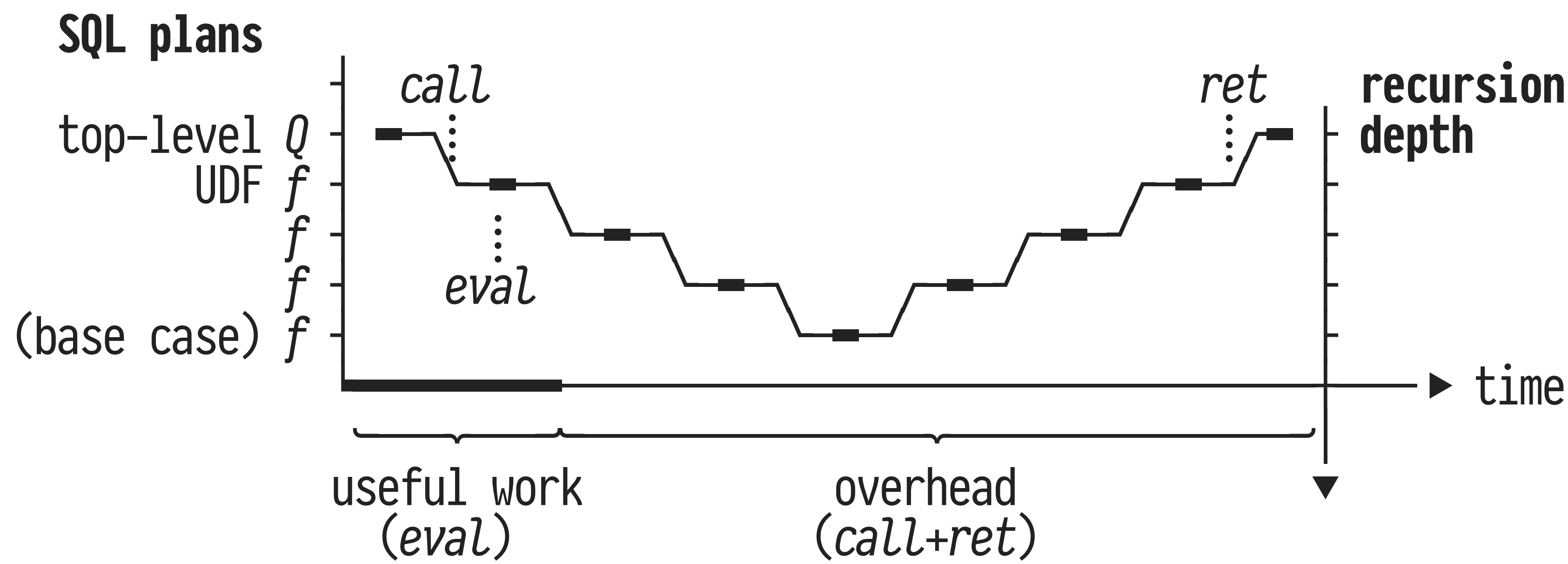
```
=# SELECT floyd(10,6,7);
⋮
=# SELECT * FROM pg_stat_user_functions;
```

…	funcname	calls	total_time	self_time
…	floyd	⚠️88573	3172.024	3172.024

- 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.

Recursive SQL UDFs Lead to Deep Stacks of Plans



- SQL supports user-defined functions—but it hardly encourages *programming* with these functions. 👎

HERE: Treat UDFs Like *Functions* (not as a Piece of Plan)

SQL	Functional Programming				SQL
	❶	❷	❸	❹	
f ➧	plain *f*	*f* in **CPS**	**defunctionalize**	**trampolined style**	➧ *Qf*
recursive UDF	recursive	tail-recursive	mutually tail-recursive	single loop	**WITH RECURSIVE**
	first order	higher order	first order	first order	

Steps ❶···❹ bank on decades-old and proven FP techniques.

❶ Put SQL Subexpressions in Boxes ❶ (+ Leave Them There)

```
CREATE FUNCTION floyd(n int, s int, e int) RETURNS int
AS $$
 SELECT CASE WHEN ⌈v₀ = 0⌋[n]❶
   THEN (⌈SELECT edge.w
          FROM   edges AS edge
          WHERE  (edge.here,edge.there) = (v₀,v₁)⌋[s,e]❷)

   ELSE ⌈LEAST(v₀, v₁+v₂)⌋[floyd(⌈v₀-1⌋[n]❹,s,e),
                            floyd(⌈v₀-1⌋[n]❹,s,n),
                            floyd(⌈v₀-1⌋[n]❹,n,e)]❸
  END;
$$ LANGUAGE SQL STABLE;
```

- ❶,…,❹: Need not peek inside ⌈⋯⌋. Unwrap at very end.

