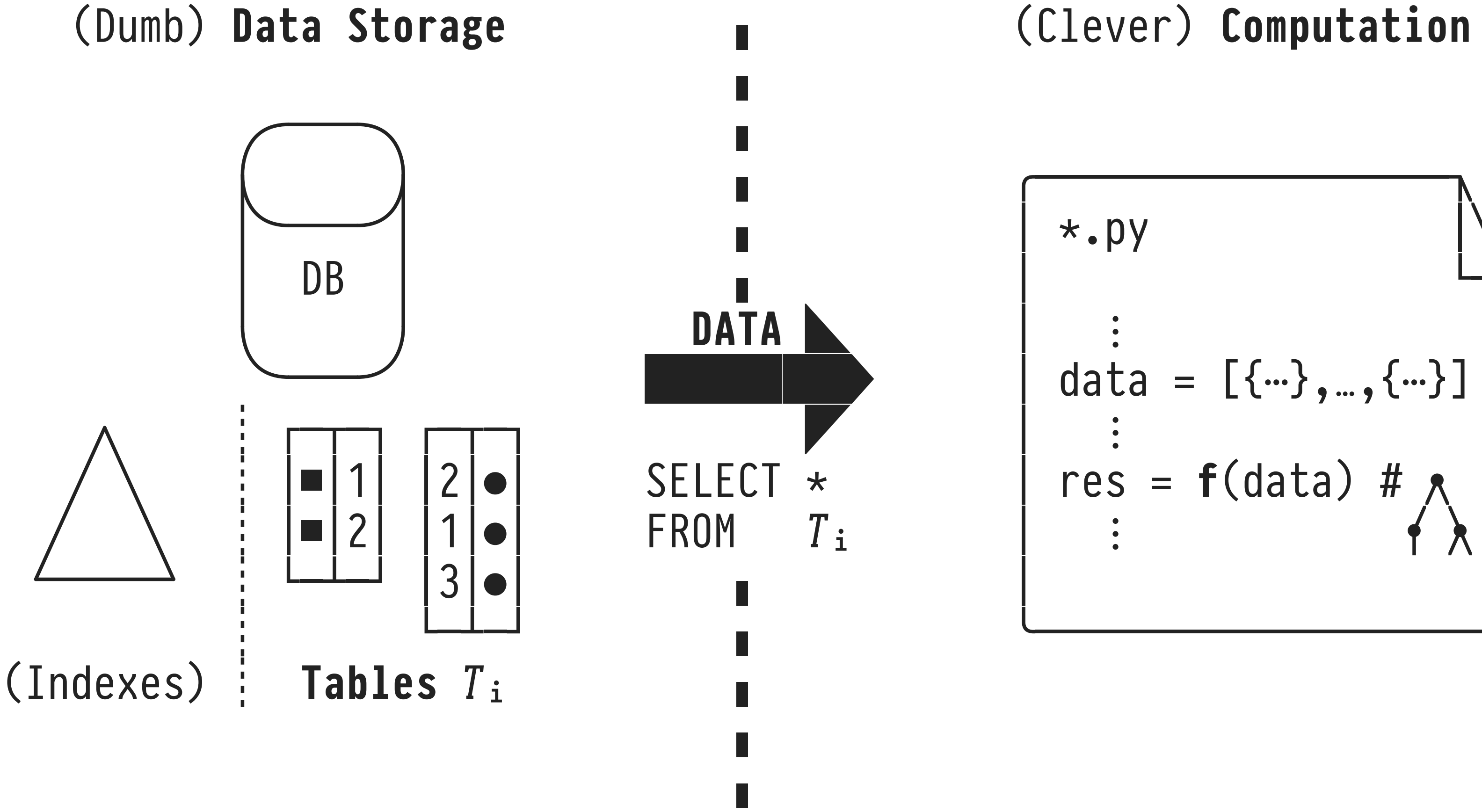
Processing Data in its Own Habitat

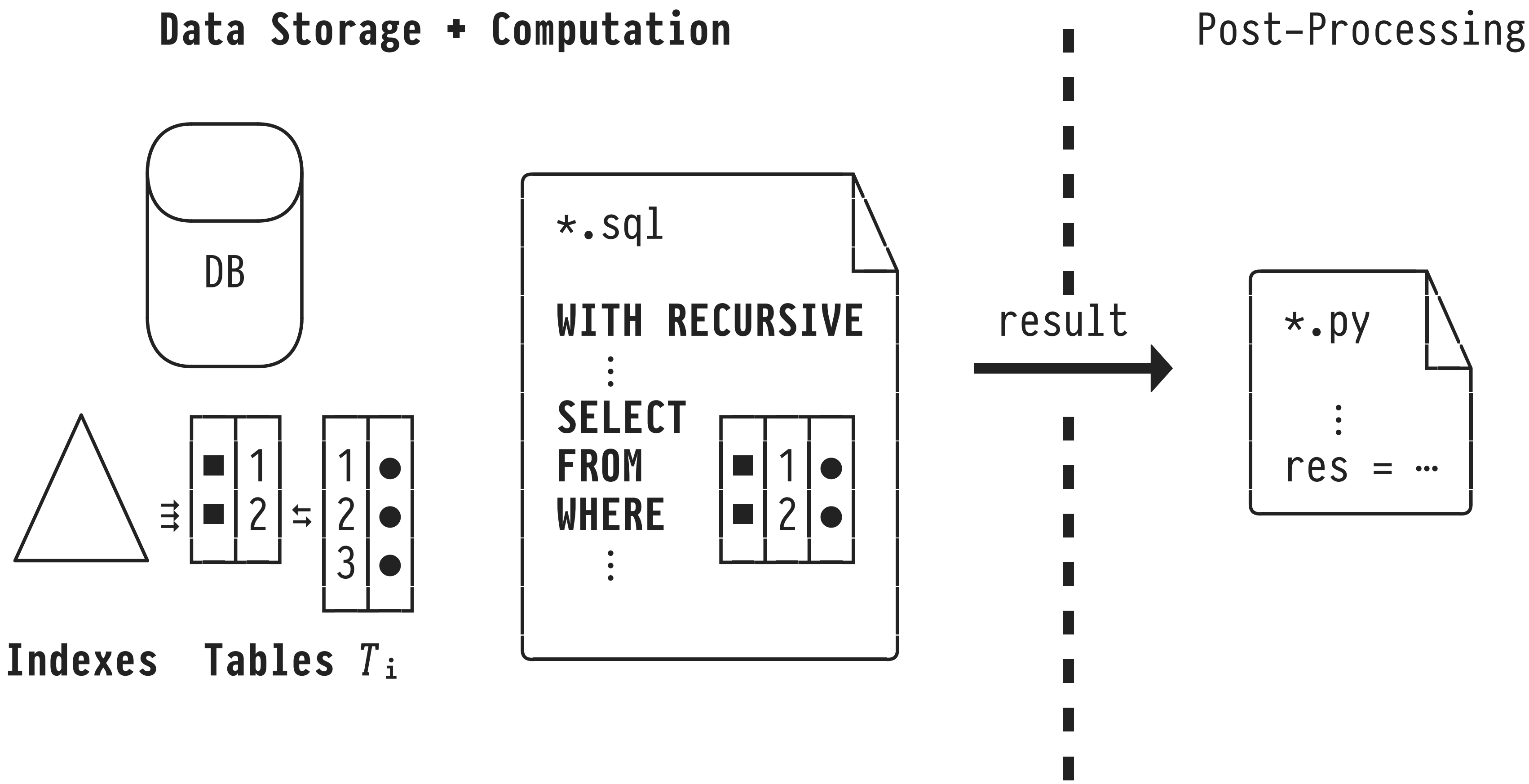
May 28, 2019

Torsten Grust
University of Tübingen

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



“Move your computation close to the data,” says Stonebraker



SQL Has Moved on Since SQL:1992

SQL Clauses and Expressions (as of SQL:2016)		
	⋮	⋮
"New" SQL	**WITH RECURSIVE**	*letrec*, iteration
	MATCH_RECOGNIZE	row pattern matching
	ROWS FROM	*zip*
	UDFs (SQL, procedural)	user-def'd functions
	WINDOW (**OVER**(…))	*scan*, …
	array_agg, unnest [**WITH ORDINALITY**]	array processing
	multisets, row values, enums	non-scalar table cells
	mode(), …, **FILTER**, **WITHIN GROUP**	filtered/ordered agg's
	WITH	*let*
	CASE…**WHEN**…**ELSE**…**END**	*if*…*then*…*else*
	LATERAL	[…| x ← xs, y ← f(x)]
Old SQL	**SELECT**…**FROM**…**WHERE**	join, selection, projection
	GROUP BY…**HAVING**	grouping/aggregation
	ORDER BY	ordering (cosmetic)

