Advanced SQL

05 — Recursion

Torsten Grust
Universität Tübingen, Germany

Computational Limits of SQL

SQL has grown to be an **expressive data-oriented language.** Intentionally, it has *not* been designed as a general-purpose programming language:

1. *SQL does not loop forever:*
   Any SQL query is expected to **terminate**, regardless of the size/contents of the input tables.

2. *SQL can be **evaluated efficiently**:*
   A SQL query over table T of c columns and r rows can be evaluated in $O(r^c)$ space and time.[1]

[1] SQL cannot compute the set of all subsets of rows in T which requires $O(2^r)$ space, for example.

A Giant Step for SQL

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

Expressiveness SQL becomes a **Turing-complete language** and thus 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 magnificient…

Recursion in SQL: WITH RECURSIVE

Recursive common table expresssion (CTE):

```
WITH RECURSIVE
  ‹T_1›(‹c_1,1›,…,‹c_1,k_1›) AS (
    ‹q_1› ),
   ⋮
  ‹T_n›(‹c_n,1›,…,‹c_n,k_n›) AS (
    ‹q_n› )
‹q›
```

Queries $\langle q_j \rangle$ may refer ***to all*** $\langle T_i \rangle$

$\langle q \rangle$ may refer ***to all*** $\langle T_i \rangle$

- In particular, any $\langle q_j \rangle$ may refer to itself (↻)! Mutual references are OK, too. (Think letrec in FP.)

- Typically, final query $\langle q \rangle$ performs post-processing only.

Shape of a Self-Referential Query

WITH RECURSIVE
  $T(c_1,\dots,c_k)$ **AS** (-- common schema of q_0 and $q^{\circlearrowleft}(\cdot)$
    q_0 -- base case query, evaluated once

   UNION [ALL]   -- either **UNION** or **UNION ALL**

    $q_0(T)$ -- recursive query refers to T itself,
)   --   evaluated repeatedly
$q(T)$ -- final post-processing query

- Semantics in a nutshell:

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

Semantics of a Self-Referential Query (UNION Variant)

Iterative and recursive semantics—both are equivalent:

```
iterate(q⟲, q₀):                 recurse(q⟲, r):
  r ← q₀                            if r ≠ ∅ then
  t ← r                             | return r ⊎ recurse(q⟲, q⟲(r) \ r)
  while t ≠ ∅                       else
  | t ← q⟲(t) \ r                   ⌊ return ∅
  ⌊ r ← r ⊎ t
  return r
```

- Invoke the recursive variant with recurse($q^{\circlearrowleft}$, q_0).
- $\uplus$ denotes disjoint set union, \ denotes set difference.
- $q^{\circlearrowleft}(\cdot)$ evaluated over **the *new* rows found in the last iteration/recursive call**. Exit if there were no new rows.

A Home-Made `generate_series()`

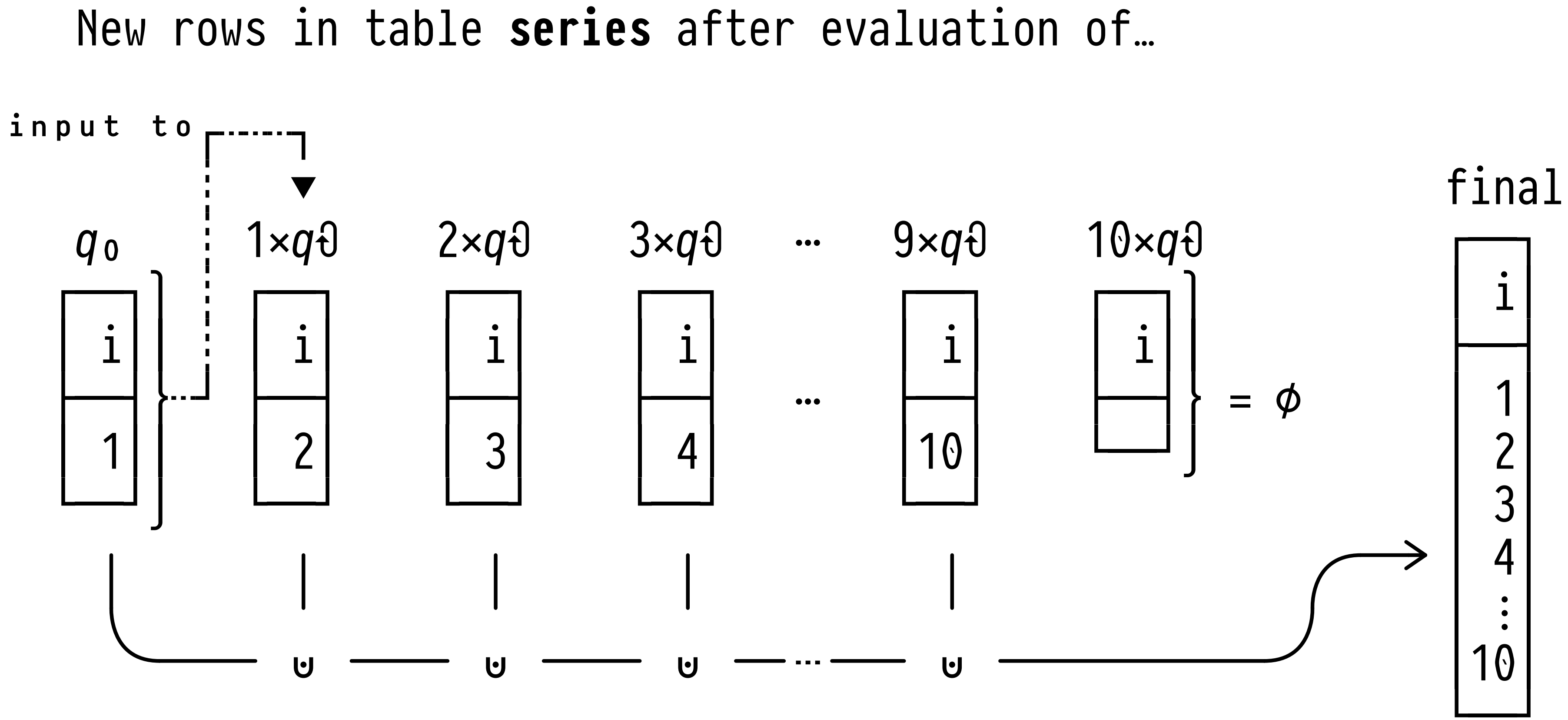
Generate a single-column table of integers i ∈ {‹*from*›,‹*from*›+1,…,‹*to*›}:

```
WITH RECURSIVE
  series(i) AS (
    VALUES (‹from›)             -- q0
       UNION
    SELECT s.i + 1 AS i         -- ⎫
    FROM   series AS s          -- ⎬ q↺(series)        self-
    WHERE  s.i < ‹to›           -- ⎭                   reference
)
TABLE series
```

- **Q:** Given the predicate `s.i < ‹to›`, will ‹*to*› indeed be in the final table?

A Home-Made **generate_series()**

- Assume ‹*from*› = 1, ‹*to*› = 10:



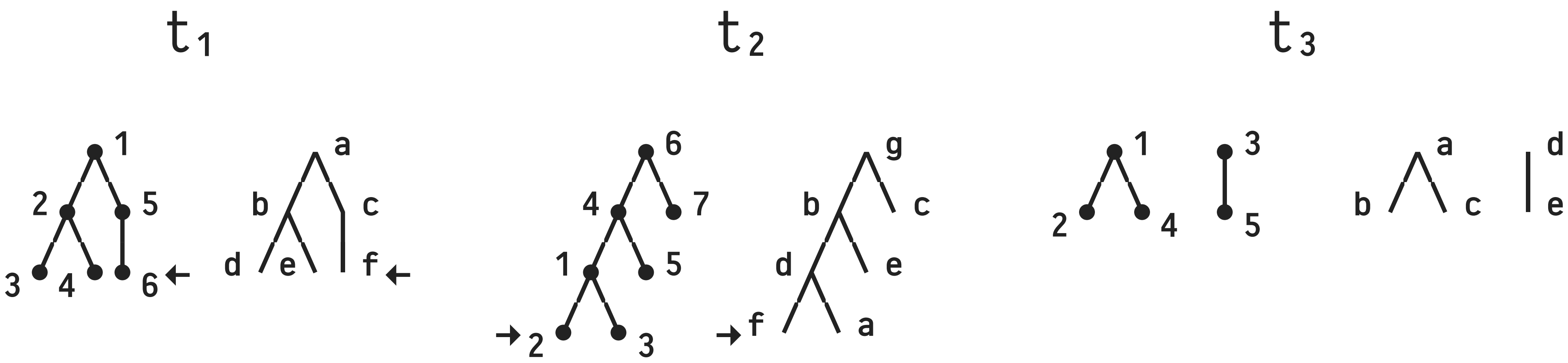
Semantics of a Self-Referential Query (UNION ALL Variant) ✎

With UNION ALL, recursive query $q^{\circlearrowleft}$ sees ***all* rows added in the last iteration/recursive call:**

```
iterate^all(q↺, q₀):          ¦ recurse^all(q↺, r):
  r ← q₀                      ¦   if r ≠ ∅ then
  t ← r                       ¦   | return r ⊎ recurse^all(q↺, q↺(r))
  while t ≠ ∅                 ¦   else
  | t ← q↺(t)                 ¦   ⌊ return ∅
  ⌊ r ← r ⊎ t                 ¦
  return r                    ¦
```

- Invoke the recursive variant via $\mathrm{recurse}^{\mathrm{all}}(q^{\circlearrowleft}, q_0)$.
- $\uplus$ denotes bag (multiset) union.
- Note: Could immediately emit t — no need to build r. 👍

1 ¦ Traverse the Paths from Nodes 'f' to their Root



Array-based tree encoding (parent of node *n* ≡ parents[*n*]):

tree	**parents (□ ≡ NULL)**	**labels**
t_1	{□, 1, 2, 2, 1, 5}	{'a','b','d','e','c','f'}
t_2	{4, 1, 1, 6, 4, □, 6}	{'d','f','a','b','e','g','c'}
t_3	{□, 1, □, 1, 3}	{'a','b','d','c','e'}
	1 2 3 4 5 6 7	1 2 3 4 5 6 7 ←-- node

Trees

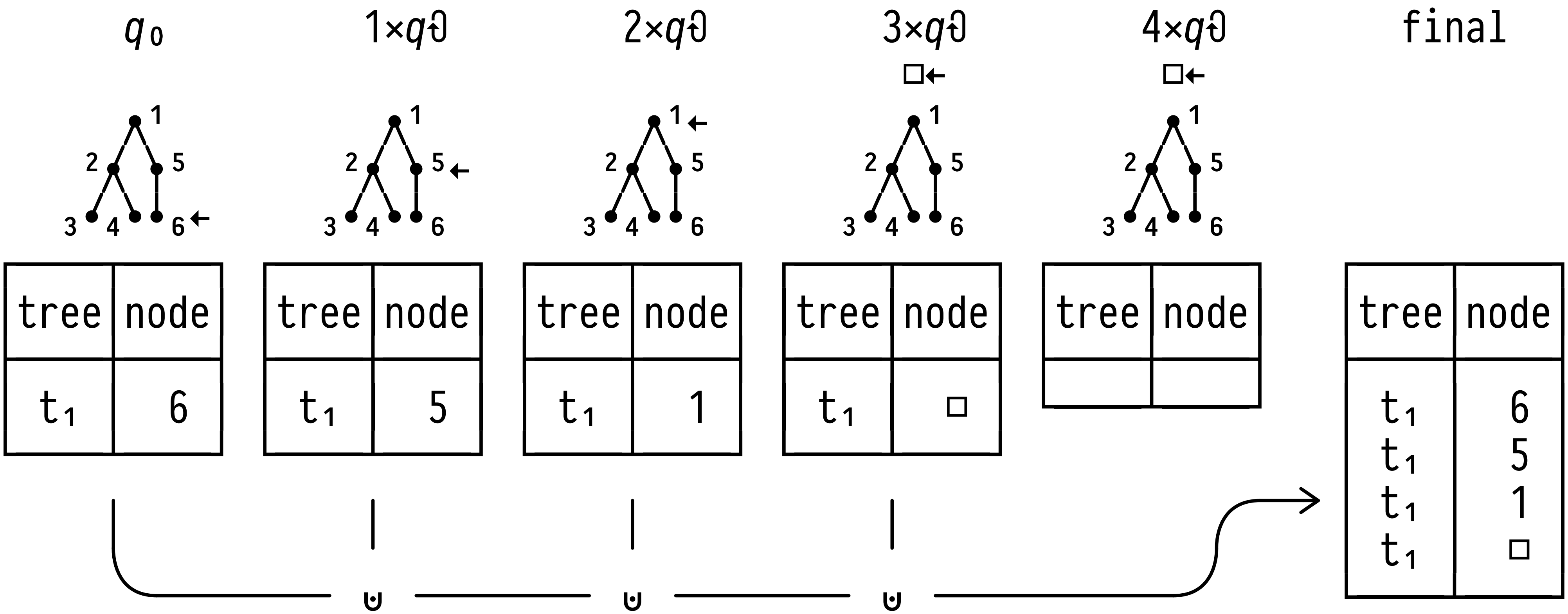
Traverse the Paths from Nodes 'f' to their Root

```
WITH RECURSIVE
  paths(tree, node) AS (
    SELECT t.tree, array_position(t.labels, 'f') AS node
    FROM   Trees AS t

      UNION

    SELECT t.tree, t.parents[p.node] AS node
    FROM   paths AS p,
           Trees AS t
    WHERE  p.tree = t.tree
)
TABLE paths
```