❶ Put SQL Subexpressions in Boxes ❶ (+ Leave Them There)

```
CREATE FUNCTION floyd(n int, s int, e int) RETURNS int
AS $$
 SELECT CASE WHEN ❶[n]
   THEN ❷[s,e]

   ELSE ❸[floyd(❹[n],s,e),
           floyd(❹[n],s,n),
           floyd(❹[n],n,e)]
  END;
$$ LANGUAGE SQL STABLE;
```

- ❶,…,❹: Need not peek inside ⌈···⌋. Unwrap at very end.

❶ Transition from SQL to FP

```
floyd : (int,int,int) → int
floyd(n,s,e) =
  case ❶[n] of
    true:  ❷[s,e]
    false: ❸[floyd(❹[n],s,e),
              floyd(❹[n],s,n),
              floyd(❹[n],n,e)]
```

- A SFW block hides inside ❷—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.

❷ Transformation into CPS: Tail Recursion Only!

```
floyd : (int,int,int,int → int) → int
floyd(n,s,e,k) =                 └──────┬──────┘
  case ❶[n] of                          k
    true:  k(❷[s,e])
    false: floyd(❹[n],s,e,
Ⓐ ············· λs1.floyd(❹[n],s,n,
Ⓑ ················· λs2.floyd(❹[n],n,e,
Ⓒ ····················· λs3.k(❸[s1,s2,s3]))))
```

- Uses continuation k to pass intermediate results s_i on.
- *floyd* is **tail-recursive** 👍, but **higher-order** 👎.

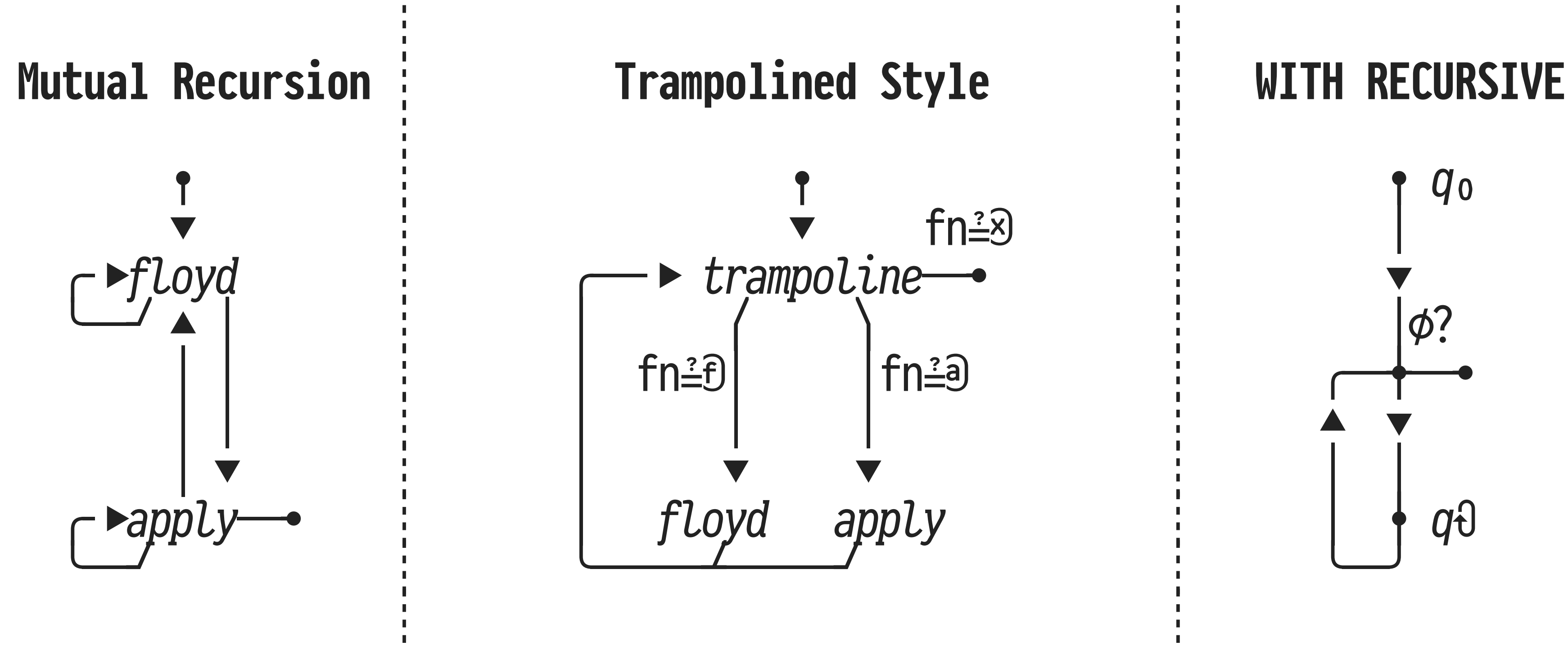
❸ Defunctionalization: Functions are Data, Too

```
floyd : (int,int,int,stack) → int
floyd(n,s,e,ks) =
  case ❶[n] of                        invoke Ⓐ!  environment
    true:  apply(❷[s,e],ks)    ↘          ⏟
    false: floyd(❹[n],s,e,PUSH(⟨Ⓐ,n,s,e,□,□⟩,ks))

apply : (int,stack) → int
apply(x,ks) = let ⟨k,n,s,e,s₁,s₂⟩ = TOP(ks) in
  case k of
    Ⓩ: x
    Ⓐ: floyd(❹[n],s,n,PUSH(⟨Ⓑ,n,□,e,x,□⟩, POP(ks)))
    Ⓑ: floyd(❹[n],n,e,PUSH(⟨Ⓒ,□,□,□,s₁,x⟩, POP(ks)))
    Ⓒ: apply(❸[s₁,s₂,x], POP(ks))
```

