

Advanced SQL

06 — Recursion

Summer 2020

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 magnificent...

Recursion in SQL: **WITH RECURSIVE**

Recursive common table expression (CTE):

WITH RECURSIVE

| | | |
|---|---|---|
| $\langle T_1 \rangle (\langle C_{11} \rangle, \dots, \langle C_{1, k_1} \rangle) \text{ AS } ($ $\quad \langle q_1 \rangle),$ $\quad \vdots$ $\langle T_n \rangle (\langle C_{n1} \rangle, \dots, \langle C_{n, k_n} \rangle) \text{ AS } ($ $\quad \langle q_n \rangle)$ | } | Queries q_j may refer to all T_i |
| $\langle q \rangle$ | } | q may refer to all T_i |

- In particular, any q_j may refer to itself (\odot)! Mutual references are OK, too. (Think **letrec** in FP.)
- Typically, final query q performs post-processing only.

Shape of a Self-Referential Query

WITH RECURSIVE

| | |
|--|---|
| $\langle T \rangle (\langle C_1 \rangle, \dots, \langle C_k \rangle)$ $\text{AS } ($ $\quad \langle q_0 \rangle$ UNION [ALL] $\quad \langle q_{\exists}(T) \rangle$ $)$ $\langle q \rangle (\langle T \rangle)$ | $--$ common schema of q_0 and $q_{\exists}(\cdot)$ $--$ base case query, evaluated once $--$ either UNION or UNION ALL $--$ recursive query refers to T $--$ itself, evaluated repeatedly $--$ final post-processing query |
|--|---|

- Semantics in a nutshell:

$$\underbrace{q(q_{\exists}(\dots q_{\exists}(q_{\exists}(q_0))\dots)) \cup \dots \cup q_{\exists}(q_{\exists}(q_0)) \cup q_{\exists}(q_0) \cup q_0}_{\text{repeated evaluation of } q_{\exists} \text{ (when to stop?)}}$$

Semantics of a Self-Referential Query (**UNION** Variant)

Iterative and recursive semantics—both are equivalent:

| | |
|---|--|
| <pre> iterate($q\vartheta$, q_0): $r \leftarrow q_0$ $t \leftarrow r$ while $t \neq \emptyset$ $t \leftarrow q\vartheta(t) \setminus r$ $r \leftarrow r \uplus t$ return r </pre> | <pre> recurse($q\vartheta$, r): if $r \neq \emptyset$ then return $r \uplus \text{recurse}(q\vartheta, q\vartheta(r) \setminus r)$ else return \emptyset </pre> |
|---|--|

- Invoke the recursive variant with **recurse($q\vartheta$, q_0)**.
- \uplus denotes disjoint set union, \setminus denotes set difference.
- **$q\vartheta(\cdot)$** is 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 `series` of integers $i \in \{from, from+1, \dots, to\}$:

```
WITH RECURSIVE
  series(i) AS (
    ▲ VALUES (<from>)                -- q0
    ┌   UNION
    │   SELECT s.i + 1 AS i           -- }
    │   FROM   series AS s           -- } q∂(series)
    │   WHERE  s.i < <to>             -- }
  )
TABLE series
```

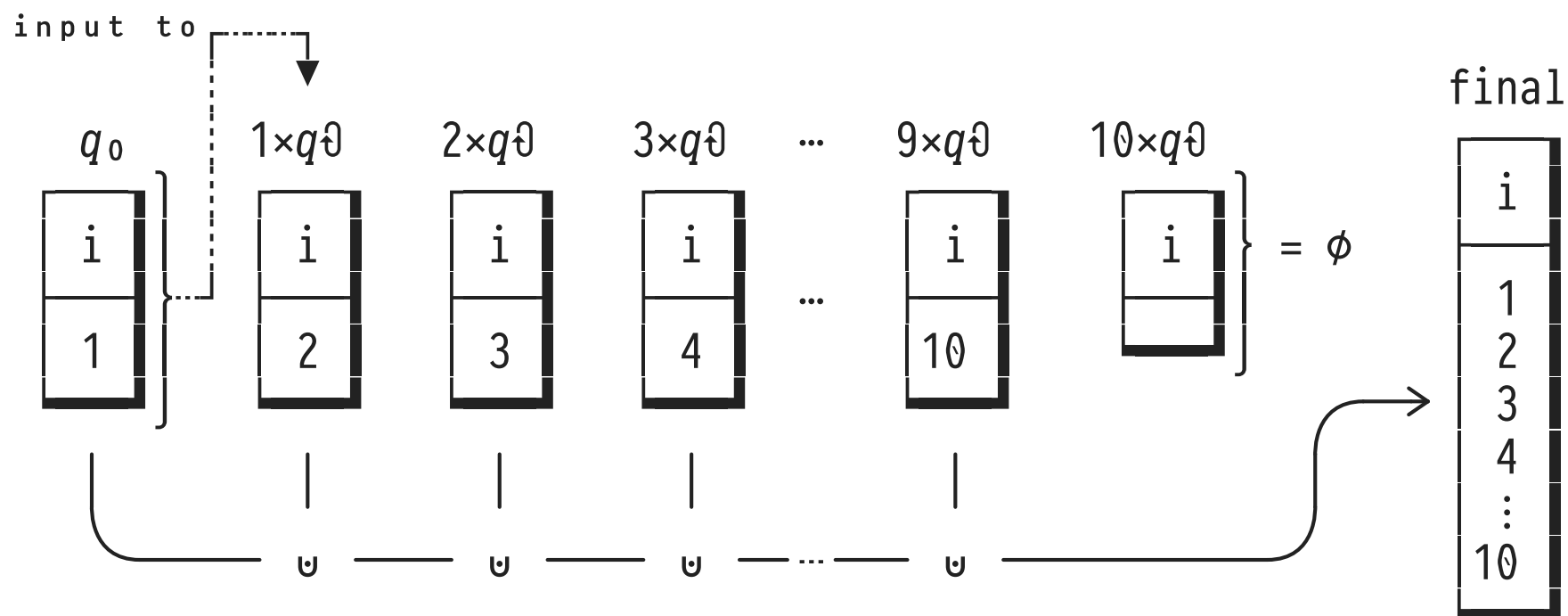
▲ self-reference
└─▶

- **Q:** Given the predicate $s.i < \langle to \rangle$, will `to` indeed be in the final table?

🔧 A Home-Made `generate_series()`

- Assume *from* = 1, *to* = 10:

New rows in table **series** after evaluation of...



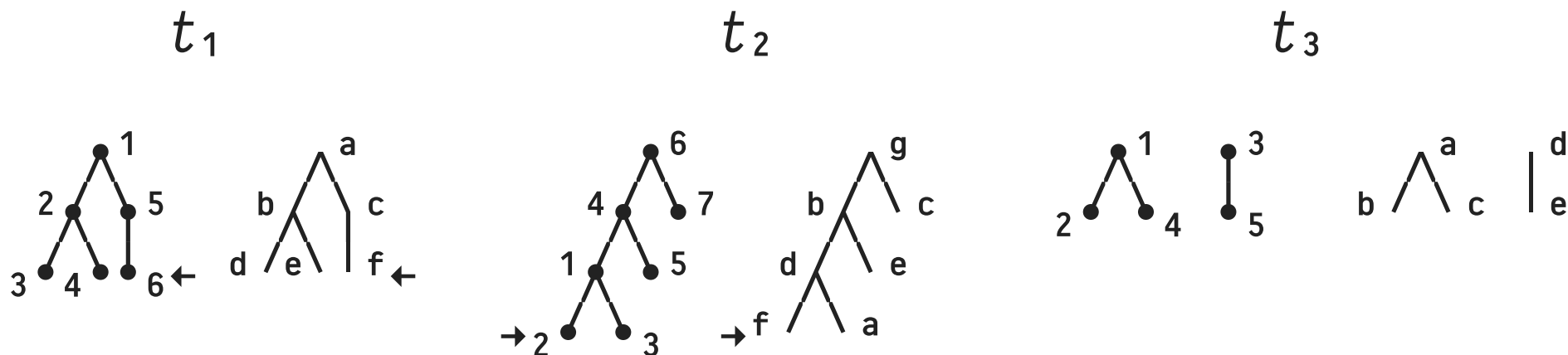
Semantics of a Self-Referential Query (**UNION ALL** Variant)

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

| | |
|--|---|
| <pre> iterate^{a11}($q\vartheta$, q_0): $r \leftarrow q_0$ $t \leftarrow r$ while $t \neq \phi$ $t \leftarrow q\vartheta(t)$ $r \leftarrow r \uplus t$ return r </pre> | <pre> recurse^{a11}($q\vartheta$, r): if $r \neq \phi$ then return $r \uplus \text{recurse}^{\text{a11}}(q\vartheta, q\vartheta(r))$ else return ϕ </pre> |
|--|---|