$(t,n) \in \text{paths} \Leftrightarrow$ node n lies on path from 'f' to t's root

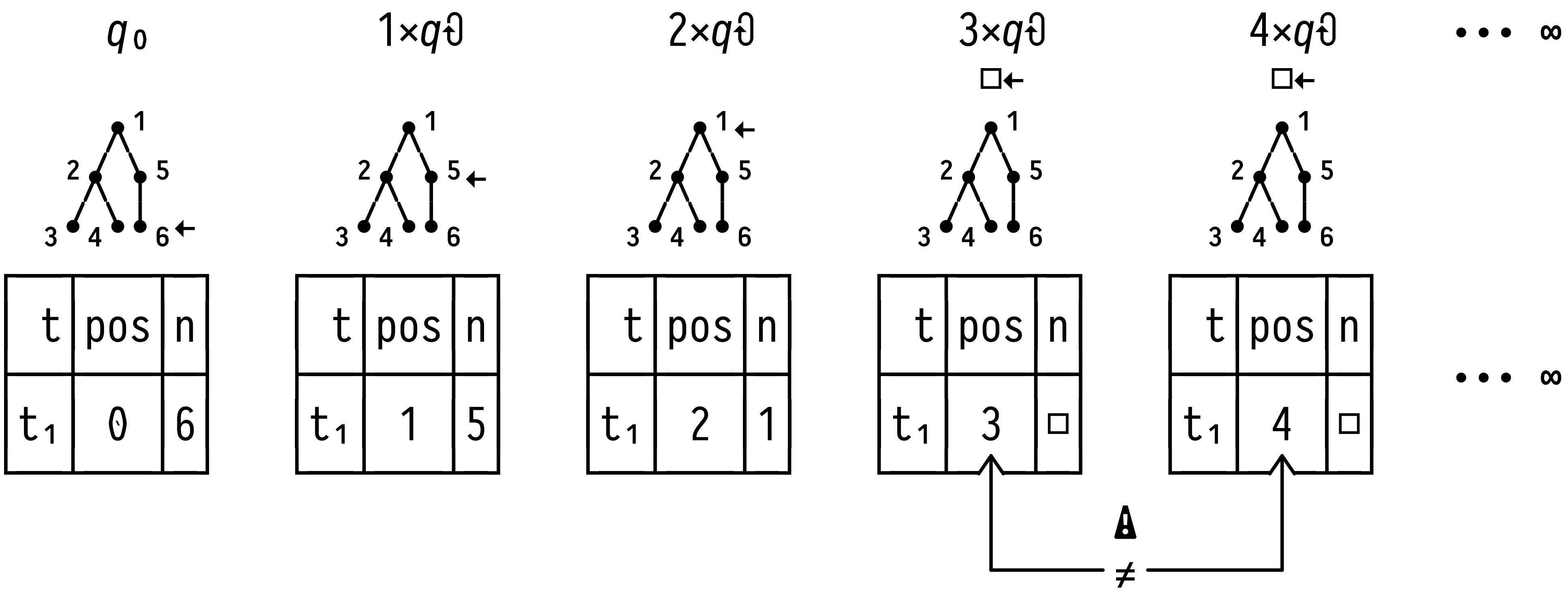
A Trace of the Path in Tree t_1

New rows produced by…



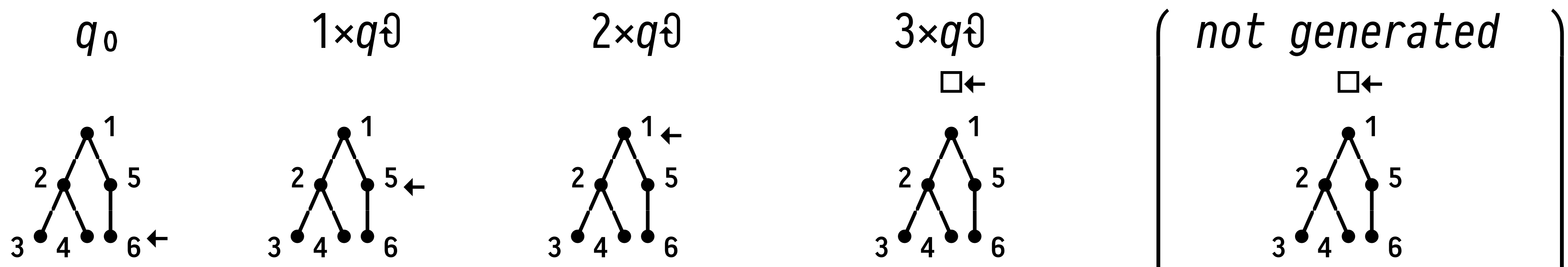
- 4×qϑ yields no new rows (recall: `t.parents[NULL] ≡ NULL`).

Ordered Path in Tree t_1 (New Rows Trace) ✎



The **(non-)generation of new rows to ensure termination** is the user's responsibility — a common source of 🐞.

Path as Array in Tree t_1 (New Rows Trace)



t	n	path
t_1	6	{}

t	n	path
t_1	5	{6}

t	n	path
t_1	1	{6,5}

t	n	path
t_1	□	{6,5,1}

❷

not generated

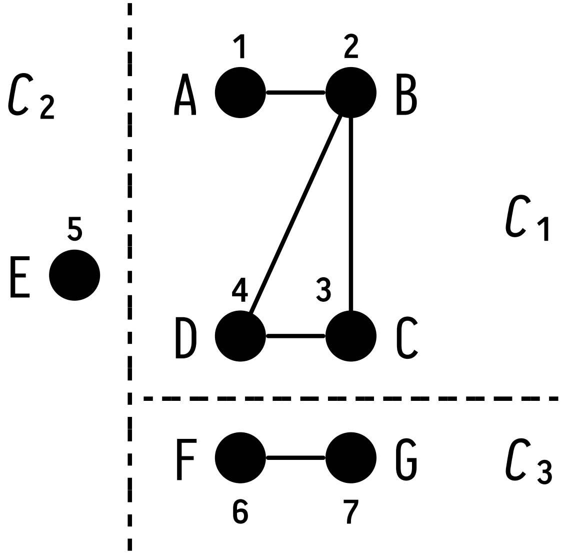
t	n	path
t_1	□	{6,5,1,□}

❶

❶ Ensure termination (enforce ϕ): filter on n ≠ □ in q⟲.

❷ Post-process: keep rows of last iteration (n = □) only.

Connected Components in a Graph



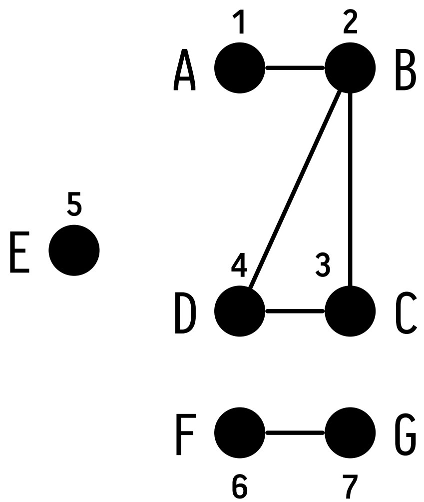
- Given an undirected graph G, find its **connected components** C_i:

  For any two nodes v_1, v_2 in C_i, there exists a path v_1—v_2 (and no connections to outside C_i exist).

- Do we need DBMSs tailored to process graph data and queries?

Graphs are (edge) *relations*. Connected components are the equivalence classes of the reachability *relation* on G.

Representing (Un-)Directed Graphs



nodes

node	label
1	A
2	B
3	C
4	D
5	E
6	F
7	G

edges

from	to
1	2
2	3
3	4
2	4
6	7

A → B

$\xrightarrow{\text{derive}}$

graph

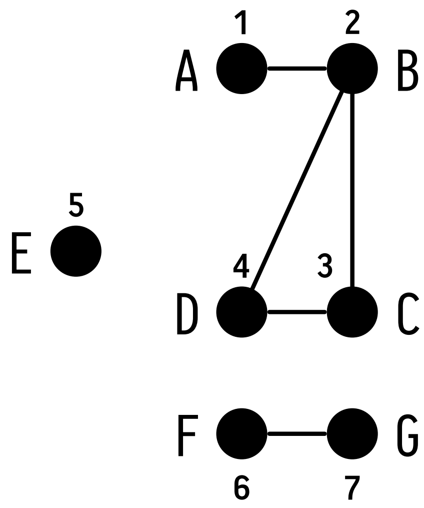
from	to
1	2
2	1
2	3
3	2
2	4
4	2
3	4
4	3
6	7
7	6

A ⇆ B

A — B

- Use tables nodes and graph to formulate the algorithm.

Computing Connected Components (Query Plan)



1. For each node *n*, start a **walk** through the graph. Record each node *f* (“front”) that we can **reach** from *n*.

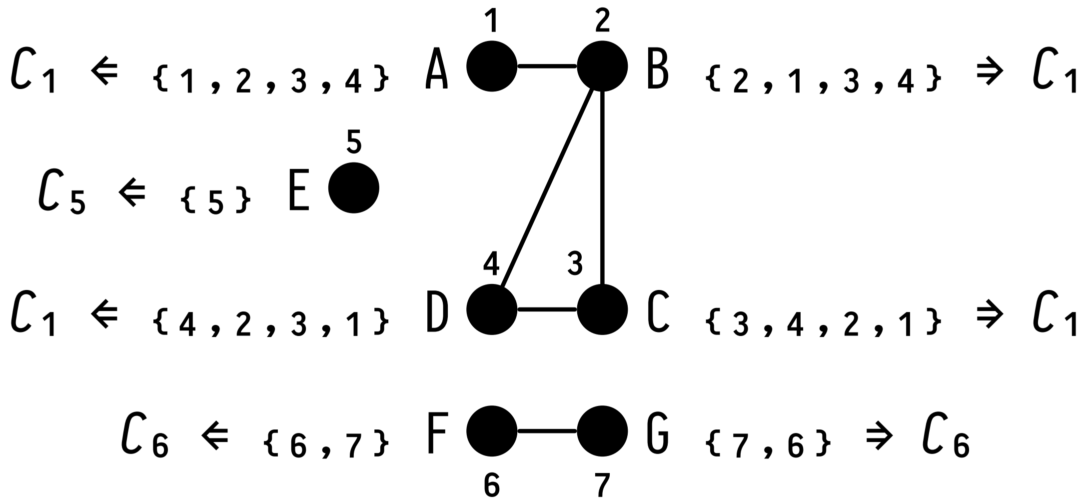
2. For each *n*, use the **minimum ID** i of all front nodes as *n*'s component C_i.

   ⇒ Nodes that can reach each other will use the same component ID.