- *floyd* and *apply*: **first-order** and **tail-recursive** 👍👍, …
- … but mutually invoke each other 👎.

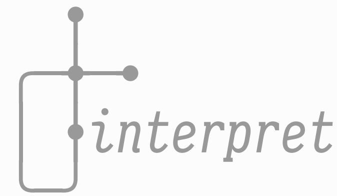
4 Trampolined Style: Single Loop



- Function *trampoline* embodies a **single-loop** computation, just like SQL's **WITH RECURSIVE** 👍.

4 An Iterative "Interpreter" for the Recursive UDF

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




```
interpret(fn,n,s,e,x,ks,res) =
 case fn of
 ⓕ: case ❶[n] of
      true:  (ⓐ,□    ,□,□,❷[s,e]    ,ks                          , □ )
      false: (ⓕ,❹[n],s,e,□          ,PUSH(〈Ⓐ,n,s,e,□,□〉,ks)      , □ )
 ⓐ: let 〈k,n,s,e,s1,s2〉 = TOP(ks) in
      case k of
       ⓩ:   (ⓧ,□    ,□,□,□          ,□                           , x )
       Ⓐ:   (ⓕ,❹[n],s,n,□          ,PUSH(〈Ⓑ,n,□,e,x,□〉,POP(ks)) , □ )
       Ⓑ:   (ⓕ,❹[n],n,e,□          ,PUSH(〈Ⓒ,□,□,□,s1,x〉,POP(ks)), □ )
       Ⓒ:   (ⓐ,□    ,□,□,❸[s1,s2,x],POP(ks)                      , □ )

            fn   n   s e     x                  ks                  res
```

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

🏁 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((ⓩ,□,□,□,□,□),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  ⓐ, □, □, □, ❸[k.s₁,k.s₂,t.x], POP(t.ks), □     ⎬ interpreter
    WHERE   t.fn = ⓐ AND k.k = ⓒ                           ⎪
                                                           ⎪
    UNION ALL                                              ⎪
      ⋮                                 instruction        ⎪
  ) AS interpret(fn,n,s,e,x,ks,res)                        ⎭
)
TABLE trampoline;
```

- Single query, **planned once**, no (recursive) UDF calls. 👍

Table *trampoline* ≡ Instruction Trace + Memoization

trampoline

fn	n	s	e	x	ks	res	
ⓕ	2	2	3	□		□	
ⓕ	1	2	3	□		□	
ⓕ	0	2	3	□		□	
ⓐ	□	□	□	**3**	⋮	□	◁
ⓐ	□	□	□	**4**	*s*	□	◁
ⓐ	□	□	□	**-2**	*t*	□	◁
ⓐ	□	□	□	**2**	*a*	□	◁
ⓕ	1	2	2	□	*c*	□	
⋮	⋮	⋮	⋮		*k*		
ⓐ	□	□	□	**3**	*s*	□	◁
ⓐ	□	□	□	**4**	⋮	□	◁
ⓐ	□	□	□	**-2**		□	◁
ⓐ	□	□	□	**2**		□	◁
ⓧ	□	□	□	□		**2**	◀

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

Memoization

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

Recursive SQL UDF	Overhead	Speedup via WITH RECURSIVE
Dynamic Time Warping (DTW)	97.59%	15.6×
Connected components	90.64%	8.1×
Floyd-Warshall	96.74%	14.7×
2D Marching Squares	89.37%	6.8×
Virtual machine simulation	98.17%	183.1×
Expression tree evaluation	96.00%	21.5×
⋮		
	≈ 95.00%	≈ **10×**

- Treat UDFs for what they are: **functions.**
- No invasion of RDBMS kernel: SQL→SQL transformation.

Process Your Data in its Own Habitat!

> *Move your computation close to the data.*
>
> —Mike Stonebraker

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

Functional Programming Close to the Data

Torsten Grust
University of Tübingen

Denis Hirn • Chris Duta • Tim Fischer

@Teggy | db.inf.uni-tuebingen.de/grust