- Invoke the recursive variant via $\text{recurse}^{\text{a11}}(q\vartheta, q_0)$.
- \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 $n \equiv \text{parents}[n]$):

| tree | parents ($\square \equiv \text{NULL}$) | labels |
|-------|--|---|
| t_1 | $\{\square, 1, 2, 2, 1, 5\}$ | $\{'a', 'b', 'd', 'e', 'c', 'f'\}$ |
| t_2 | $\{4, 1, 1, 6, 4, \square, 6\}$ | $\{'d', 'f', 'a', 'b', 'e', 'g', 'c'\}$ |
| t_3 | $\{\square, 1, \square, 1, 3\}$ | $\{'a', 'b', 'd', 'c', 'e'\}$ |
| | 1 2 3 4 5 6 7 | 1 2 3 4 5 6 7 \leftarrow 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
```

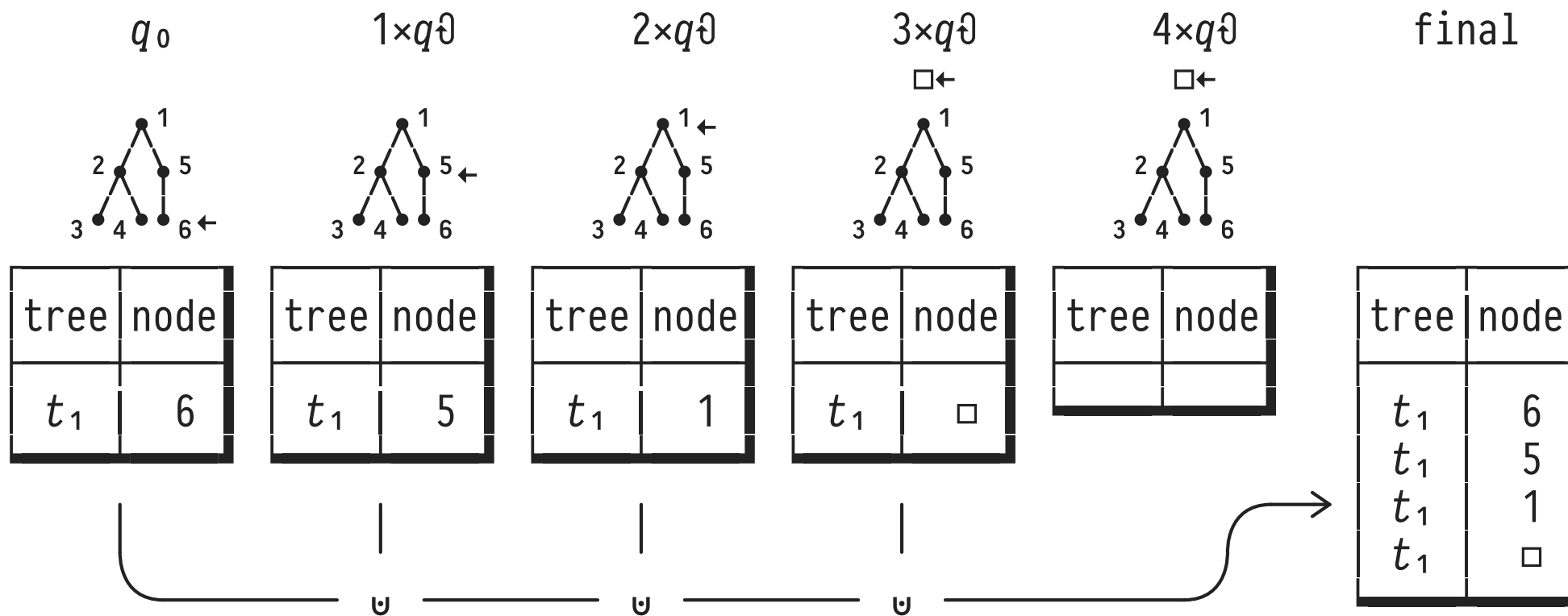
```
)
```

```
TABLE paths
```

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

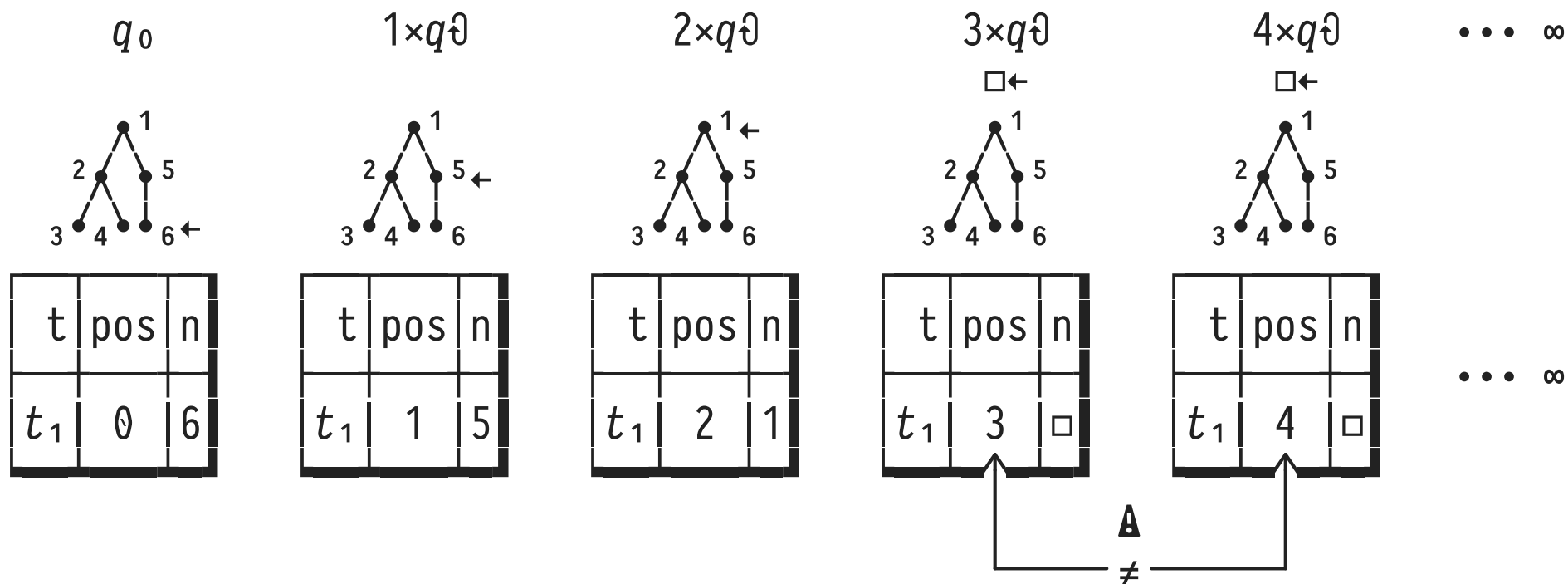
🔧 A Trace of the Path in Tree t_1

New rows produced by...



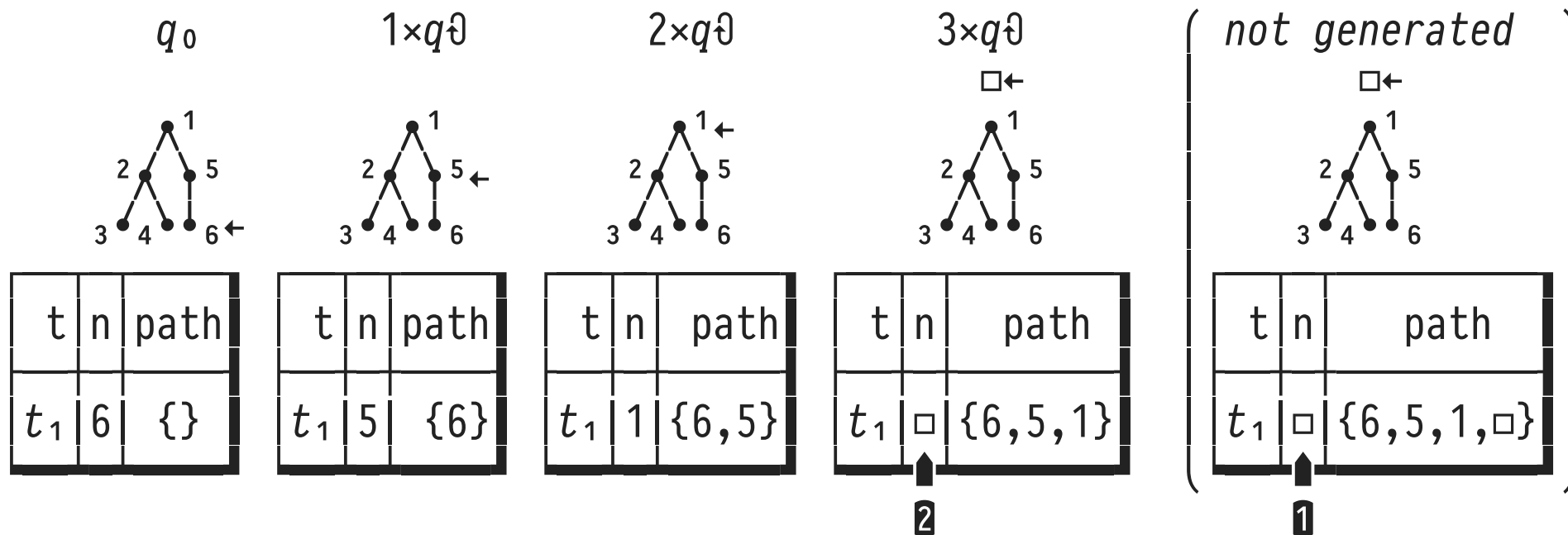
- $4 \times q_0$ yields no new rows (recall: $t.\text{parents}[\text{NULL}] \equiv \text{NULL}$).

🔧 Ordered Path in Tree t_1 (New Rows Trace)



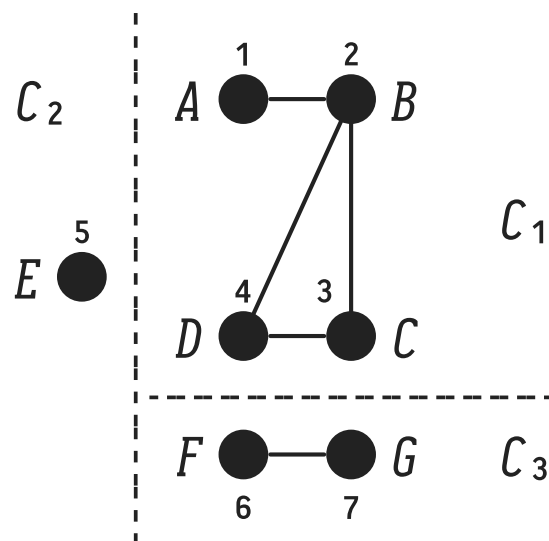
The (non-)generation of new rows to ensure termination is the user's responsibility—a common source of ~~✖~~.

🔧 Path as Array in Tree t_1 (New Rows Trace)



- 1 Ensure termination (enforce ϕ): filter on $n \neq \square$ in q_0 .
- 2 Post-process: keep rows of last iteration ($n = \square$) only.

2 | Connected Components in a Graph



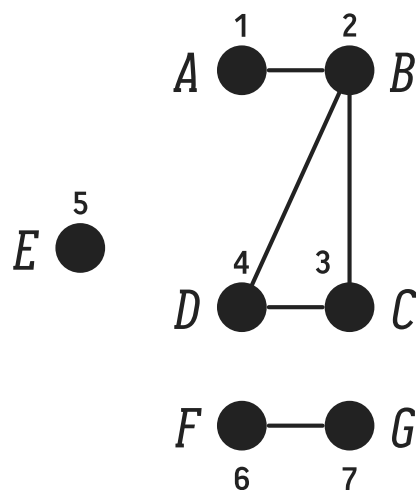
- Given an undirected graph G , find its **connected components** C_i :

For any two nodes v_1, v_2 in C_i , there exists a path $v_1 \text{---} v_2$ (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



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

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

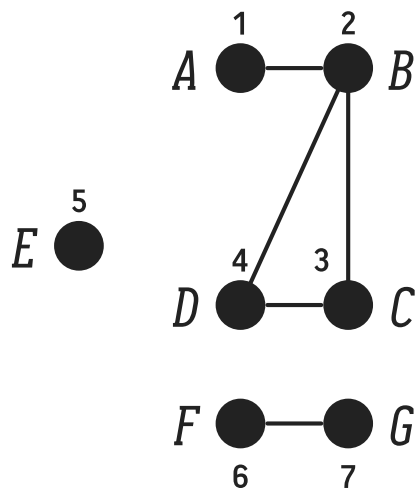
→
derive

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

} $A \rightleftharpoons B$
 $A — B$

- Use tables `nodes` and `graph` to formulate the algorithm.

🔧 Computing Connected Components (Query Plan)



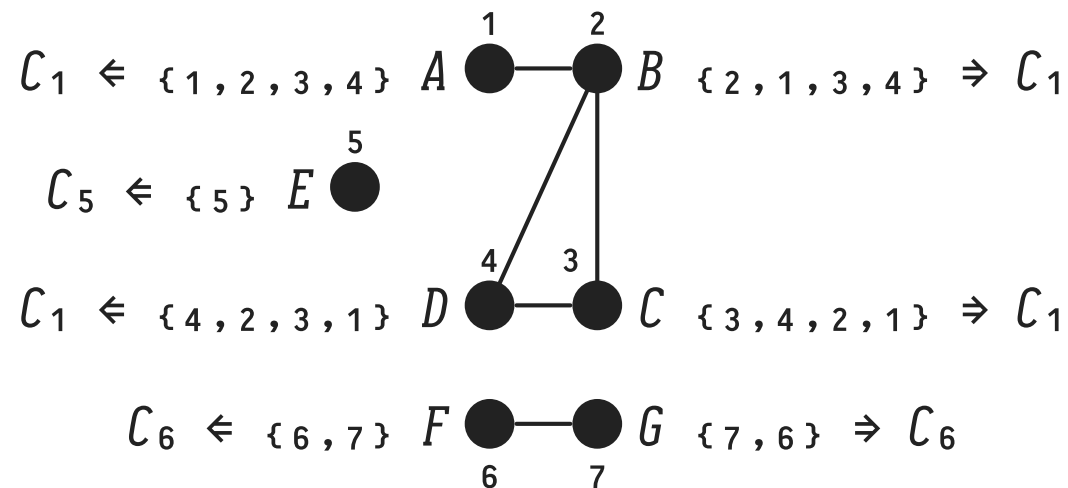
1. For each node n , start a **walk** through the graph. Record each node f (“front”) that we can **reach** from n .
2. For each n , use the **minimum ID** i of all front nodes as n 's component C_i .

⇒ 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, \mathcal{C}_i derived component ID:



- Tasks for further post-processing:
 - Assign sane component IDs ($\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$).
 - Extract subgraphs based on components' node sets.

🔧 Recursive Graph Walks, From All Nodes at the Same Time

WITH RECURSIVE

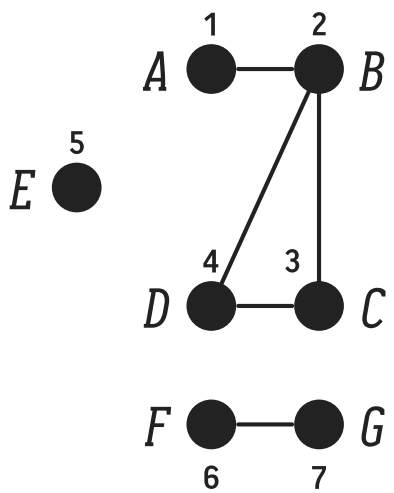
```
walks(node, front) AS (
  SELECT n.node, n.node AS front  -- (n,n) ∈ walks: we can
  FROM   nodes AS n              -- reach ourselves