⚠️ In Step 1, take care to not walk into **endless cycles**.

Computing Connected Components (Query Plan)

- {…}: Reachable front nodes, C_i derived component ID:



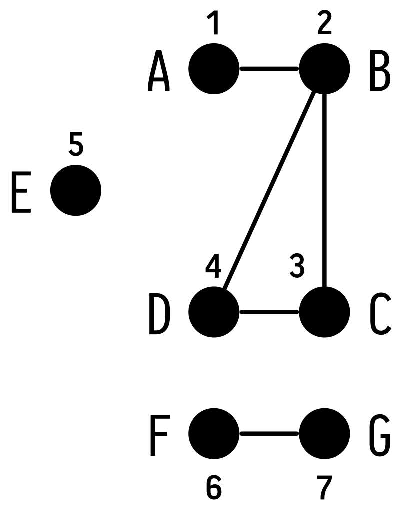
- Tasks for further post-processing:
  - Assign sane component IDs (C_1,C_2,C_3).
  - Extract subgraphs based on components' node sets.

Recursive Graph Walks, From All Nodes at the Same Time

```
WITH RECURSIVE

walks(node, front) AS (
  SELECT n.node, n.node AS front  -- (n,n) ∈ walks: we can
  FROM   nodes AS n               -- reach ourselves

    UNION       -- only new front nodes will be recorded ✓

  SELECT w.node, g."to" AS front  -- record front node
  FROM   walks AS w, graph AS g   -- ⎱ finds all incident
  WHERE  w.front = g."from"       -- ⎰ graph edges
)
```

Invariant: If $(n,f) \in$ walks, node f is reachable from n.

Recursive Graph Walks, From All Nodes at the Same Time



q_0

node	front
1	1
2	2
3	3
4	4
5	5
6	6
7	7

$1\times q\circlearrowleft$

node	front
1	2
2	1
2	3
2	4
3	2
3	4
4	2
4	3
6	7
7	6

$2\times q\circlearrowleft$

node	front
1	3
1	4
3	1
4	1

$3\times q\circlearrowleft$

node	front

- Tree path finding and connected component search used **node adjacency information** to explore graph structure, iteration by iteration.

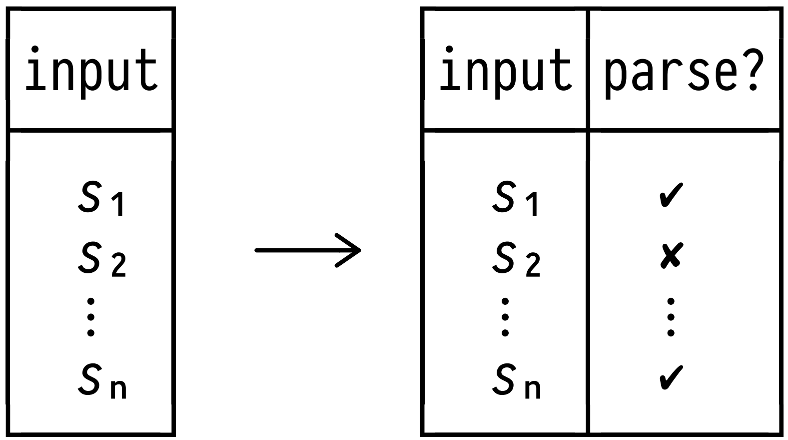
- In a variant of this theme, let us view **text as lists of adjacent characters** that we recursively explore.

- We particularly use the observation (let s :: text, $n \geq 1$):

$$s = \underbrace{\mathrm{left}(s,\ n)}_{\text{prefix of } s \text{ of length } n} \mathbin{||} \underbrace{\mathrm{right}(s,\ -n)}_{\text{all but the first } n \text{ chars of } s}$$

Set-Oriented (Bulk) Regular Expression Matching

Goal: Given a — potentially large — table of input strings, **validate all strings against a regular expression:**[2]



- Plan: Parse all s_i in parallel (run n matchers at once).

[2] We consider parsing given a context-free grammar in the sequel.

Match the **formulæ of chemical compounds** against the regular expression:

$$([A..Za..z]+[_{0}.._{9}]*([^{0}..^{9}]*[^{+-}])?)+$$

compound	formula
citrate	$C_6H_5O_7{}^{3-}$
glucose	$C_6H_{12}O_6$
hydronium	H_3O^+
⋮	⋮

Table compounds

- Generally: support regular expressions *re* of the forms *c*, $[c_1c_2\ldots c_n]$, re_1re_2, $re*$, $re+$, $re?$, $re_1|re_2$.

From Regular Expression to Finite State Machine (FSM)

- Represent *re* in terms of a **deterministic FSM:**

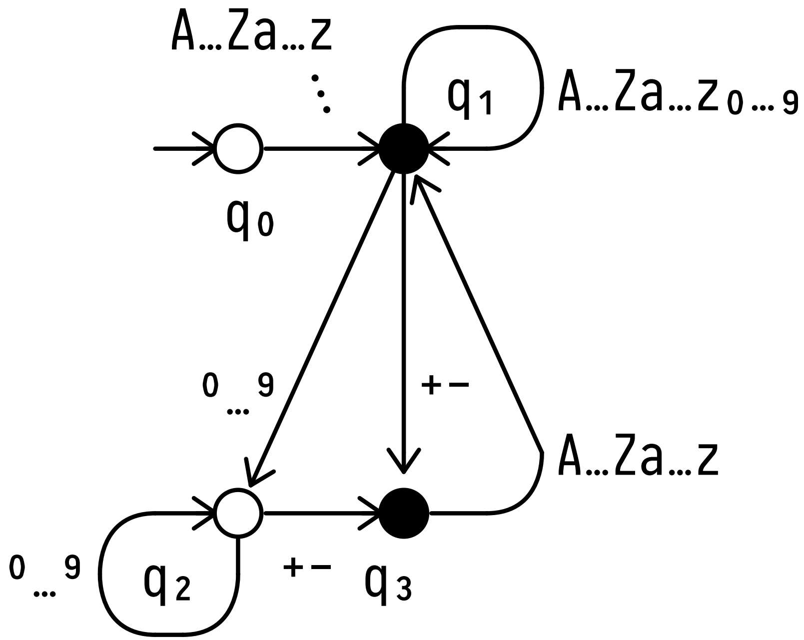


Table **fsm**

source	labels	target	final?
q_0	A…Za…z	q_1	false
q_1	A…Za…z0…9	q_1	true
q_1	0…9	q_2	true
q_1	+-	q_3	true
q_2	0…9	q_2	false
q_2	+-	q_3	false
q_3	A…Za…z	q_1	true

- We tolerate the non-key-FD source→final? for simplicity.

Driving the Finite State Machines (Query Plan)

1. For n entries in table compounds, operate n instances of the FSM “in parallel”:

    - Each FSM instance maintains its current state and the residual input still to match.

2. **Invariant:**

<u>compound</u>	<u>step</u>	state	input
c	s	q	f

Table match

- After $s \geqslant 0$ transitions, FSM for compound c has reached state q. Residual input is f (a suffix of c's formula).

Driving the Finite State Machines (SQL Code)

```
WITH RECURSIVE
match(compound, step, state, input) AS (
 SELECT c.compound, 0 AS step, 0 AS state,
        c.formula AS input  --
 FROM   compounds AS c      --      ≡ q0

   UNION ALL -- ⚠ bag semantics (see below)

 SELECT m.compound, m.step + 1 AS step, f.target AS state,
        right(m.input, -1) AS input
 FROM   match AS m, fsm AS f
 WHERE  length(m.input) > 0
 AND    m.state = f.source
 AND    strpos(f.labels, left(m.input, 1)) > 0
)
```

Matching Progess (by Compound / by Step)

1 Focus on indivdiual compound

compound	step	state	input
citrate	0	0	$C_6H_5O_7^{3-}$
citrate	1	1	${}_6H_5O_7^{3-}$
citrate	2	1	$H_5O_7^{3-}$
citrate	3	1	${}_5O_7^{3-}$
citrate	4	1	O_7^{3-}
citrate	5	1	${}_7^{3-}$
citrate	6	1	${}^{3-}$
citrate	7	2	$-$
citrate	8	3	ε
hydronium	0	0	H_3O^+
hydronium	1	1	${}_3O^+$
hydronium	2	1	O^+
hydronium	3	1	${}^+$
hydronium	4	3	

empty string

final state

=

2 Focus on parallel progress

step	compound	state	input
0	citrate	0	$C_6H_5O_7^{3-}$
0	hydronium	0	H_3O^+
1	citrate	1	${}_6H_5O_7^{3-}$
1	hydronium	1	${}_3O^+$
2	citrate	1	$H_5O_7^{3-}$
2	hydronium	1	O^+
3	citrate	1	${}_5O_7^{3-}$
3	hydronium	1	${}^+$
4	citrate	1	O_7^{3-}
4	hydronium	3	ε
5	citrate	1	${}_7^{3-}$
6	citrate	1	${}^{3-}$
7	citrate	2	$-$
8	citrate	3	ε

Termination and Bag Semantics (UNION ALL)

The recursive CTE in regular expression matching uses **bag semantics** (UNION ALL). Will matching always **terminate**?

- Column `step` is increased in each iteration, thus...
  1. $q_{\circlearrowleft}$ **will never produce duplicate rows** and
  2. there is no point in computing the difference $q_{\circlearrowleft}(t) \setminus r$ in $\mathrm{iterate}(q_{\circlearrowleft}, q_0)$: $q_{\circlearrowleft}(t) \cap r = \phi$.

- $q_{\circlearrowleft}$ **is guaranteed to evaluate to ∅ at one point**, since...
  1. one character is chopped off in each iteration and `length(m.input) > 0` will yield *false* eventually, or
  2. the FSM gets stuck due to an invalid input character (`strpos(f.labels, left(m.input, 1))` yields 0).

4 ⁝ Recursive Array Processing: Solving Sudoku[3] Puzzles

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Fill in the blanks with digits $\in \{1,\ldots,9\}$ such that

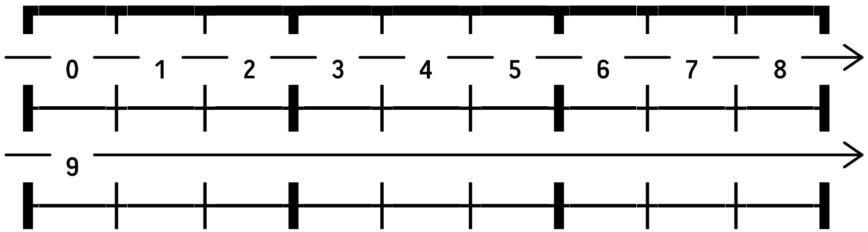
  1. no 3×3 square and
  2. no row or column

  carries the same digit twice.

- Here: encode board as digit array.

[3] Japanese: *sū(ji)* + *doku(shin)*, “number with single status.”

Row-Major Array-Encoding of a 2D Grid



- Build row-wise int[] array of 81 cells ∈ {0,…,9}, with 0 ≡ blank.

- Derive **row/column/square index** from cell *c* ∈ {0,…,80}:
  - Row of *c*: (*c* / 9) * 9 ∈ {0,9,18,27,36,45,54,63,72}
  - Column of *c*: *c* % 9 ∈ {0,1,2,3,4,5,6,7,8}
  - Square of *c*: ((*c* / 3) % 3) * 3 + (*c* / 27) * 27 ∈ {0,3,6,27,30,33,54,57,60}

- (Clunky — But: relational encodings of grids upcoming.)

Finding All Puzzle Solutions (Query Plan)

board	blank
{5,3,0,0,7,...}	$b \in \{0,\ldots,80\} \cup \{\square\}$

Table sudoku

1. **Invariant:**

   ◦ Column board encodes a valid (but partial) Sudoku board in which the first blank (≡ 0) occurs at index b. If the board is complete, $b = \square$.