> "*SQL, Lisp, and Haskell 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

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

```
WITH RECURSIVE
  T(…) AS (
      q_0              -- base case query, evaluated once

    UNION [ ALL ]
                       -- recursive query refers to T itself,
      q↺(T)            --   evaluated repeatedly
  )
q_1(T)                 -- final post-processing query
```

- (Almost the) Semantics in a nutshell:

$$q_1(\underbrace{q_{\circlearrowleft}(\cdots q_{\circlearrowleft}(q_{\circlearrowleft}(q_0))\cdots) \cup \cdots \cup q_{\circlearrowleft}(q_{\circlearrowleft}(q_0)) \cup q_{\circlearrowleft}(q_0)}_{\text{repeated evaluation of } q_{\circlearrowleft} \text{ (when to stop?)}} \cup q_0)$$

Semantics of a Recursive SQL Query (**UNION ALL** Variant)

Iterative and recursive semantics—both are equivalent:

<pre>iterate(q⟲, q0):
  r ← q0
  t ← r
  while t ≠ ∅
  │ t ← q⟲(t)
  └ r ← r ⊎ t
  return r</pre>
	<pre>recurse(q⟲, r):
	  if r ≠ ∅ **then**
	  │ **return** r ⊎ recurse^all(q⟲, q⟲(r))
	  else
	  └ **return** ∅</pre>

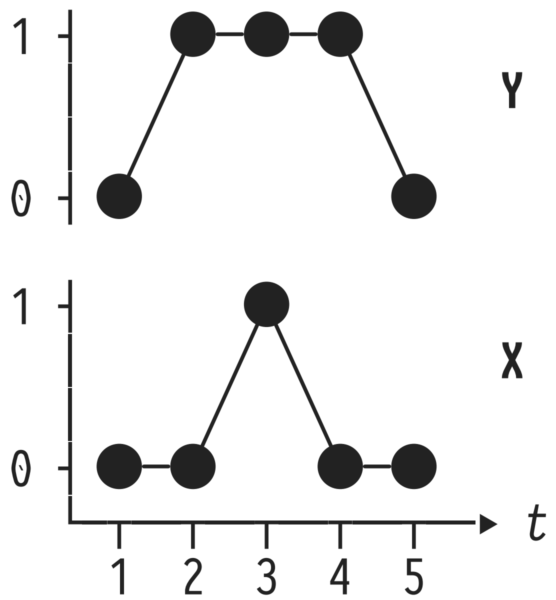
- Invoke the recursive variant via recurse(q⟲, q_0).
- Recursive query q⟲ sees *all rows added in the last iteration/recursive call*. Exit if no rows added.
- Could **immediately emit t** — no need to build r.

X

t	*val*
1	0
2	0
3	1
4	0
5	0

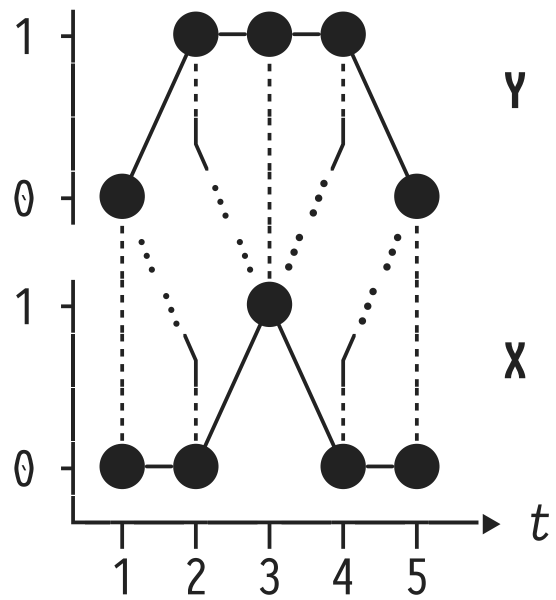
Y

t	*val*
1	0
2	1
3	1
4	1
5	0



Dynamic Time-Warping Distance (DTW)

Stretch/compress series **X** = (x_i) and **Y** = (y_j) along the time axis to align both optimally:



Q: DTW — Hand-Crafted SQL Code (Not in Any Textbook…)

```
WITH RECURSIVE matrix(i,j,val) AS (
    (SELECT X.t, Y.t, abs(X.val - Y.val)
     FROM   X, Y
     ORDER BY X.t, Y.t
     LIMIT 1)

  UNION

⋆   SELECT step.i, step.j, MIN(step.val)
    FROM (SELECT step.i, step.j, m.val + step.val
          FROM   matrix AS m,
                 (VALUES (1,1), (0,1), (1,0)) AS Δ(i,j),
                 LATERAL (
                   SELECT m.i+Δ.i, m.j+Δ.j, abs(X.val - Y.val)
                   FROM X, Y
                   WHERE (X.t,Y.t) = (m.i+Δ.i,m.j+Δ.j)) AS step(i,j,val)
          WHERE (step.i,step.j) <= (i,j)
⋆        ) AS step(i,j,val)
⋆   GROUP BY step.i, step.j                                  --
  )                                                          --
  SELECT MIN(m.val) AS dtw                                   -- with (i,j) = (5,5):
  FROM   matrix AS m                                         --
  WHERE  (m.i,m.j) = (i,j);                                  -- ⌛ ≈ 3ms
```

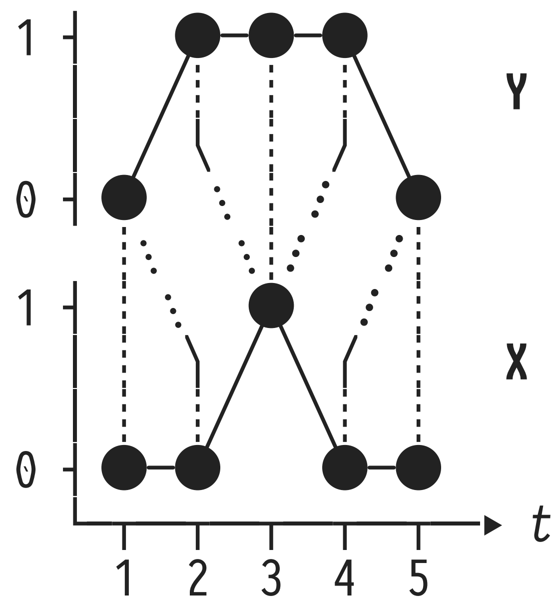
dtw
0.0

Q: DTW — Textbook Recursive Algorithm

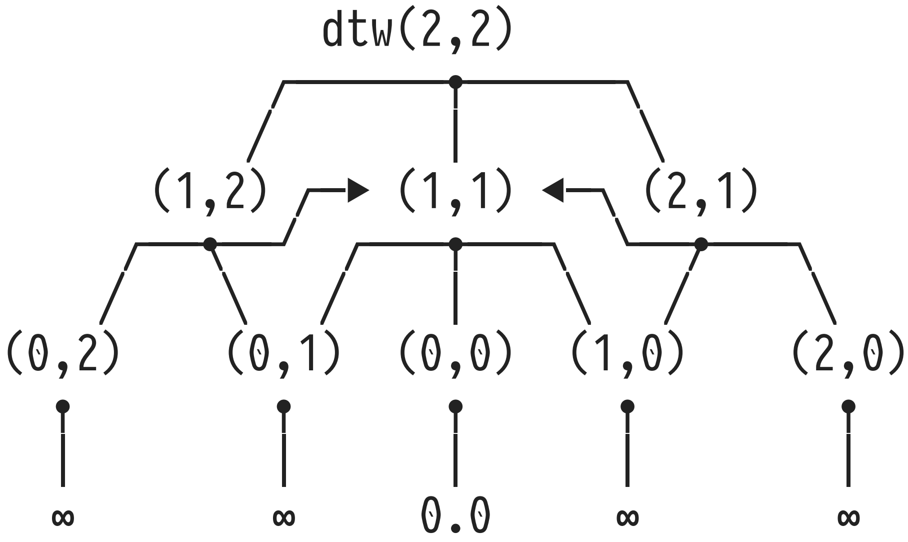
$$\mathrm{dtw}(0, 0) = 0.0$$
$$\mathrm{dtw}(i, 0) = \infty$$
$$\mathrm{dtw}(0, j) = \infty$$
$$\mathrm{dtw}(i, j) = |x_i - y_j| + \min \begin{Bmatrix} \mathrm{dtw}(i-1, j-1) \\ \mathrm{dtw}(i-1, j\ \ ) \\ \mathrm{dtw}(i\ \ , j-1) \end{Bmatrix} \quad \begin{matrix} \text{-- match} \\ \text{-- stretch} \\ \text{-- compress} \end{matrix}$$