  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 $(n, f) \in \text{walks}$, 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 |

$$1 \times q_0$$

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

$$2 \times q_0$$

| node | front |
|------|-------|
| 1 | 3 |
| 1 | 4 |
| 3 | 1 |
| 4 | 1 |

$$3 \times q_0$$

| node | front |
|------|-------|
| | |

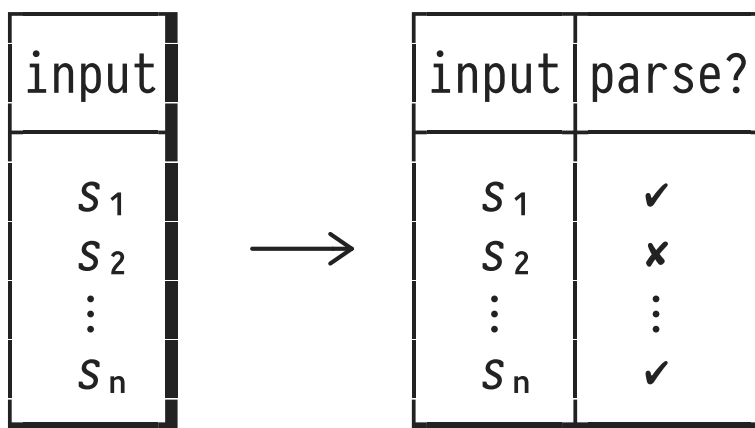
3 | Recursive Text Processing

- 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 $s :: \text{text}$, $n \geq 1$):

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

🔧 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.

🔧 Breaking Bad (Season 2)

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

```
([A...Za...z]+[0...9]*([0...9]*[+-]))?)+
```

| compound | formula |
|-----------|--|
| citrate | C ₆ H ₅ O ₇ ³⁻ |
| glucose | C ₆ H ₁₂ O ₆ |
| hydronium | H ₃ O ⁺ |
| ⋮ | ⋮ |

Table compounds

- Generally: support regular expressions *re* of the forms *c*, *[c₁c₂...c_n]*, *re₁re₂*, *re**, *re+*, *re?*, *re₁|re₂*.