2. In each iteration, **fill in all digits** $\in \{1,\ldots,9\}$ at b and **keep all boards that turn out valid.**

Finding All Puzzle Solutions (SQL Code)

```
WITH RECURSIVE
sudoku(board, blank) AS (
 SELECT i.board AS board, array_position(i.board, 0)-1 AS blank
 FROM   input AS i                           --      ↑
                                             -- encodes blank
   UNION ALL

 SELECT   s.board[1:s.b] || fill_in || s.board[s.b+2:81] AS board,
        array_position(
          s.board[1:s.b] || fill_in || s.board[s.b+2:81], 0)-1 AS blank
 FROM  sudoku AS s(board, b), generate_series(1,9) AS fill_in
                              --
                              --   try to fill in all 9 digits
 WHERE s.b IS NOT NULL AND NOT EXISTS (
   SELECT NULL
   FROM   generate_series(1,9) AS i
   --
   --    9 cells in row/column/square
   WHERE  fill_in IN (‹digits in row/column/square of s.b at offset i›))
)
```

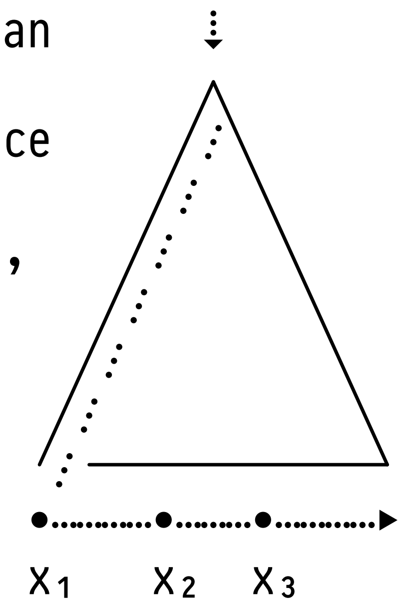
5 ¦ Emulating Physical Operator Behavior: Loose Index Scans

Implement `SELECT DISTINCT t.dup FROM t` efficiently, given

- column `dup` contains a sizable number of **duplicates**, and
- **B-tree index** support on column `dup`.

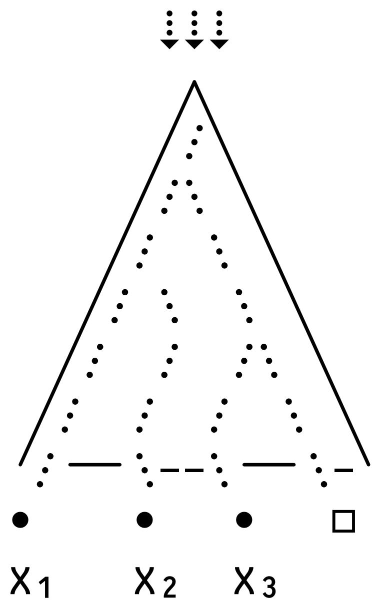
1 Regular Index Scan

- Enter B-tree once
- Scan leaf level, skipping over duplicates
- Implemented by PostgreSQL



2 Loose Index Scan

- Re-enter B-tree from root
- Search for next larger x_i only
- Not implemented by PostgreSQL



Emulating Physical Operator Behavior: Loose Index Scans

```
WITH RECURSIVE
loose(xᵢ) AS (
  SELECT MIN(t.dup) AS xᵢ       -- ⎫ find smallest value x₁
  FROM   t                      -- ⎭ in column dup

    UNION ALL

  SELECT (SELECT MIN(t.dup)           -- ⎫ find next larger
          FROM   t                    -- ⎬ value xᵢ (≡ NULL
          WHERE  t.dup > l.xᵢ) AS xᵢ  -- ⎭ if no such value)
  FROM   loose AS l
  WHERE  l.xᵢ IS NOT NULL           -- last search successful?
)
SELECT l.xᵢ
FROM   loose AS l
WHERE  l.xᵢ IS NOT NULL
```

Loose Index Scans: Does Recursion Pay Off?

Micro benchmark: |t| = 10^6 rows, number of duplicates in column dup :: int varies:[4]

# of distinct values in dup	index scan [ms]	loose index scan [ms]
10	428	< 1 ⚠
100	440	2 ⚠
1000	442	31
10000	454	194
100000	672	1778

Performance comparison

- Recursion beats the built-in index scan if the number of B-tree root-to-leaf traversals is not excessive.

[4] PostgreSQL 9.6 on macOS Sierra (10.12.5), 3.3GHz Intel Core i7, 16GB RAM @ 2133 MHz, SATA SSD. Each query run multiple times, average reported here.

6 ┆ How SQL Can Tackle Problems in Machine Learning[5]

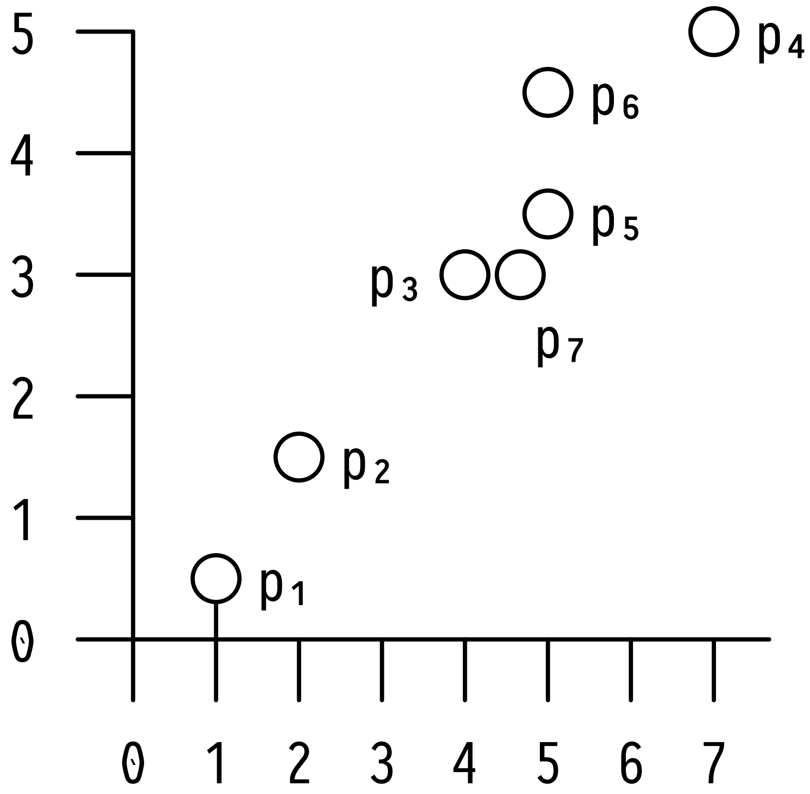
Most sizable *source data* for Machine Learning (ML) problems reside **inside** database systems. Actual *ML algorithms* are predominantly implemented **outside** the DBMS — Python, R, MatLab — however:

- Involves data serialization, transfer, and parsing. 👎
- The main-memory based ML libraries and programming frameworks are challenged by the data volume. 👎

Demonstrate how ML algorithms (here: **K-Means** clustering) may be expressed in SQL and thus executed close to the data.

[5] I apologize for the hype vocabulary.

K-Means Clustering

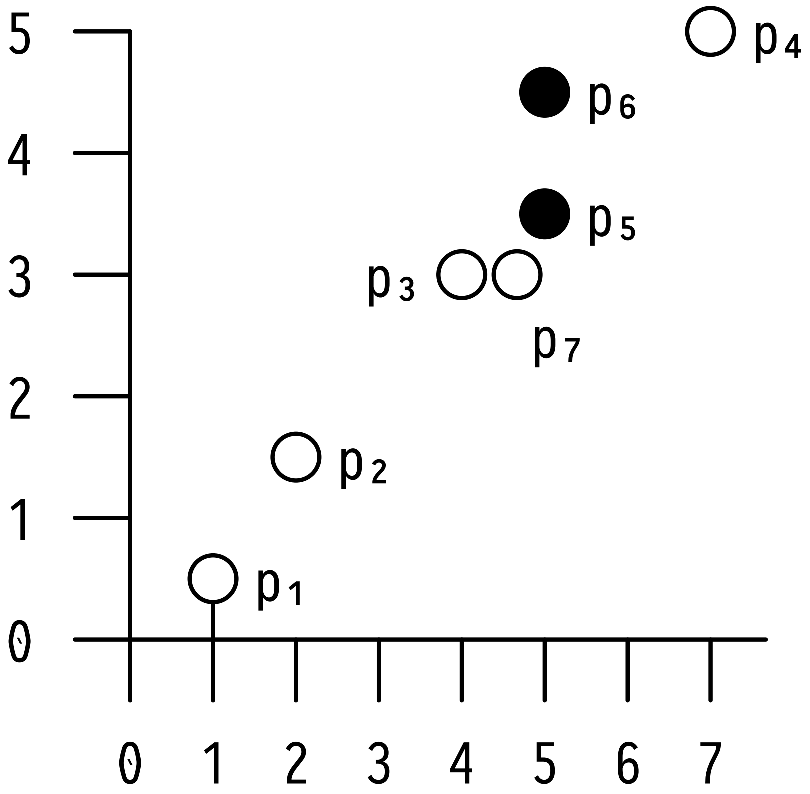


- **Goal:** Assign each *n*-dimensional point p_i to one of ***k* clusters** (*k* given).

- Once done, each p_i shall belong to the cluster with the nearest **mean** (a point that serves as "the prototype of the cluster").

K-Means is computationally difficult (NP-hard) but good approximations/heuristics exist.

K-Means: Lloyd's Algorithm with Forgy Initialization



- Pick k random points (here: p_5, p_6 for $k = 2$) as initial means.

1. **Assignment:**
   Assign each p_i to nearest mean.

2. **Update:**
   Determine k new means to be the **centroids** of the points assigned to each cluster.

Iterate 1. + 2. until assignments **no longer change.**

K-Means: Forgy Initialization (Query Plan)

point	loc
1	point(1.0, 1.0)
2	point(2.0, 1.5)
⋮	⋮

Table points

- Picking random rows in table ‹T›:

```
TABLE ‹T›
ORDER BY random()
LIMIT k
```

```
SELECT t.*
FROM ‹T› AS t
TABLESAMPLE BERNOULLI(s)  -- pick ≈ s% random rows in ‹T›
```

K-Means: Lloyd's Algorithm (Query Plan)

Invariant:

iter	point	cluster	mean
i	p	c	m

Table k_means

- In iteration i, point p has been assigned to cluster c. The mean of cluster c is at location m :: point.
  - After iteration 0 (initialization), k_means will have k rows; later on we have |k_means| = |points|.
- Again: we tolerate the embedded FD cluster → mean.

🔧 K-Means: Core of the SQL Code ✎

```sql
WITH RECURSIVE
 k_means(iter, point, cluster, mean) AS (
 ⋮
 -- 2. Update
 SELECT assign.iter+1 AS iter, assign.point, assign.cluster,
        point(AVG(assign.loc[0]) OVER cluster,
              AVG(assign.loc[1]) OVER cluster) AS mean
        -- 1. Assignment
 FROM   (SELECT DISTINCT ON (p.point)
                    k.iter, p.point, k.cluster, p.loc
         FROM   points AS p, k_means AS k
         ORDER BY p.point, p.loc <-> k.mean) AS assign
 WHERE  assign.iter < ‹iterations›
 WINDOW cluster AS (PARTITION BY assign.cluster)
)
```

SQL Notes and Grievance (1)

- We first deconstruct and later reconstruct the points for centroid computation:

```
    point(AVG(assign.loc[0]) OVER cluster,
          AVG(assign.loc[1]) OVER cluster) AS mean
```

- Wanted: aggregate AVG() :: bag(point) → point.

  💡 In PostgreSQL, we can build **user-defined aggregates.**[6]

[6] See CREATE AGGREGATE at https://www.postgresql.org/docs/9.6/static/xaggr.html.

SQL Notes and Grievance (2)

- K-Means is the prototype of an algorithm that searches for a **fixpoint**. Still, we were using UNION ALL semantics and manually maintain column iter 😨. Why?

  - There is **no equality operator** = :: point × point → bool in PostgreSQL, a requirement to implement set semantics and \ (recall functions iterate(•,•) and recurse(•,•)).
    💡 **User-defined equality** or split point (•[0],•[1]).