Matrix
dtw(i,j)

$$\begin{array}{cc} & \\ j & \begin{array}{c} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{array} \end{array} \begin{array}{c} \begin{pmatrix} 3 & 3 & 1 & 0 & \mathbf{0} \\ 3 & 3 & 0 & 1 & 2 \\ 2 & 2 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \\ \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ i \end{array}$$



Phase ❶: Resulting Call Graph



- Table **call_graph** encodes dtw(·,·)'s call *graph*.
- Compute each missing result ▢ once, bottom-up
  ⇛ realizes sharing during evaluation.

Table **call_graph**

(i,j)	**site**	**(i↗,j↗)**	**result**
(2,2)	❶	(1,1)	▢
(2,2)	❷	(1,2)	▢
(2,2)	❸	(2,1)	▢
(1,1)	❶	(0,0)	▢
(1,1)	❷	(0,1)	▢
(1,1)	❸	(1,0)	▢
(1,2)	❶	(0,1)	▢
(1,2)	❷	(0,2)	▢
(1,2)	❸	(1,1)	▢
⋮	⋮	⋮	⋮
(0,0)	▢	▢	0.0
(0,1)	▢	▢	∞
(1,0)	▢	▢	∞
(0,2)	▢	▢	∞
(2,0)	▢	▢	∞

Phase ❷: Final Result of Evaluation

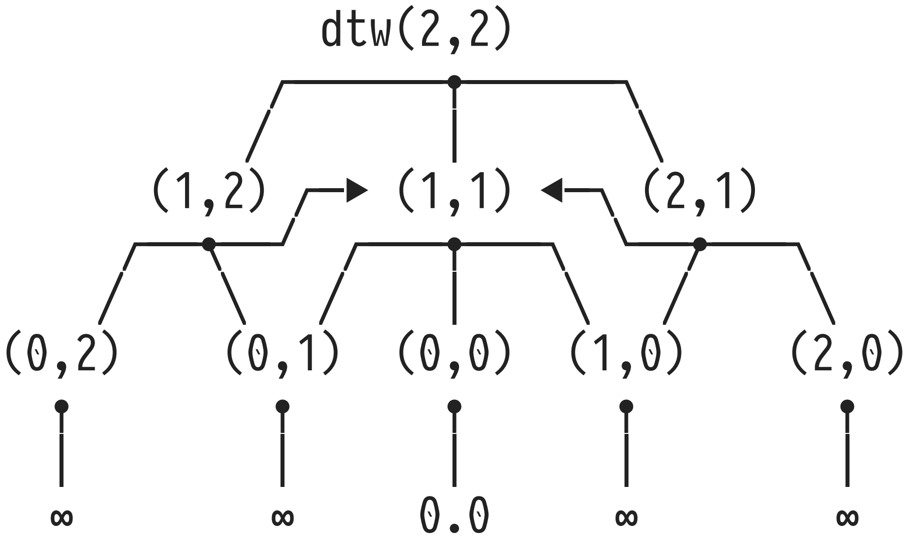


Table **evaluation**

(i,j)	result
(2,2)	1
(2,1)	0
(1,1)	0
(1,2)	1
(2,0)	∞
(1,0)	∞
(0,2)	∞
(0,1)	∞
(0,0)	0

- Column result of row (i,j) = (2,2) holds function call result.
- Hold on to table evaluation to **memoize** function dtw(·,·).

Write SQL UDFs in Functional Style!

- Compilation of “functional-style” SQL UDFs into recursive CTEs can come close to hand-crafted CTEs:

DTW Variant	⌛ Wall-Clock Time for *dtw(5,5)*
Hand-crafted Recursive CTE	3 ms
Recursive SQL UDF (native)	≈1100 ms
Recursive SQL UDF (compiled into CTE)	15 ms

- Opportunities:
  1. **Memoization:** keep table evaluation (STABLE UDFs: within transaction, IMMUTABLE UDFs: across transactions).
  2. **Parallelization:** chop call graph, evaluate sub-graphs in ∥.

Process Your Data in its Own Habitat!

> *Move your computation close to the data. [paraphrased]*
>
> —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