🔧 From Regular Expression to Finite State Machine (FSM)

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

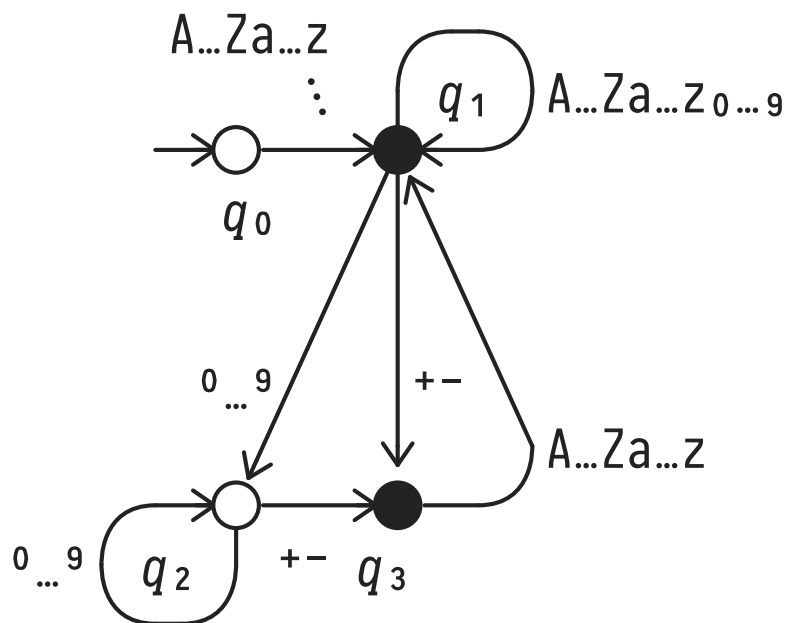


Table fsm

| 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> | <u>state</u> | <u>input</u> |
|-----------------|-------------|--------------|--------------|
| c | s | q | f |

Table `match`

- After $s \geq 0$ 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      --                       $\equiv q_0$ 
```

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 Progress (by Compound / by Step)

1 Focus on individual 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 ← empty string |
| ⋮ | ⋮ | ⋮ | ⋮ |
| hydronium | 0 | 0 | H_3O^+ |
| hydronium | 1 | 1 | $_3O^+$ |
| hydronium | 2 | 1 | O^+ |
| hydronium | 3 | 1 | $^+$ |
| hydronium | 4 | 3 ← 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_{step} will never produce duplicate rows** and
 2. there is no point in computing the difference $q_{\text{step}}(t) \setminus r$ in **$\text{iterate}(q_{\text{step}}, q_0)$: $q_{\text{step}}(t) \cap r = \emptyset$.**
- **q_{step} is guaranteed to evaluate to \emptyset at one point**, since...
 1. one character is chopped off in each iteration and **$\text{length}(m.\text{input}) > 0$** will yield **false** eventually, or
 2. the FSM gets stuck due to an invalid input character (**$\text{strpos}(f.\text{labels}, \text{left}(m.\text{input}, 1))$** yields 0).

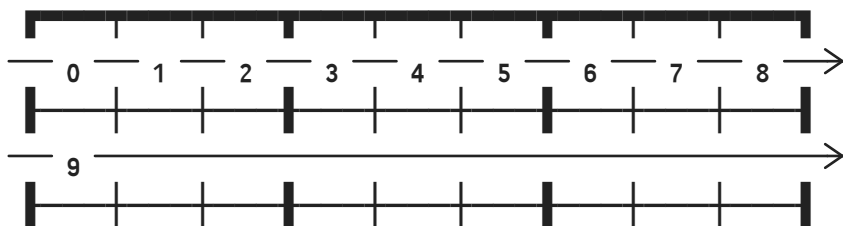
4 : Recursive Array Processing: Solving Sudoku³ Puzzles

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

- Fill in the blanks with digits $\in \{1, \dots, 9\}$ such that
 1. no 3×3 box 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.” (Yes, this board has a unique solution.)

🔧 Row-Major Array-Encoding of a 2D Grid



- Build row-wise `int[]` array of 81 cells $\in \{0, \dots, 9\}$, with $0 \equiv \text{blank}$.
- Derive **row/column/square index** from cell $c \in \{0, \dots, 80\}$:
 - Row of c : $(c / 9) * 9 \in \{0, 9, 18, 27, 36, 45, 54, 63, 72\}$
 - Column of c : $c \% 9 \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
 - Box of c : $((c / 3) \% 3) * 3 + (c / 27) * 27 \in \{0, 3, 6, 27, 30, 33, 54, 57, 60\}$
- (Clunky—But: relational encodings of grids upcoming.)

🔧 Finding All Puzzle Solutions (Query Plan)

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

Table `sudoku`

1. Invariant:

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

- In each iteration, **fill in all digits** $\in \{1, \dots, 9\}$ 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/box of s.b at offset i>))
)

```

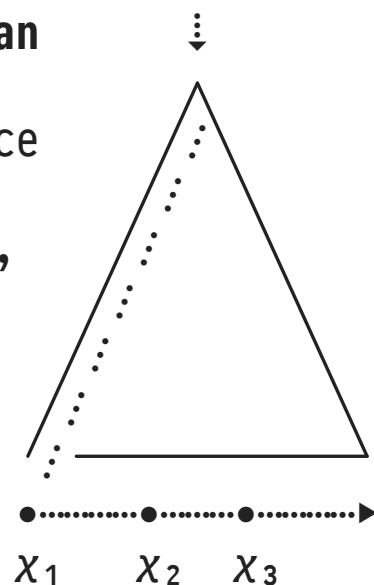

5 | Emulating Physical Operator Behavior: Loose Index Scans

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`.

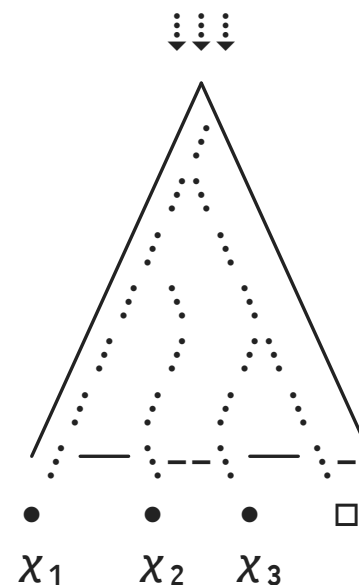
1 Regular Index Scan

- 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| = 106` 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 12.1 on macOS Mojave (10.14.6), 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 Learning⁵

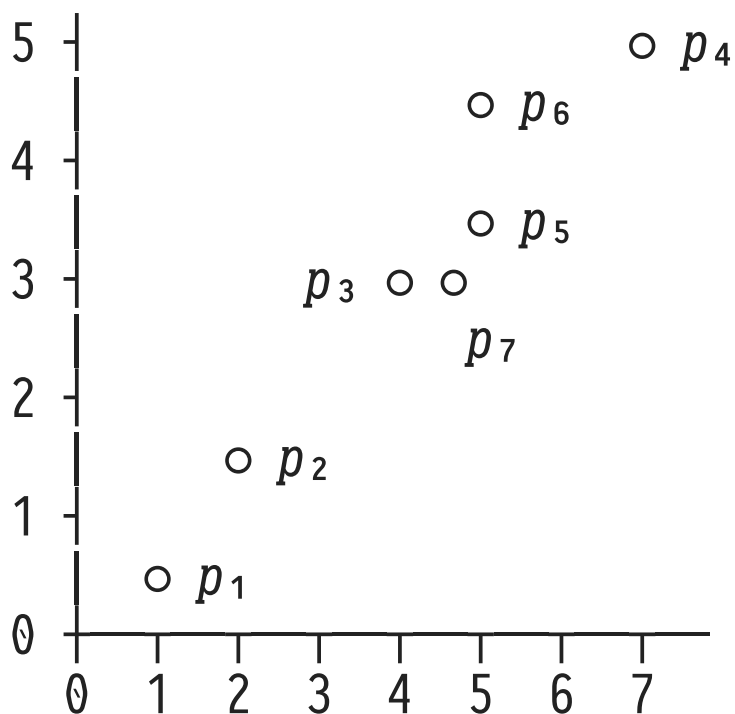
Most sizable *source data* for Machine Learning (ML) problems reside **inside** database systems. Actual *ML algorithms* are predominantly implemented **outside** the DBMS—Python, R, MatLab—however:

- 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.

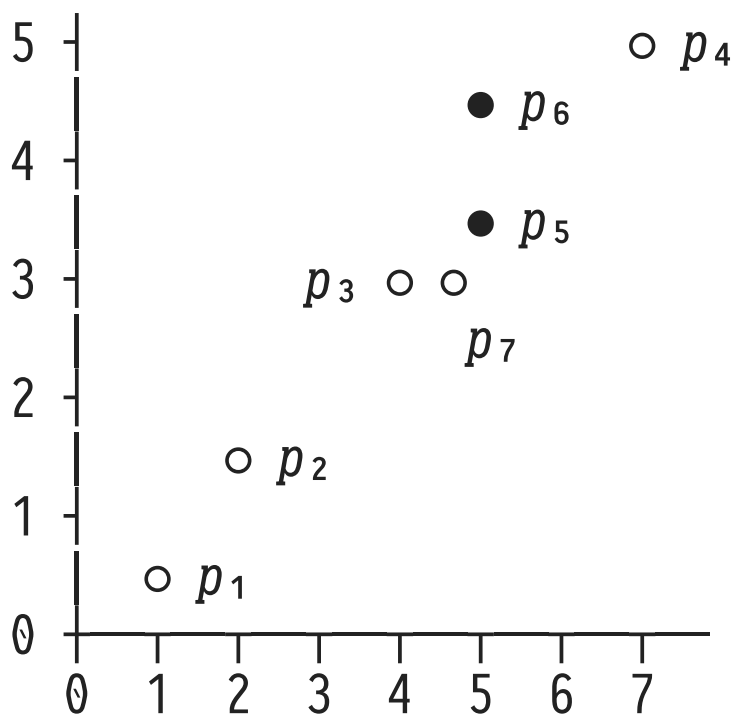
🔧 K-Means Clustering



- **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)

| <u>point</u> | loc |
|--------------|-----------------|
| 1 | point(1.0, 1.0) |
| 2 | point(2.0, 1.5) |
| ⋮ | ⋮ |

Table `points`

- Picking random rows from table `T`:

```

TABLE <T>
ORDER BY random()
LIMIT <k>                                -- pick (at most) k rows

SELECT t.*
FROM <T> AS t
TABLESAMPLE BERNOULLI(s)                 -- pick ≈ s% random rows

```

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

Invariant:

| <u>iter</u> | <u>point</u> | <u>cluster</u> | <u>mean</u> |
|-------------|--------------|----------------|-------------|
| <i>i</i> | <i>p</i> | <i>c</i> | <i>m</i> |