  - Without column iter, we cannot identify the resulting cluster assignment found in the final iteration.
    💡 Use the **"flip trick."** ✎

- Is the subquery (*Assignment*) in the recursive query q⇡ of Lloyd's algorithm the nicest solution? Can't we write:

```
  ⋮
SELECT …, (SELECT k.cluster
           FROM   k_means AS k  -- ◀ invalid placement
           ORDER BY p.loc <-> k.means
           LIMIT 1) AS cluster, …
FROM   points AS p
  ⋮
```

- **A:** No. References to *recursive table* k_means inside a subquery in the SELECT or WHERE clause are forbidden. ⚠️

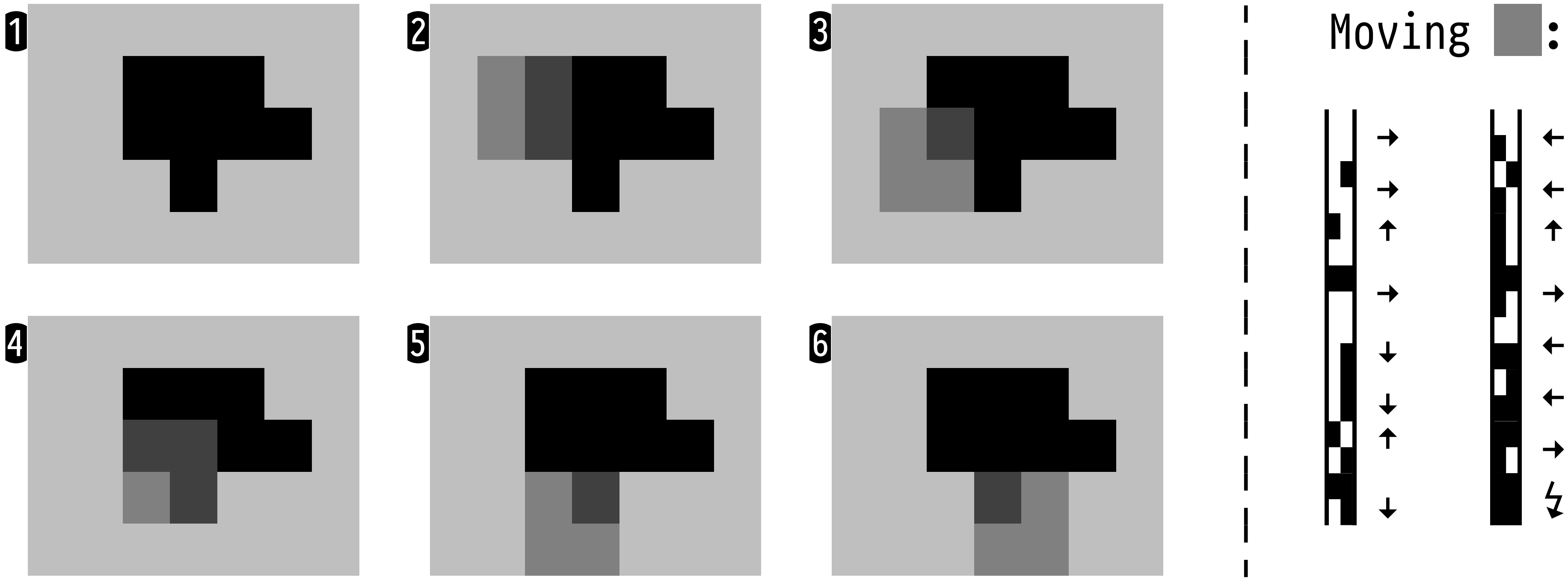
SQL provides a family of constructs to encode the **logic** (in the sense of **control flow**) of algorithms:

1. Obviously: `WHERE <p>`, `HAVING <p>`,
2. `<q₁> UNION ALL <q₂> UNION ALL …`
   in which the `<qᵢ>` contain guards (predicates) that control their contribution,
3. `CASE <p> WHEN … THEN … ELSE … END`.

SQL being a data-oriented language additionally suggests the option to **turn control flow into data flow. Encoding query logic in tables** can lead to compact, self-describing, and extensible query variants.

Find Isobaric or Contour Lines: Marching Squares

Goal: Trace the boundary of the object in ❶ (█ ≡ object):



- **15 cases** define the movement of the 2×2 pixel mask.

🔧 Marching Squares (Query Plan)

1. **Encode mask movement** in table directions that maps 2×2 pixel patterns to $(\Delta x, \Delta y) \in \{-1,0,1\} \times \{-1,0,1\}$.
   Examples: ▛ maps to (1,0), ▘| maps to (0,-1).

2. For each 2D-pixel p_0, read pixels at $p_0+(1,0)$, $p_0+(0,1)$, $p_0+(1,1)$, to form a 2×2 squares map [table squares].

3. Iteratively fill table march(x,y):
   - [q_0]: Start with $(1,1) \in$ march.
   - [$q_{\circlearrowleft}$]: Find 2×2 pixel pattern at (x,y) in squares, lookup pattern in directions to move mask to $(x,y) + (\Delta x, \Delta y)$.

Marching Squares (SQL Code) ✎

```
WITH RECURSIVE
⋮
march(x,y) AS (
 SELECT 1 AS x, 1 AS y
    UNION
 SELECT new.x AS x, new.y AS y
 FROM   march AS m, squares AS s,
        directions AS d,                                          ⎫
        LATERAL (VALUES (m.x + (d.dir).Δx,                        ⎬ ⁑
                         m.y + (d.dir).Δy)) AS new(x,y)           ⎪
 WHERE  (s.ll,s.lr,s.ul,s.ur) = (d.ll,d.lr,d.ul,d.ur)             ⎭
 AND    (m.x,m.y) = (s.x,s.y)
)
```

⁑ Table lookup replaces a 15-fold case distinction. 👍

Cellular automata (CA)[7] are discrete state-transition systems that can model a variety of phenomena in physics, biology, chemistry, maths, or the social sciences:

- **Cells** populate a regular *n*-dimensional **grid**, each cell being in one of a finite number of **states**.
- A cell can interact with the cells of its **neighborhood**.
- State of cell *c* changes from **generation to generation** by a fixed set of **rules**, dependent on *c*'s state and those of its neighbors.

[7] Discovered by Stanislaw Ulam and John von Neumann in the 1940s at Los Alamos National Laboratory.

Cell State Change in Cellular Automata

Here, we will distinguish *two flavors* of CA:

❶ Cell *c* is **influenced by** its neighborhood (*c*'s next state is a function of the cell states in the neighborhood)

[Conway's *Game of Life*]

❷ Cell *c* **influences** cells in its neighborhood (*c* contributes to state changes to be made in the neighborhood)

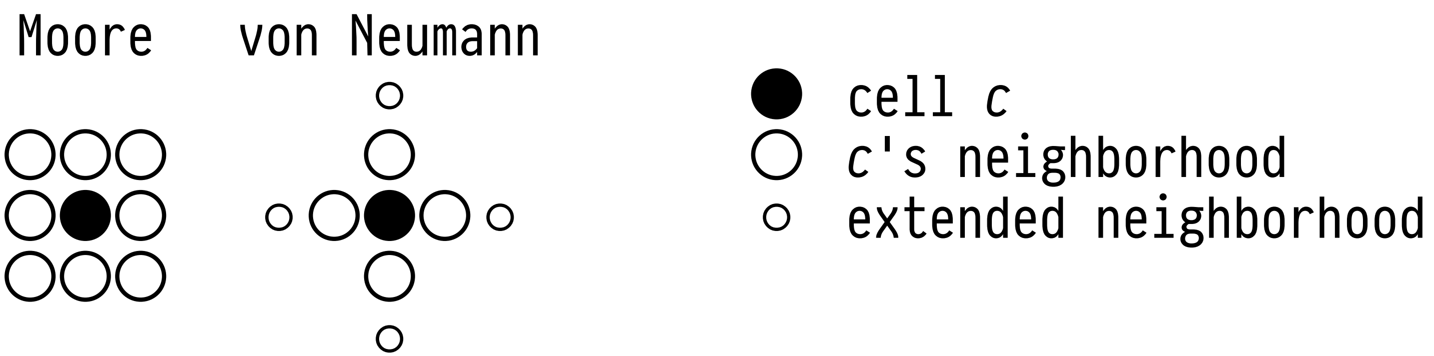
[Fluid simulation]

Both flavors lead to quite different SQL implementations. ❶ is (almost) straightforward, ❷ is more involved. Let us discuss both.

Cell Neighborhood

Cell **neighborhood** is flexibly defined, typically referring to (a subset of) a cell's *adjacent* cells:

- Types of neighborhoods, for n = 2 (2D grid):



x	y	cell
x	*y*	*cell state*

Table grid

Accessing the Cell Neighborhood — Non-Solution! 😦

- Excerpt of code in *q*🔗 (computes next generation of grid), access the Moore neighbors *n* of cell *c*:

```
WITH RECURSIVE
ca(x,y,cell) AS (
  ⋮
  SELECT c.x, c.y, f(c.cell, agg(n.cell)) AS cell
  FROM   ca AS c, ca AS n -- ◀ ⚠️ two references to ca
  WHERE  (c.x - n.x)^2 + (c.y - n.y)^2 <= 2
  GROUP BY c.x, c.y, c.cell
  ⋮
)
```

- Looks like a suitable CA core (*f*, *agg* encode CA rules).
- **BUT** refers to recursive table *more than once*: ↯ **in SQL.**

Interlude: WITH RECURSIVE — Syntactic Restrictions

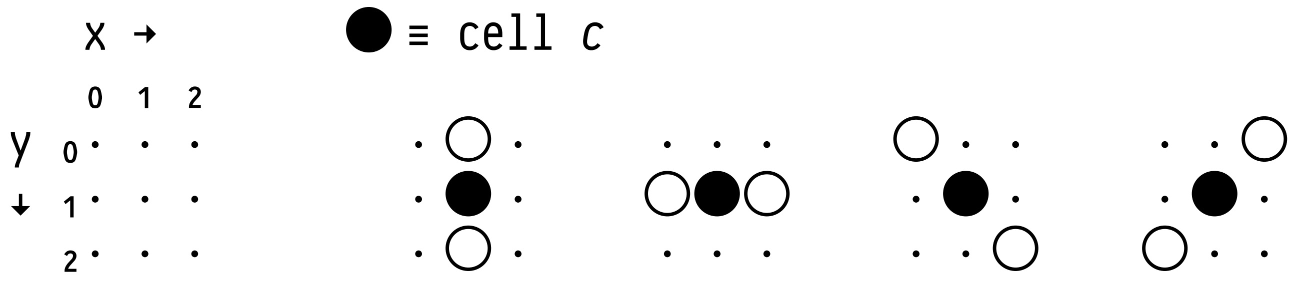
WITH RECURSIVE **syntactically restricts** query forms, in particular the **references to the recursive table** T:

1. No references to T in q_0.
2. A single reference to T in $q_{\circlearrowleft}$ only (**linear recursion**).
3. No reference to T in subqueries outside the FROM clause.
4. No reference to T in INTERSECT or EXCEPT.
5. No reference to T in the null-able side of an outer join.
6. No aggregate functions in $q_{\circlearrowleft}$ (window functions *do* work).
7. No ORDER BY, OFFSET, or LIMIT in $q_{\circlearrowleft}$.

Enforces **distributivity:** $q_{\circlearrowleft}(T \cup \{t\}) = q_{\circlearrowleft}(T) \cup q_{\circlearrowleft}(\{t\})$, allowing for incremental evaluation of WITH RECURSIVE.

Accessing the Cell Neighborhood — A Solution! 😃

💡 **Window functions** admit access to rows in **cell vicinity:**



PARTITION BY	*c*.x	*c*.y	*c*.x−*c*.y	*c*.x+*c*.y
ORDER BY	*c*.y	*c*.x	*c*.x+*c*.y	*c*.x−*c*.y

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

```
SELECT … f(c.cell, agg(c.cell) OVER (<frame>)) …
FROM   ca AS c(x,y,cell)
```

Conway's Game of Life

Life[8] simulates the evolution of cells c (state: either *alive* or *dead*) based on the population count $0 \leq p \leq 8$ of c's Moore neighborhood:

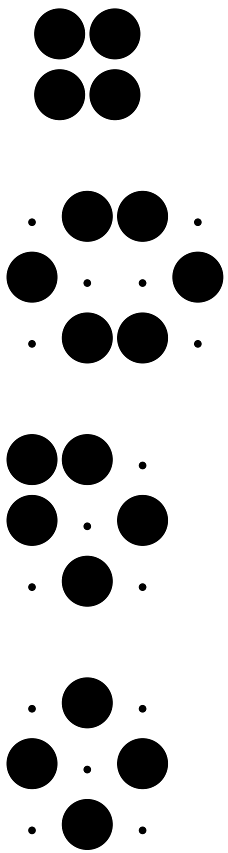
1. If c is alive and $p < 2$, c dies (underpoulation).
2. If c is alive and $2 \leq p \leq 3$, c lives on.
3. If c is alive and $3 < p$, c dies (overpopulation).
4. If c is dead and $p = 3$, c comes alive (reproduction).

Note: The next state of c is a function of the neighborhood states. c does *not* alter cell states in its neighborhood.

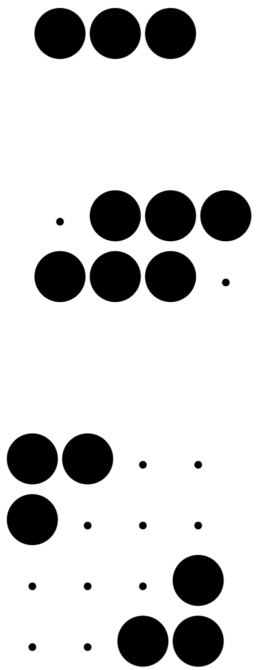
[8] John Horton Conway, column *Mathematical Games* in *Scientific American* (October 1970).

Life — A Few Notable Cell Patterns

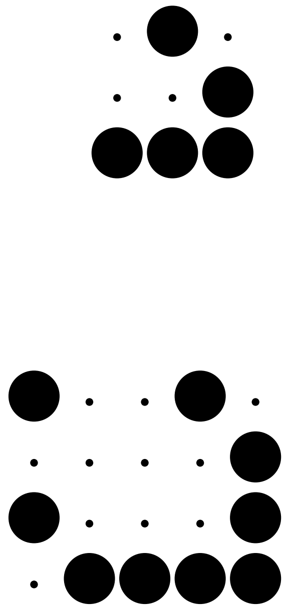
Still



Oscillators (period: 2)



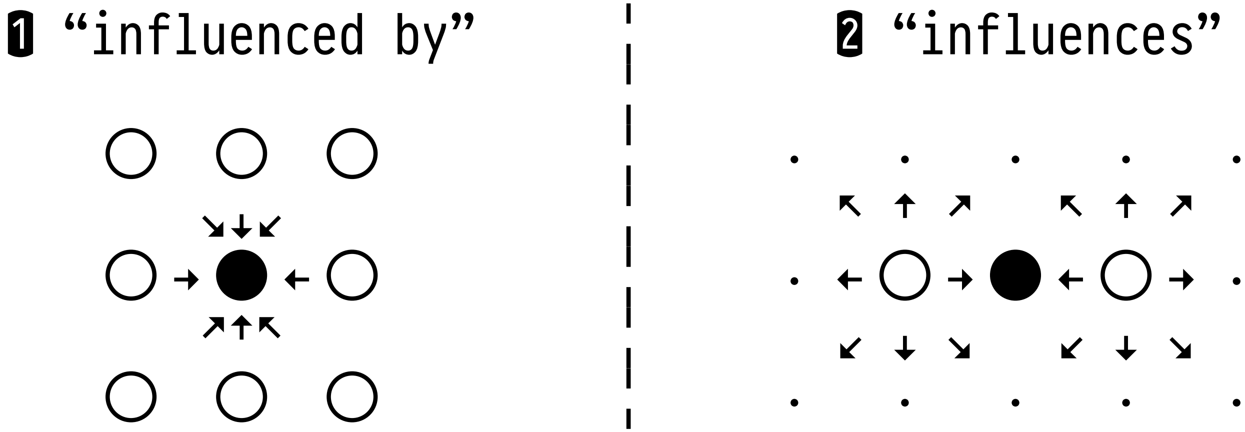
Spaceships



Life — SQL Encoding of Rules (*f*: below, *agg* ≡ SUM) ✎

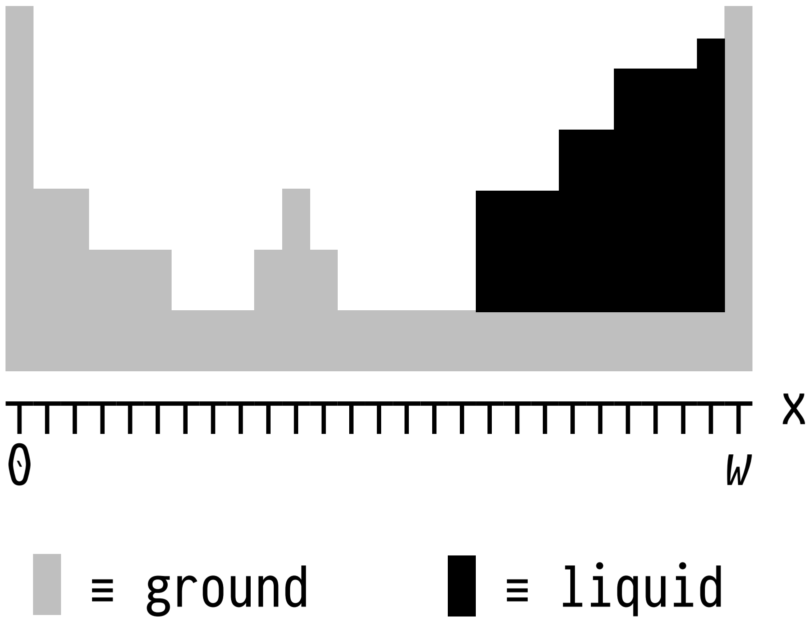
```
WITH RECURSIVE
life(gen,x,y,cell) AS (
 ⋮
 SELECT l.gen + 1 AS gen, l.x, l.y,
        CASE (l.cell, (  SUM(l.cell) OVER (‹horizontal ⋯›)
                       + SUM(l.cell) OVER (‹vertical ⋮›)
                       + SUM(l.cell) OVER (‹diagonal ⋱›)
                       + SUM(l.cell) OVER (‹diagonal ⋰›)
                       - 4 * l.cell)
             )
          --   (c, p): c ≡ state of cell, p ≡ # of live neighbors
          WHEN (1, 2) THEN 1 --
          WHEN (1, 3) THEN 1 --  } alive
          WHEN (0, 3) THEN 1 --
          ELSE             0 -- dead
        END AS cell
 FROM life AS l
 ⋮
)
```

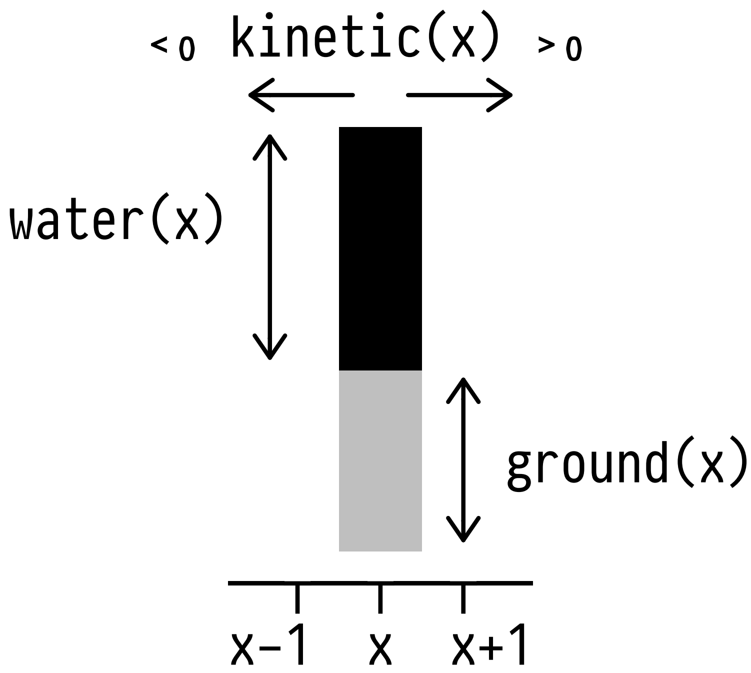
If cells assume an **active role** in influencing the next generation, this suggests a different SQL implementation.



- In type ❷, cells ○ actively influence their neighbors. Affected cells ● need to **accumulate** these individual influences (up to 8 in this grid — only two shown here).

🔧 Simulate the Flow of Liquid (in a 1D Landscape)





Goal: Model two forms of energy in this system:

- **potential energy** at x (**pot**(x) ≡ ground(x) + water(x))
- left/right **kinetic energy** at x (**kinetic**(x))

Liquid Flow: Cellular Automaton[9]

```
Δwater ← (0,0,…,0)  -- changes to water and energy levels
Δkin   ← (0,0,…,0)  -- in next generation
for x in 1…w-1:
│ -- liquid flow to the left
│ if pot(x)-kin(x) > pot(x-1)+kin(x-1):
│ │ flow ← ¼ × min(water(x), pot(x)-kin(x)-(pot(x-1)+kin(x-1)))
│ │ Δwater(x-1) ← Δwater(x-1)+flow                  -- ⎫ aggregate the
│ │ Δwater(x)   ← Δwater(x)  -flow                  -- ⎬ influences on
│ └ Δkin(x-1)   ← Δkin(x-1) - ½×kin(x-1) - flow     -- ⎭ cells @ x / x-1
│ -- liquid flow to the right
│ if pot(x)+kin(x) > pot(x+1)-kin(x+1):
│ ┆  -- "mirror" the above code
└
water ← water + Δwater          -- ⎫ apply the aggregated influences
kin   ← kin   + Δkin            -- ⎭ to all cells (ground is constant)
```

[9] CA rules adapted from those posted by user *YankeeMinstrel* on the *Cellular Automata*. $\frac{1}{4}$, $\frac{1}{2}$ are (arbitrary) dampening/friction factors. See https://www.reddit.com/r/cellular_automata/.

CA with Neighborhood Influencing Rules: SQL Template

```
WITH RECURSIVE
cells(iter,x,y,state) AS (
 ⋮
 SELECT c0.iter + 1 AS iter, c0.x, c0.y,
        c0.state ⊕ COALESCE(agg.Δstate, ‹z›) AS state
 FROM   cells AS c0 LEFT OUTER JOIN
        -- find and aggregate influences on all cells @ x,y
        (■                                   -- ⎫ ■ encodes rules
         ■) AS agg(x,y,Δstate)               -- ⎭   of the CA
        -- extract the influences on cell c0 (□ if none)
        ON (c0.x, c0.y) = (agg.x, agg.y)
 WHERE  c0.iter < ‹iterations›
)
```

- Design: no agg(*x*,*y*,_) if cell @ *x*,*y* doesn't change state.
- Assume that ‹*z*› is neutral element for ⊕: *s* ⊕ ‹*z*› = *s*.

CA: From Individual to Aggregated Influences (SQL Template)

```
⋮
SELECT c0.iter + 1 AS iter, c0.x, c0.y,
       c0.state ⊕ COALESCE(agg.Δstate, ‹z›) AS state
FROM   cells AS c0 LEFT OUTER JOIN
       -- find and aggregate influences on all cells @ x,y
       (SELECT infs.x, infs.y, ‹agg›(infs.Δstate) AS Δstate
        FROM    ([redacted]) AS infs(x,y,Δstate)
        GROUP BY infs.x, infs.y
       ) AS agg(x,y,Δstate)
       -- extract the influences on cell c0 (□ if none)
       ON (c0.x, c0.y) = (agg.x, agg.y)
⋮
```

- (*x*,*y*,Δstate) ∈ infs: individual influence on cell @ *x*,*y*.
- Typically, we will have ‹*agg*› = (ϕ, ‹*z*›, ⊕).