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

```

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/current/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])`.
 - A strictly increasing `iter` counter will never lead to a fixpoint anyway \Rightarrow endless recursion.
 - 💡 **User-defined type** that admits counting but whose values are all considered equal.

SQL Notes and Grievance (3)

- Is the subquery (1. Assignment) in the recursive query [q3](#) 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 [SELECT](#) or [WHERE](#) clauses are forbidden. ⚠

7 | Table-Driven Query Logic (Control Flow → Data Flow)

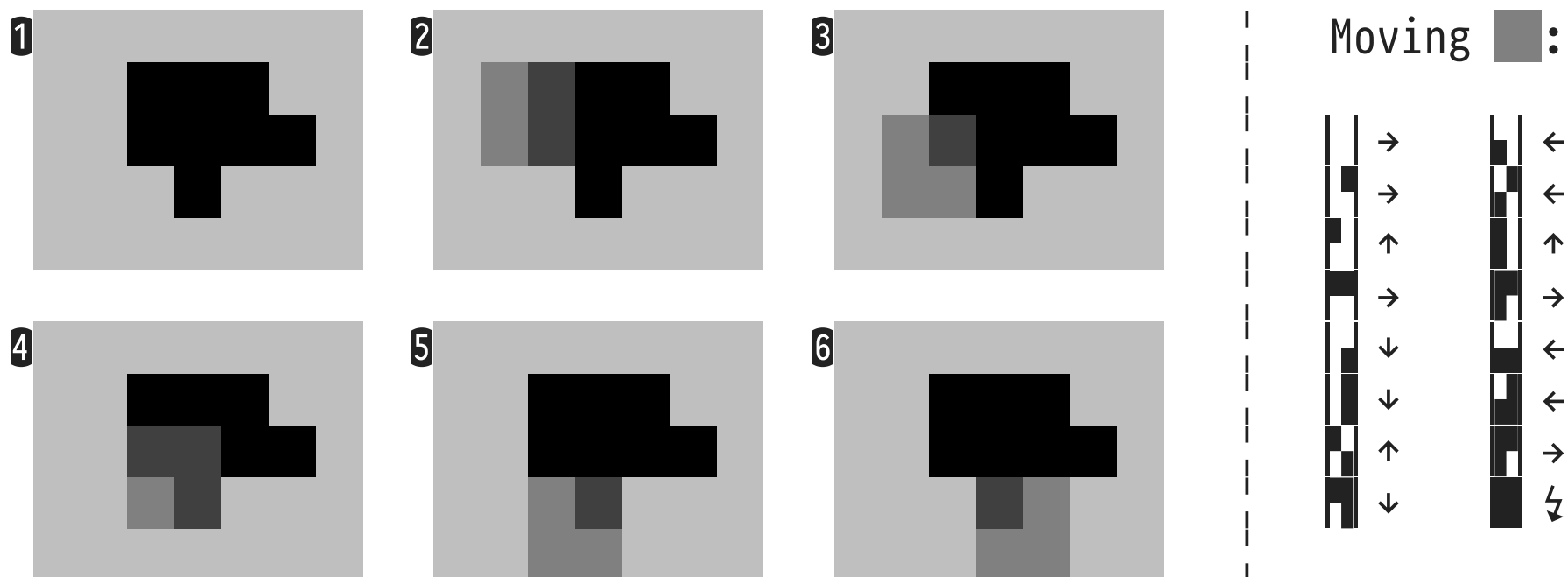
SQL provides a family of constructs to encode the **logic** (in the sense of **control flow**) of algorithms:

1. Obviously: `WHERE <p>`, `HAVING <p>`,
2. `<q1> UNION ALL <q2> UNION ALL ... UNION ALL <qn>`
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:



- **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 $(1, 0) \rightarrow$,  maps to $(0, -1) \uparrow$.
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)`:
 - `[q0]`: Start with $(1, 1) \in \text{march}$.
 - `[q∂]`: 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. 👍

8 | Encoding Cellular Automata in SQL

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]

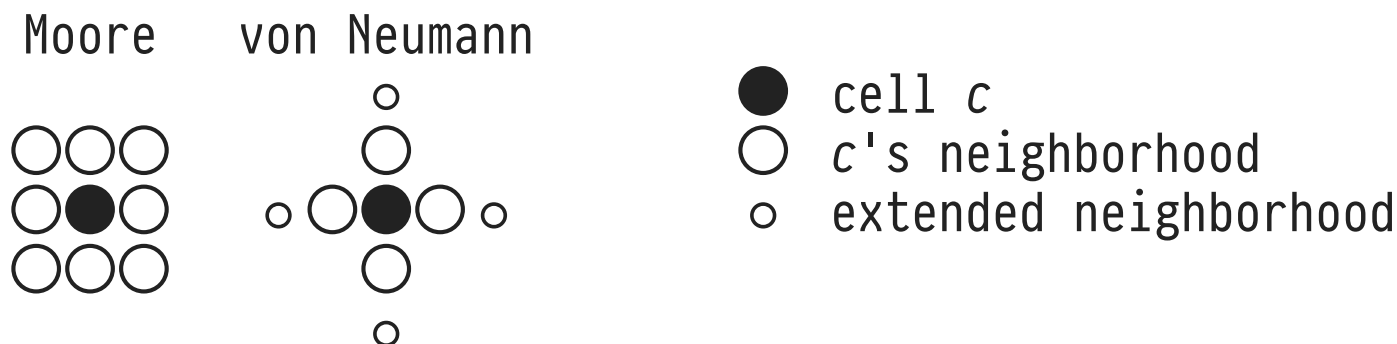
Both flavors lead to quite different SQL implementations.

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 [q4](#) (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

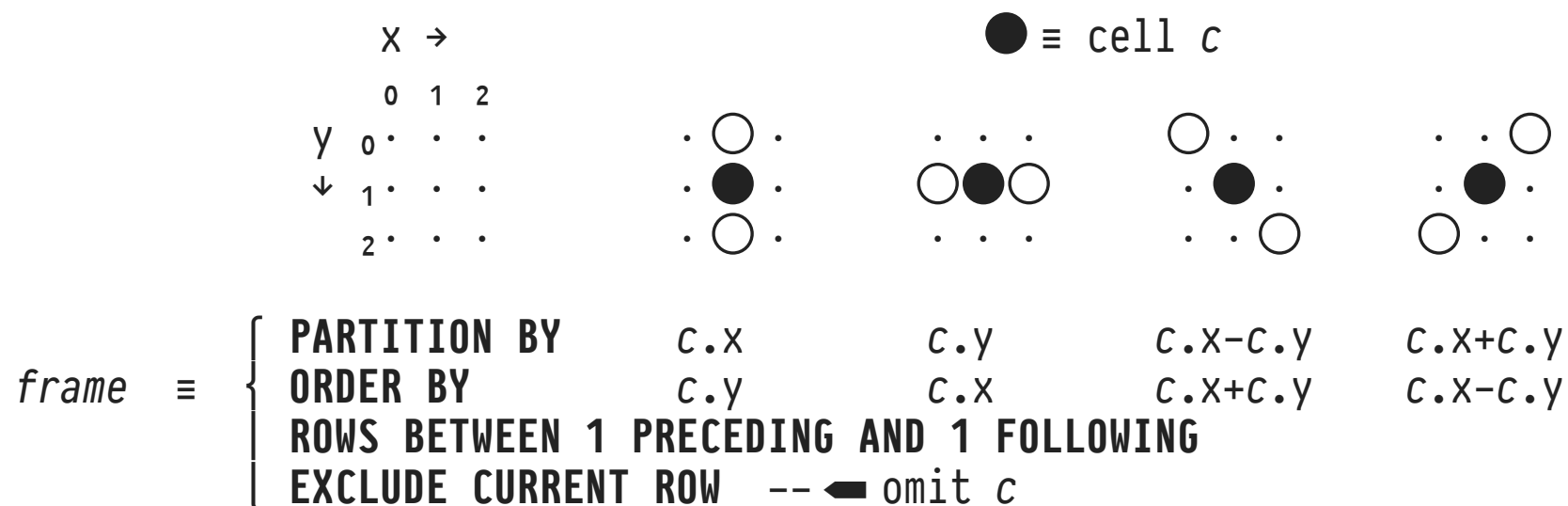
WITH RECURSIVE syntactically restricts query forms, in particular the references to the recursive table T :

1. No references to T in q_0 .
2. A single reference to T in $q\vartheta$ 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 $q\vartheta$ (window functions *do* work).
7. No **ORDER BY**, **OFFSET**, or **LIMIT** in $q\vartheta$.

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

Accessing the Cell Neighborhood — A Solution! ☺

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



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

🔧 Conway's Game of Life

Life⁸ 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:

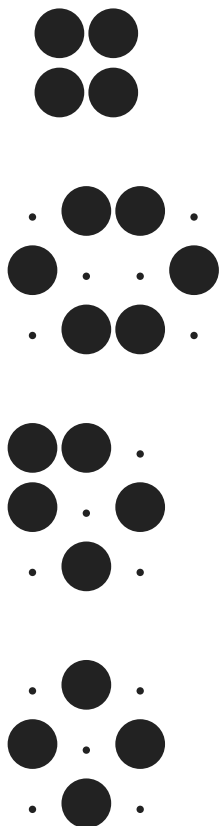
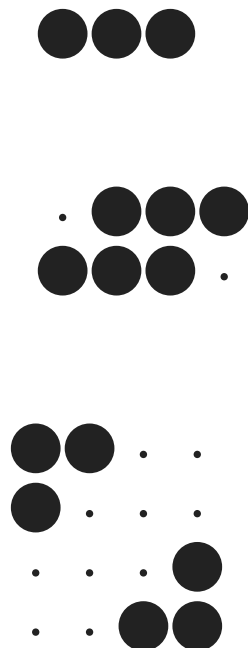
1. If c is alive and $p < 2$, c dies (underpoulation).
2. If c is alive and $2 \leq p \leq 3$, c lives on.
3. If c is alive and $3 < p$, c dies (overpopulation).
4. If c is dead and $p = 3$, c comes alive (reproduction).

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

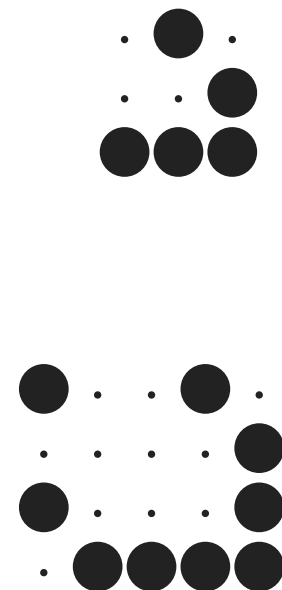
⁸ John H. Conway († April 2020), column *Mathematical Games* in *Scientific American* (October 1970).

🔧 Life — A Few Notable Cell Patterns

Still

Oscillators
(period: 2)

Spaceships



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