CA: Individual Neighborhood Influences (SQL Template)

```
⋮
-- find and aggregate influences on all cells @ x,y
(SELECT infs.x, infs.y, ‹agg›(infs.Δstate) AS Δstate
 FROM   (SELECT [redacted]    -- ⎱ all influences that c1 has on
                              -- ⎰ its neighborhood (≡ CA rules)
         FROM   cells AS c1) AS inf(influence),
         LATERAL unnest(inf.influence) AS infs(x,y,Δstate)
 GROUP BY infs.x, infs.y
) AS agg(x,y,Δstate)
⋮
```

- For each cell c1, [redacted] computes an **array of influence** influence with elements $(x,y,\Delta state)$: c1 changes the state of cell @ x,y by Δstate.
- For each c1, influence may have 0, 1, or more elements.

CA: Encoding Neighborhood Influencing Rules (SQL Template)

```
 ⋮
(SELECT (CASE WHEN ‹p1› THEN     -- if ‹p1› holds, then c1 has ...
           array[ROW(c1.x-1, c1.y, ▒▒▒),    -- leftward influence
                 ROW(c1.x, c1.y+1, ▓▓▓)]    -- downward influence
         END
      || CASE WHEN ‹p2› THEN
           array[ROW(c1.x, c1.y, ███)] -- influence on c1 itself
         END      --   ↑     ↑    ↑
          ⋮       --   x     y  Δstate
        ) AS influence
 FROM   cells AS c1
 WINDOW horizontal AS …  -- ⎱ provide frames to access neighbors
 WINDOW vertical   AS …  -- ⎰ of c1 in ‹pi›, ▒▒▒, ▓▓▓, and ███
) AS inf(influence)
 ⋮
```

- Admits straightforward transcription of rules into SQL.

CA: Summary of Influence Data Flow (Example)

- Assume Δstate :: int, *<agg>* ≡ SUM (i.e., *<z>* ≡ 0, ⊕ ≡ +):

❶ Table **inf**

influence
{(1,3,+4),(1,4,-2)}
{(1,3,-3),(1,3,+1)}
{(2,2,-5)}
{(1,4,+2)}

neighborhood influence, computed based on current cell generation

❷ Table **infs**

x	y	Δstate
1	3	+4
1	3	-3
1	3	+1
1	4	-2
1	4	+2
2	2	-5

❸ Table **aggs**

x	y	Δstate
1	3	+2
1	4	0
2	2	-5

apply to current cell states using ⊕ to find next generation

Working Around the Linear Recursion Restriction

Once we unfold the black boxes: the CA SQL template reads table cells *twice*, leading to **non-linear** recursion. ϟ

- Work around[10] linearity restriction for recursive table *T*:
  ❶ read rows of *T* once to form an **array of rows**,
  ❷ unnest() this array as often as needed.

```
SELECT …
FROM (SELECT DISTINCT array_agg(row) OVER () AS T    -- 1
      FROM T AS row) AS T̄,
    … LATERAL unnest(T̄.T) AS t1 …                     -- 2
    … LATERAL unnest(T̄.T) AS t2 …                     -- 2
```

[10] This is closer to a hack than conceptual beauty. Also, recall that LATERAL may have negative performance implications.

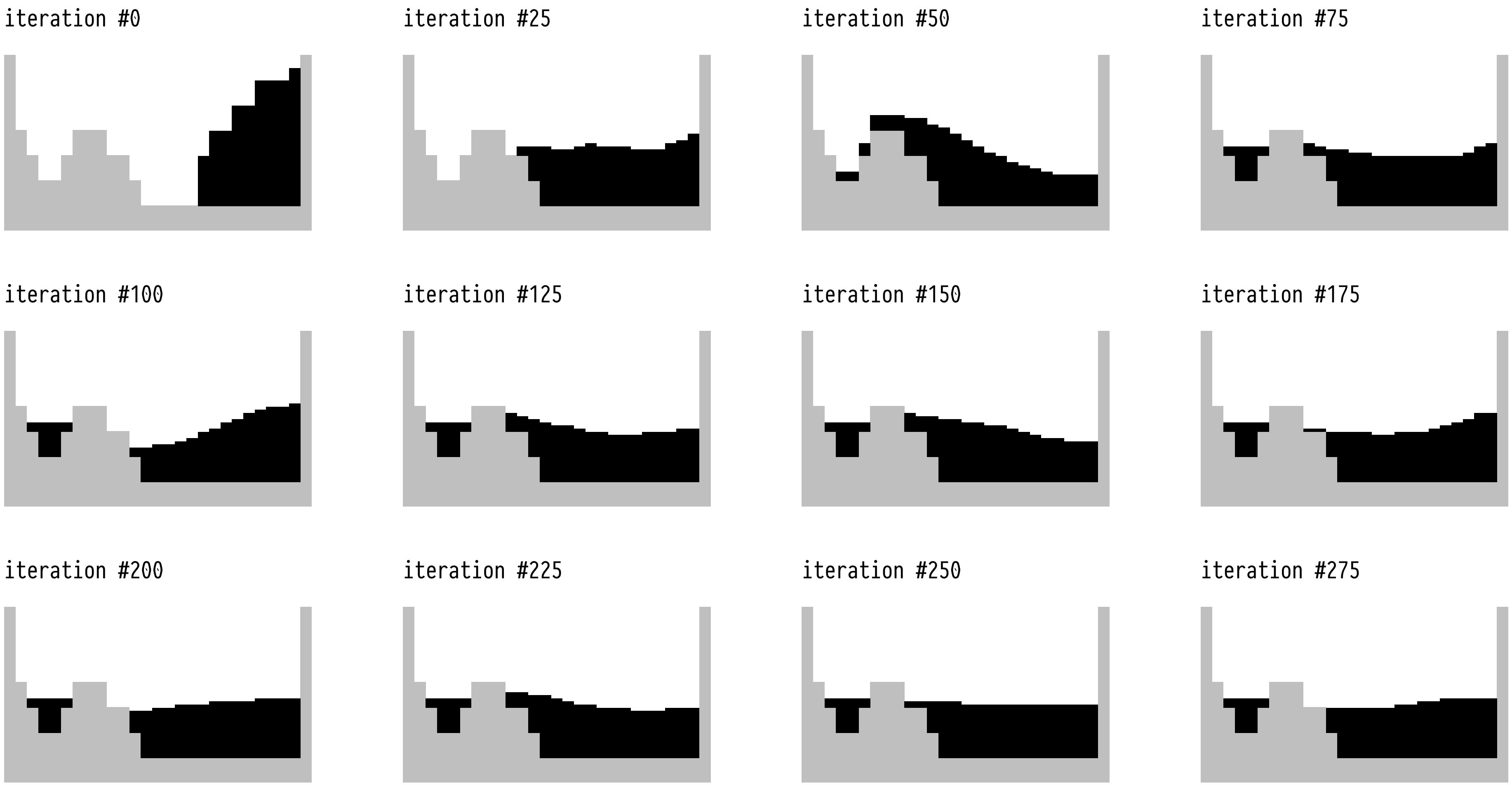
Liquid Flow (SQL Code)

```
WITH RECURSIVE
sim(iter,x,ground,water,kinetic) AS (
  SELECT 0 AS iter, f.x, f.ground, f.water, 0.0 AS kinetic
  FROM   fluid AS f

    UNION ALL

  SELECT s0.iter + 1 AS iter, s0.x, s0.ground,
         s0.water   + COALESCE(agg.Δwater  , 0) AS water,
         s0.kinetic + COALESCE(agg.Δkinetic, 0) AS kinetic
  FROM   (SELECT DISTINCT array_agg(row) OVER () AS sim FROM sim AS row) AS _,
         LATERAL unnest(sim) AS s0(iter int, x int, ground int, water numeric, kinetic numeric)
           LEFT OUTER JOIN
         LATERAL (SELECT infs.x, SUM(infs.Δwater) AS Δwater, SUM(infs.Δkinetic) AS Δkinetic
                  FROM   (SELECT (-- flow to the left                               --
                                  CASE WHEN ‹p₁›                                    --
                                  THEN array[ROW(s1.x-1, ‹Δwater›, ‹Δkinetic›),     --
                                             ROW(s1.x  , ‹Δwater›, ‹Δkinetic›),     --
                                             ROW(s1.x-1, ‹Δwater›, ‹Δkinetic›)      --
                                            ]                                       --
                                  END                                               --
                                  ||                                                --
                                  -- flow to the right                              --
                                  CASE WHEN ‹p₂›                                    --
                                  THEN array[ROW(s1.x+1, ‹Δwater›, ‹Δkinetic›),     --
                                             ROW(s1.x  , ‹Δwater›, ‹Δkinetic›),     --
                                             ROW(s1.x+1, ‹Δwater›, ‹Δkinetic›)      --
                                            ]                                       --
                                  END                                               --
                                 ) AS influence
                          FROM   unnest(sim) AS s1(iter int, x int, ground int, water numeric, kinetic numeric)
                          WINDOW horizontal AS (ORDER BY s1.x)
                         ) AS inf(influence),
                         LATERAL unnest(inf.influence) AS infs(x int, Δwater numeric, Δkinetic numeric)
                  GROUP BY infs.x
                  ) AS agg(x, Δwater, Δkinetic)
                  ON (s0.x = agg.x)
  WHERE  s0.iter < 300
)
SELECT s.iter, s.x, s.ground, s.water
FROM   sim AS s
ORDER BY s.iter, s.x;
```

Specific rules for the Liquid Flow CA, the enclosing SQL code is generic.

- Use **CASE** … **WHEN** … **THEN** … **END** to implement conditional rules.
- Use windows to access cell neighborhood.
- Use array concatenation (||) to implement sequences of rules.

Liquid Flow (First 275 Intermediate Simulation States)



One of *the* classic problems in Computer Science: **parsing.**

- Given the productions of a **context-free grammar**, can the input string be parsed (≡ generated) by the grammar?

start symbol *production rule (lhs→rhs)*

```
Expr → Expr Plus Term | Term
Term → Term Mult Fact | Fact
Fact → '1'
Plus → '+'
Mult → '×'
```

choice

non-terminal *terminal*

Grammar for simple arithmetic expressions:

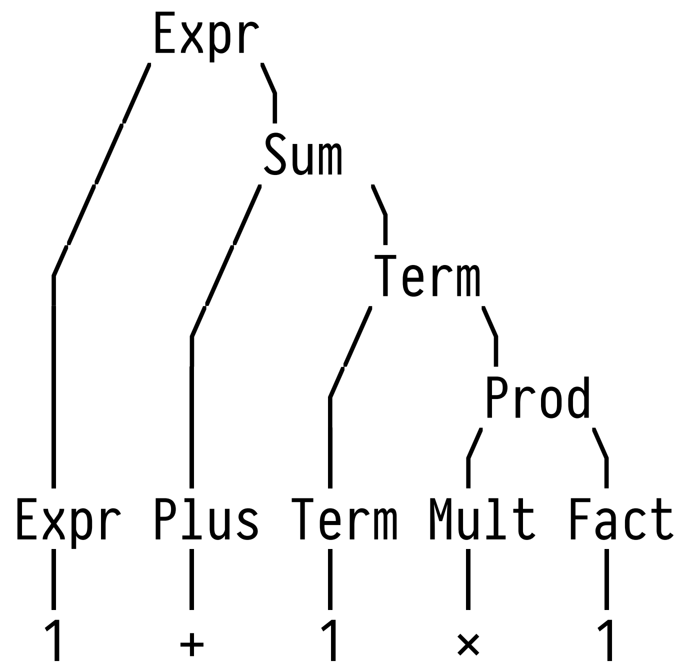
- operators +/×, literal 1,
- +/× left-associative,
- priority: × > +.