```

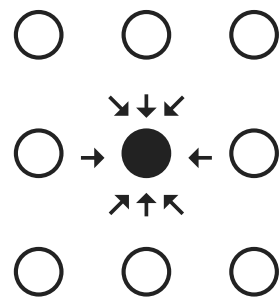
WITH RECURSIVE
life(gen,x,y,cell) AS (
  ⋮
  SELECT 1.gen + 1 AS gen, 1.x, 1.y,
         CASE (1.cell, ( SUM(1.cell) OVER <horizontal ...>
                        + SUM(1.cell) OVER <vertical :>
                        + SUM(1.cell) OVER <diagonal :>
                        + SUM(1.cell) OVER <diagonal :>
                      )
              )
              -- (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 1
  ⋮
)

```

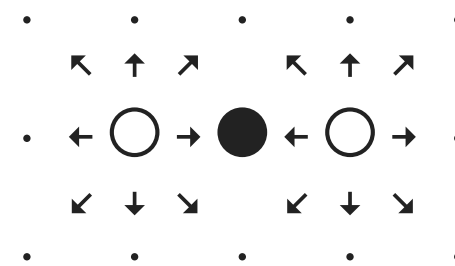
9 | CA with Cells That Influence Their Neighborhood

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

❶ “influenced by”

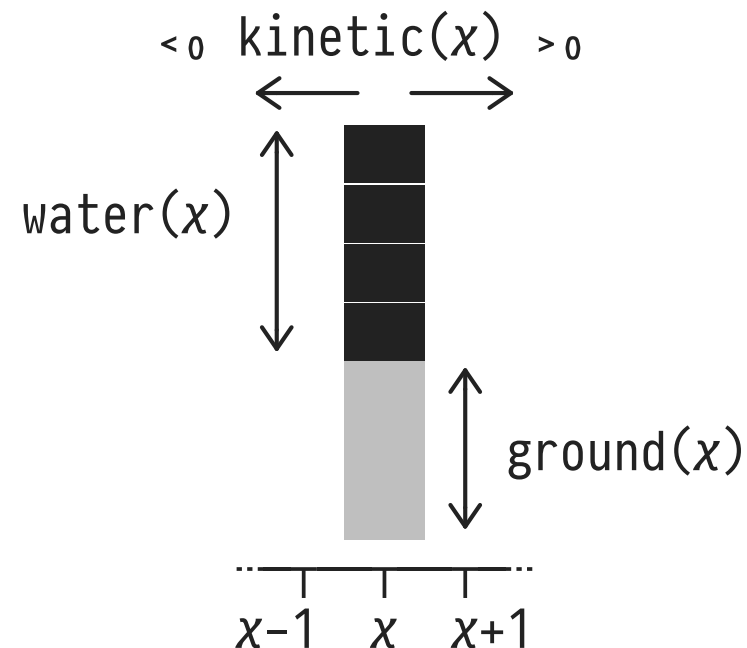
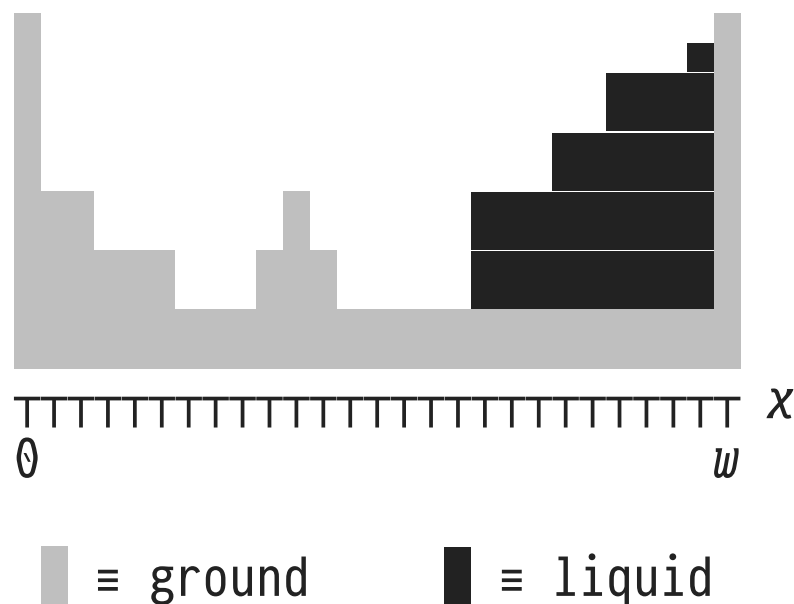


❷ “influences”



- 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 ($\text{pot}(x) \equiv \text{ground}(x) + \text{water}(x)$)
- left/right **kinetic energy** at x ($\text{kinetic}(x)$)

🔧 Liquid Flow: Cellular Automaton⁹

```

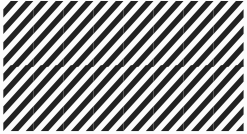

Δwater ← (0,0,...,0)  -- changes to water and energy levels
Δkin   ← (0,0,...,0)  -- in next generation
for x in 1...w-1:
    -- liquid flow to the left?
    if pot(x)-kin(x) > pot(x-1)+kin(x-1):           -- force ← > force →
        flow ←  $\frac{1}{4} \times \min(\text{water}(x), \text{pot}(x)-\text{kin}(x)-(\text{pot}(x-1)+\text{kin}(x-1)))$ 
        Δwater(x-1) ← Δwater(x-1)+flow              -- } aggregate the
        Δwater(x)   ← Δwater(x) -flow                -- } influences on
        Δkin(x-1)   ← Δkin(x-1) -  $\frac{1}{2} \times \text{kin}(x-1) - \text{flow}$   -- } cells @ x / x-1
    -- liquid flow to the right?
    if pot(x)+kin(x) > pot(x+1)-kin(x+1):           -- force → > force ←
        : -- "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* subreddit. $\frac{1}{4}$, $\frac{1}{2}$ 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
         (  ) AS agg(x,y,Δstate) -- }  encodes rules
         -- extract all influences on cell c0 (□ if none)
         ON (c0.x, c0.y) = (agg.x, agg.y)
  WHERE  c0.iter < <iterations>
)

```

- Design: no $\text{agg}(x,y,_)$ if cell @ x,y doesn't change state.
- Assume that z is neutral element for \oplus : $s \oplus z = s$.

CA: From Individual to Aggregated Influences (SQL Template)

```

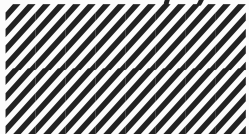
:
SELECT c0.iter + 1 AS iter, c0.x, c0.y,
       c0.state  $\oplus$  COALESCE(agg. $\Delta$ state, <z>) AS state
FROM   cells AS c0 LEFT OUTER JOIN
       -- find and aggregate influences on all cells @ x,y
       (SELECT infs.x, infs.y, <agg>(infs. $\Delta$ state) AS  $\Delta$ state
        FROM   (  ) AS infs(x,y, $\Delta$ state)
        GROUP BY infs.x, infs.y
       ) AS agg(x,y, $\Delta$ state)
       -- extract all influences on cell c0 ( $\square$  if none)
       ON (c0.x, c0.y) = (agg.x, agg.y)
:

```


- $(x,y,\Delta\text{state}) \in \text{infs}$: individual influence on cell @ x,y .
- Typically, we will have $\text{agg} = (\phi, z, \oplus)$.

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  -- } 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**,  computes an **array of influence** **influence** with elements **(x,y,Δstate)**: **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, ),      -- influence on ← cell
                ROW(c1.x, c1.y+1, )]      -- influence on ↓ cell
      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 $\Delta\text{state} :: \text{int}$, $\text{agg} \equiv \text{SUM}$ (*i.e.*, $z \equiv 0$, $\oplus \equiv +$):

1 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

2 Table **infs**

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

3 Table **agg**

| x | y | Δstate |
|---|---|----------------------|
| 1 | 3 | +2 |
| 1 | 4 | 0 |
| 2 | 2 | -5 |

apply to current cell
states using \oplus to *
find next generation

Working Around the Linear Recursion Restriction

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

- Work around¹⁰ linearity restriction for recursive table `T`:
 - ① Use non-recursive `WITH` to introduce new name \bar{T} for `T`,
 - ② refer to \bar{T} as often as needed.