Chomsky Normal Form and Parse Trees

Consider grammars in **Chomsky Normal Form** only: rules read *lhs → terminal* or *lhs → non-terminal non-terminal*.

```
Expr → Expr Sum
Expr → Term Prod
Expr → '1'
Term → Term Prod
Term → '1'
Sum  → Plus Term
Prod → Mult Fact
Fact → '1'
Plus → '+'
Mult → '×'
```

Parse tree for input 1+1×1:



A Tabular Encoding of Chomsky Grammars

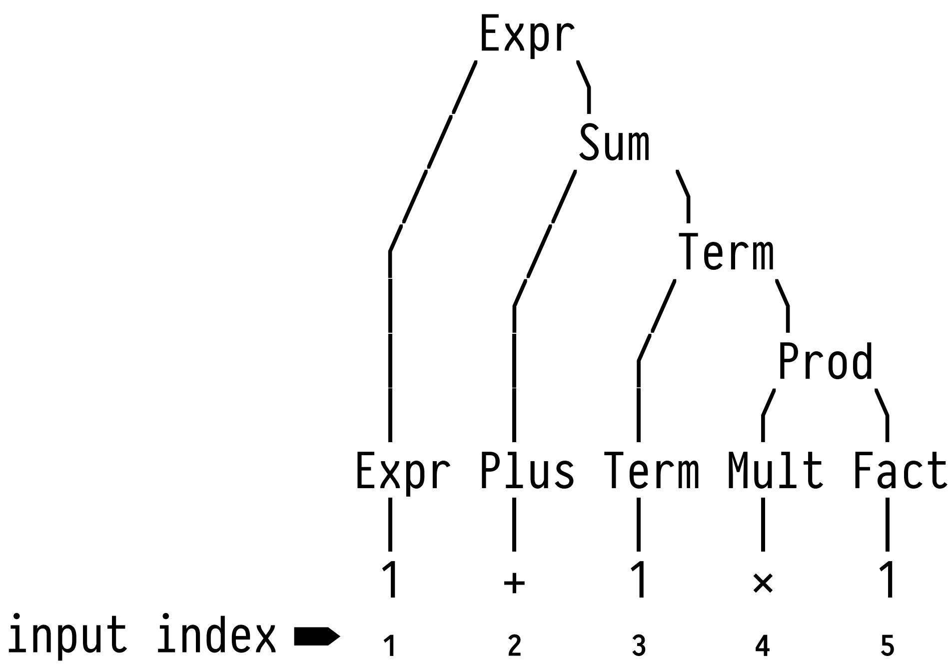
Simple encoding of the sample arithmetic expression grammar:

lhs	sym	rhs_1	rhs_2	start?
Expr	□	Expr	Sum	true
Expr	□	Term	Prod	true
Expr	1	□	□	true
Term	□	Term	Prod	false
Term	1	□	□	false
Sum	□	Plus	Term	false
Prod	□	Mult	Fact	false
Fact	1	□	□	false
Plus	+	□	□	false
Mult	×	□	□	false

- Exploits that rules can have one of two forms only.
- Embedded FD lhs → start? identifies one non-terminal as the grammar's start symbol.

Building a Parse Tree, *Bottom Up*

Invariant: Keep track of which part of the input (index from to to) can be generated by the lhs of a rule:



lhs	from	to
Expr	1	1
Plus	2	2
Term	3	3
Mult	4	4
Fact	5	5
Prod	4	5
Term	3	5
Sum	2	5
Expr	**1**	**5**

indicates parse successful

start symbol

spans entire input

Building a Tree in Layers Requires Access to the Past

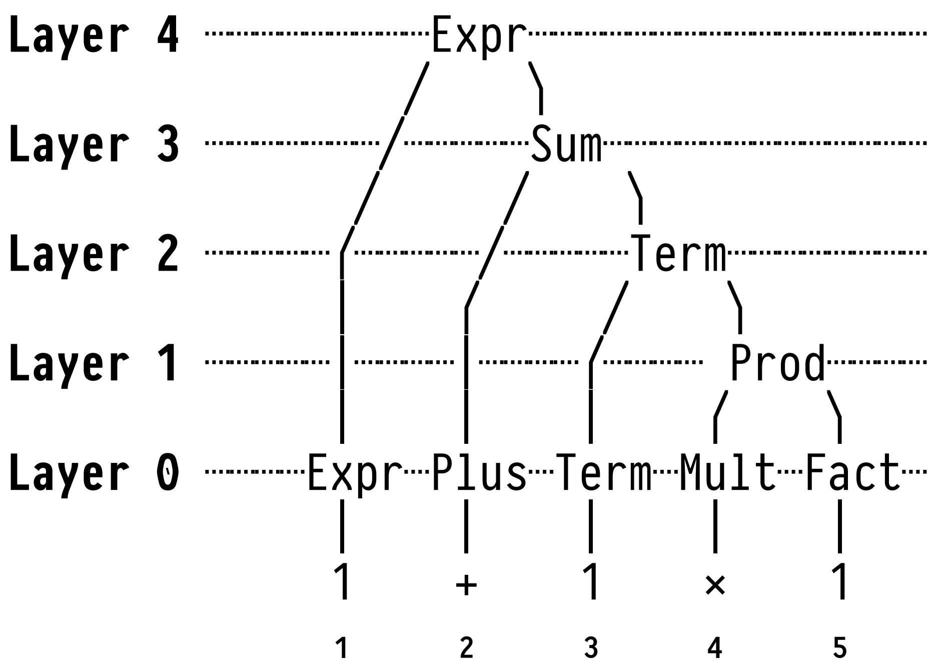


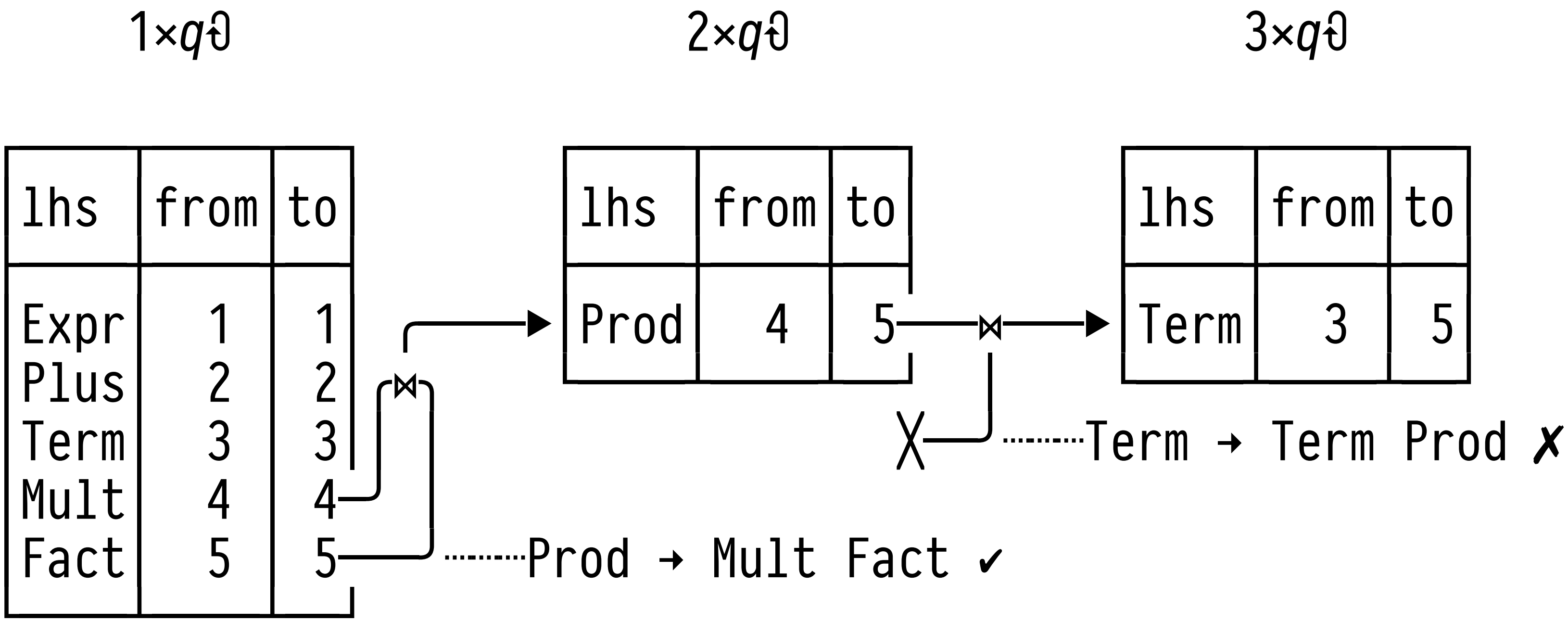
Table **parse**

lhs	from	to	
Expr	1	5	⬅ iteration #4
Sum	2	5	⬅ iteration #3
Term	3	5	⬅ iteration #2
Prod	4	5	⬅ iteration #1
Expr	1	1	found in iteration #0
Plus	2	2	
Term	3	3	
Mult	4	4	
Fact	5	5	

- To establish Term at **Layer 2** (iteration #2), we need Prod (**Layer 1**, iter #1 ✓) and Term (**Layer 0**, iter #0 ϟ).

WITH RECURSIVE's Short-Term Memory

Rows seen in table **parse** by…



- Parsing fact (Term,3,3) has been discovered by q_0 — more than one iteration ago — and is *not* available to 2×q↻.

Re-Injecting Early Iteration Results (SQL Template)

```
WITH RECURSIVE
T(iter, c1, …, cn) AS (
  SELECT 0 AS iter, t.*                          -- ⎱ add column iter (= 0) to
  FROM   (q0) AS t                               -- ⎰ result of q0

    UNION ALL

  SELECT t.iter + 1 AS iter, t.*
  FROM   (SELECT DISTINCT array_agg(row) OVER () AS T -- ⎱ multiple reads
          FROM   T AS row) AS T̄),                     -- ⎰ of T needed
         LATERAL (
           SELECT known.*                        -- ⎫ to the result of q↻ add already
           FROM   unnest(T̄.T) AS known           -- ⎬ discovered rows (will be kept
              UNION                              -- ⎭ since column iter advances)
           q↻                                    -- q↻ (access T via unnest(T̄.T))
         ) AS t
  WHERE p                                        -- stop condition
)
```

WITH RECURSIVE With Long-Term Memory

Rows seen in table **parse** by…

1×*q*↺

iter	lhs	from	to
0	Expr	1	1
0	Plus	2	2
0	Term	3	3
0	Mult	4	4
0	Fact	5	5

2×*q*↺

iter	lhs	from	to
1	Prod	4	5
1	Expr	1	1
1	Plus	2	2
1	Term	3	3
1	Mult	4	4
1	Fact	5	5

3×*q*↺

iter	lhs	from	to
2	Term	3	5
2	Prod	4	5
2	Expr	1	1
2	Plus	2	2
2	Term	3	3
2	Mult	4	4
2	Fact	5	5

⫽ ≡ row added by reinjection

Parsing: Cocke-Younger-Kasami Algorithm (CYK)

The **CYK algorithm** builds parse trees bottom up, relying on formerly discovered partial parses (dynamic programming):

- Iteratively populate table parse(lhs,from,to):
  - $[q_0]$: For each $lhs \rightarrow terminal$: if $terminal$ is found at index $from \ldots to$ in input, add $(lhs,from,to)$ to parse.
  - $[q_{\circlearrowleft}]$: For each pair $(lhs_1,from_1,to_1)$, $(lhs_2,from_2,to_2)$ in parse × parse:[11] add $(lhs_3,from_1,to_2)$ if
    1. $to_1 + 1 = from_2$ and
    2. $lhs_3 \rightarrow lhs_1\ lhs_2$.

[11] Implies a self-join of parse, leading to non-linear recursion.

Parsing Using CYK (Core SQL Code) ✎

```
WITH RECURSIVE
parse(…, lhs, "from", "to") AS (
  SELECT …, g.lhs, i AS "from", i + length(g.sym) - 1 AS "to"
  FROM   grammar AS g,
         generate_series(1, length(input)) AS i,
  WHERE  g.sym IS NOT NULL
  AND    substr(input, i, length(g.sym)) = g.sym

    UNION ALL

  ⋮                                     -- ⚠ re-injection code omitted
  SELECT …, g.lhs, l."from", r."to"
  FROM   grammar AS g,
         parse AS l, parse AS r         -- ⚠ need 2 × unnest() here
  WHERE  l."to" + 1 = r."from"
  AND    (g.rhs1, g.rhs2) = (l.lhs, r.lhs)
  ⋮
)
```