```
(WITH  $\bar{T}$ (...) AS (TABLE T)                                -- ①
  -- original recursive query  $q$  ⬇
  SELECT ...
  FROM   ...,  $\bar{T}$  AS t1, ...,  $\bar{T}$  AS t2, ...                -- ②
)
```

¹⁰ This is closer to a hack than conceptual beauty. Due to SQL's scoping rules, however, we may choose $\bar{T} = T$ such that the original query may be left untouched.

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

  (WITH sim(iter,x,ground,water,kinetic) AS (TABLE sim)
   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   sim AS s0
          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   sim AS s1
                   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 < <iterations>
  )
)
SELECT s.iter, s.x, s.ground, s.water
FROM   sim AS s
ORDER BY s.iter, s.x;

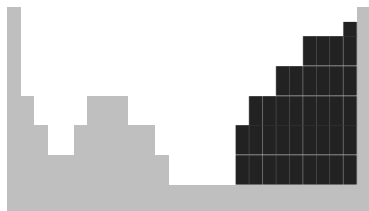
```

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

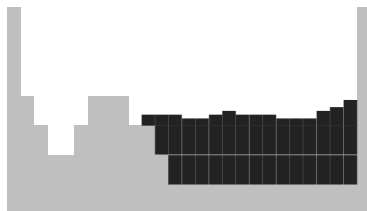
- Use **CASE ... WHEN ... THEN ... END** to implement conditional rules.
- Use windows to access cell neighborhood.
- Use array concatenation (**||**) to implement sequences of rules.

Liquid Flow (First 275 Intermediate Simulation States)

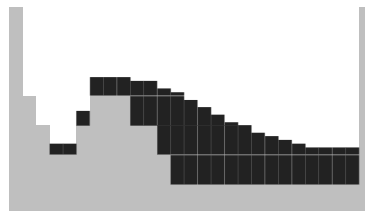
iteration #0



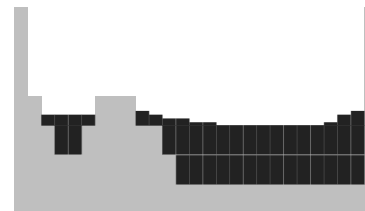
iteration #25



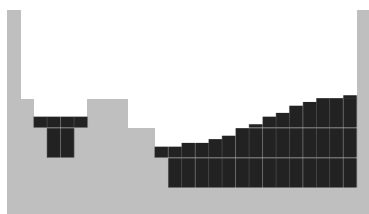
iteration #50



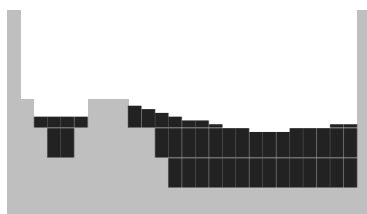
iteration #75



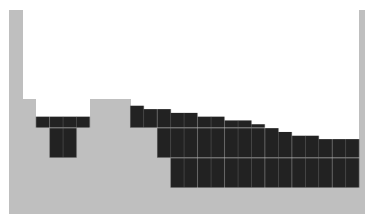
iteration #100



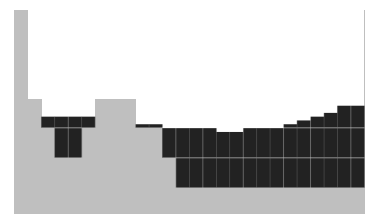
iteration #125



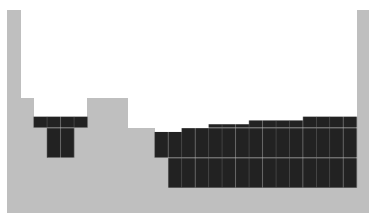
iteration #150



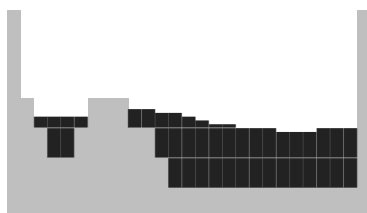
iteration #175



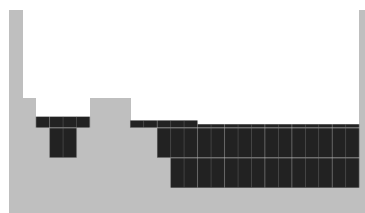
iteration #200



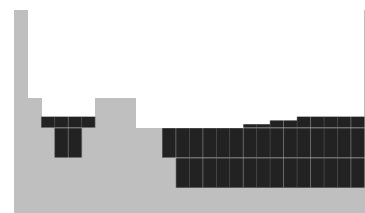
iteration #225



iteration #250



iteration #275



10 | Parsing with Context-Free Grammars

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

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

start symbol *production rule (lhs→rhs)*

\Downarrow \Downarrow
 $Expr \rightarrow Expr \ Plus \ Term \mid Term$
 $Term \rightarrow Term \ Mult \ Fact \mid Fact$
 $Fact \rightarrow '1'$
 $Plus \rightarrow '+'$
 $Mult \rightarrow 'x'$

non-terminal

terminal

Grammar for simple arithmetic expressions:

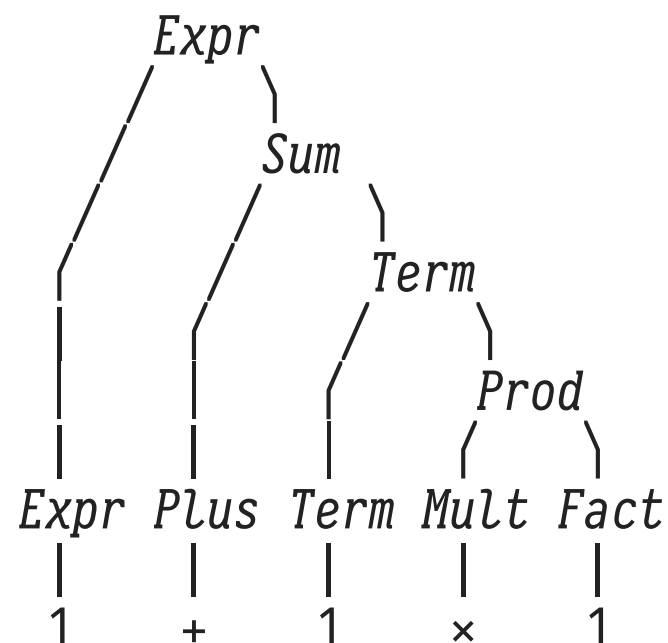
- operators $+/\times$, literal 1,
- $+/\times$ left-associative,
- op precedence: $\times > +$.

🔧 Chomsky Normal Form and Parse Trees

We consider grammars in **Chomsky Normal Form** only: rules read $lhs \rightarrow terminal$ or $lhs \rightarrow non-terminal\ non-terminal$.

$Expr \rightarrow Expr\ Sum$
 $Expr \rightarrow Term\ Prod$
 $Expr \rightarrow '1'$
 $Term \rightarrow Term\ Prod$
 $Term \rightarrow '1'$
 $Sum \rightarrow Plus\ Term$
 $Prod \rightarrow Mult\ Fact$
 $Fact \rightarrow '1'$
 $Plus \rightarrow '+'$
 $Mult \rightarrow '\times'$

Parse tree for input 1+1×1:



🔧 A Tabular Encoding of Chomsky Grammars

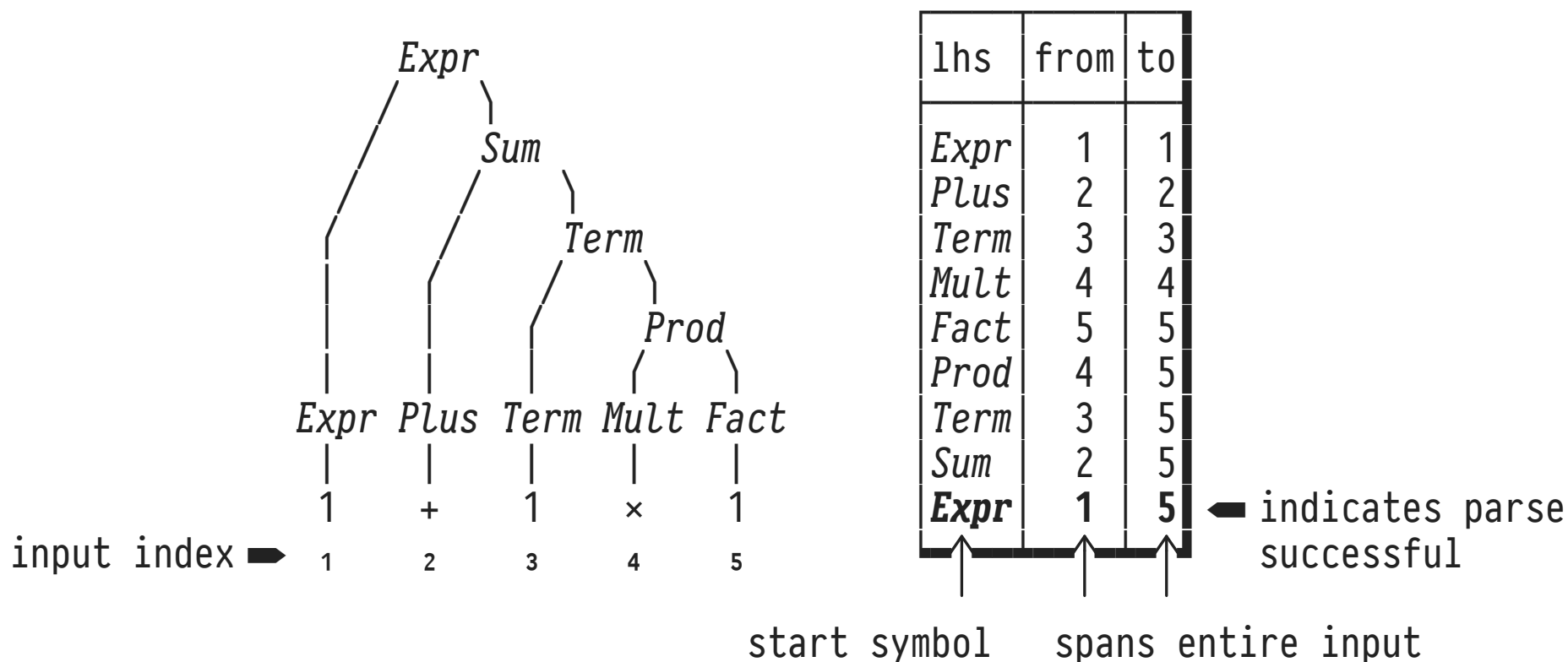
Simple encoding of the sample arithmetic expression grammar:

| lhs | sym | rhs ₁ | rhs ₂ | start? |
|-------------|-----|------------------|------------------|--------|
| <i>Expr</i> | □ | <i>Expr</i> | <i>Sum</i> | true |
| <i>Expr</i> | □ | <i>Term</i> | <i>Prod</i> | true |
| <i>Expr</i> | 1 | □ | □ | true |
| <i>Term</i> | □ | <i>Term</i> | <i>Prod</i> | false |
| <i>Term</i> | 1 | □ | □ | false |
| <i>Sum</i> | □ | <i>Plus</i> | <i>Term</i> | false |
| <i>Prod</i> | □ | <i>Mult</i> | <i>Fact</i> | false |
| <i>Fact</i> | 1 | □ | □ | false |
| <i>Plus</i> | + | □ | □ | false |
| <i>Mult</i> | × | □ | □ | 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 (here: *Expr*).

🔧 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:



Building a Tree in Layers Requires Access to the Past

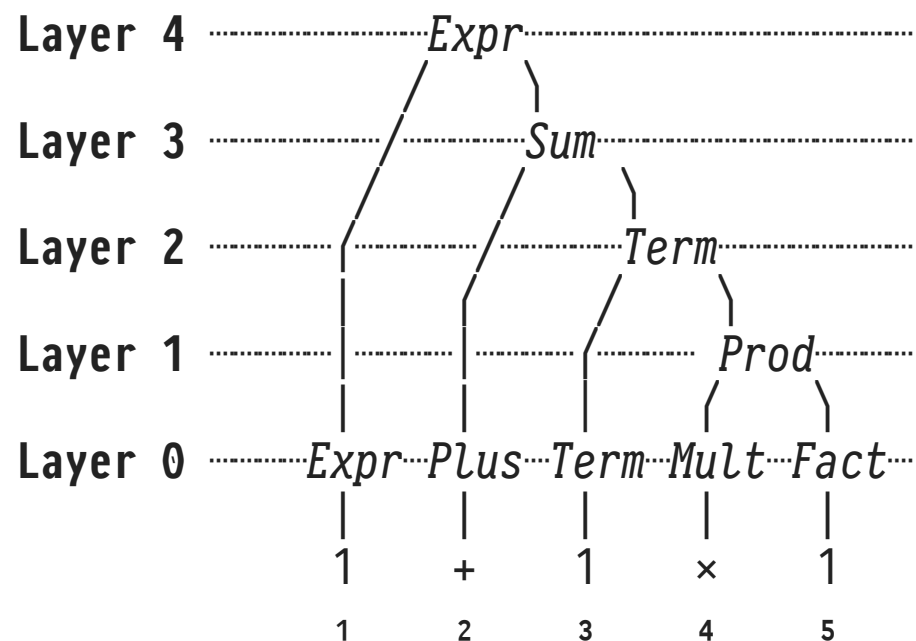


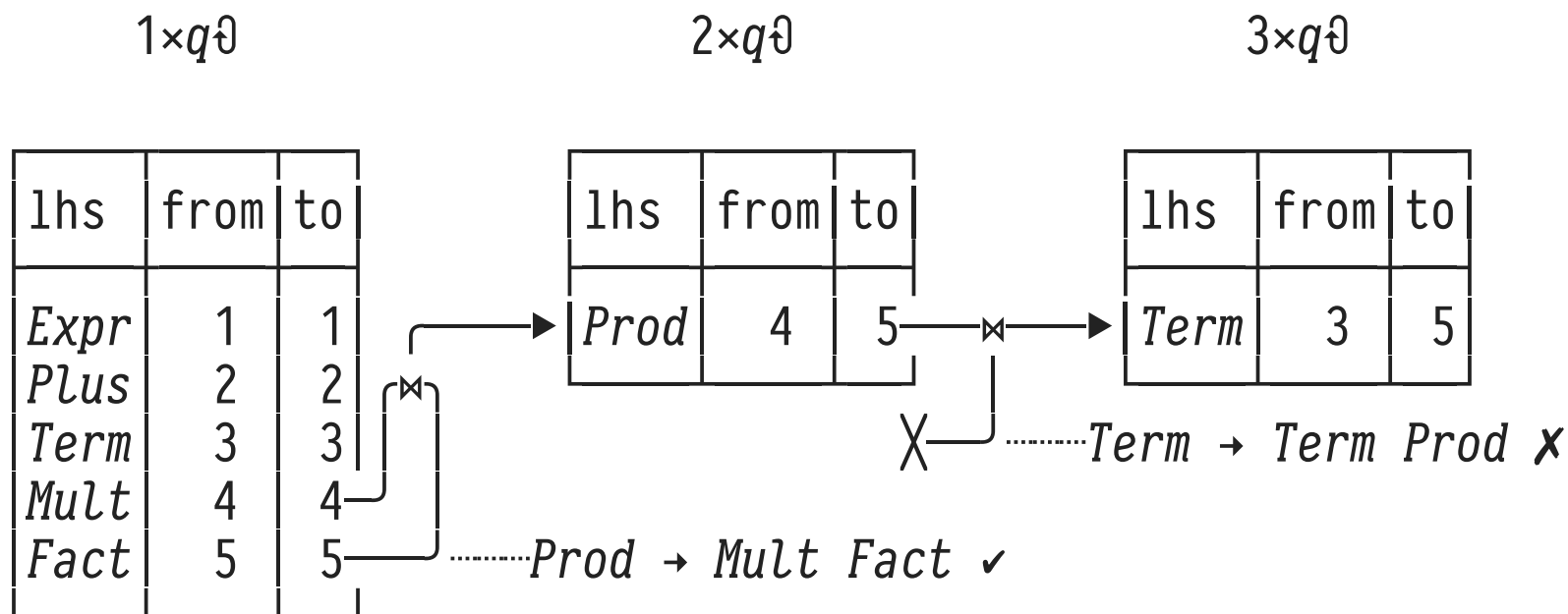
Table parse

| lhs | from | to | |
|-------------|------|----|----------------------------|
| <i>Expr</i> | 1 | 5 | iteration #4 |
| <i>Sum</i> | 2 | 5 | iteration #3 |
| <i>Term</i> | 3 | 5 | iteration #2 |
| <i>Prod</i> | 4 | 5 | iteration #1 |
| <i>Expr</i> | 1 | 1 | } found in iteration #0 |
| <i>Plus</i> | 2 | 2 | |
| <i>Term</i> | 3 | 3 | |
| <i>Mult</i> | 4 | 4 | |
| <i>Fact</i> | 5 | 5 | |

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

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 $2 \times q_0$.

Re-Injecting Early Iteration Results (SQL Template)

WITH RECURSIVE

T(iter, c₁, ..., c_n) AS (
 SELECT 0 AS iter, t.*
 FROM (q₀) AS t

-- } add column **iter** (= 0) to
 -- } result of q₀

UNION ALL

(WITH T(iter, c₁, ..., c_n) AS (TABLE T) -- non-linear recursion

SELECT t.iter + 1 AS iter, t.*
FROM (TABLE T

-- * re-inject rows in T found so far
 -- (will be kept since **iter** advances)

UNION

q₀
) AS t
WHERE p

-- original q₀ (refers to T)

-- stop condition

)
)

WITH RECURSIVE With Long-Term Memory

Rows seen in table **parse** by...

$1 \times q \emptyset$

| iter | lhs | from | to |
|------|-------------|------|----|
| 0 | <i>Expr</i> | 1 | 1 |
| 0 | <i>Plus</i> | 2 | 2 |
| 0 | <i>Term</i> | 3 | 3 |
| 0 | <i>Mult</i> | 4 | 4 |
| 0 | <i>Fact</i> | 5 | 5 |

$2 \times q \emptyset$

| iter | lhs | from | to |
|------|-------------|------|----|
| 1 | <i>Prod</i> | 4 | 5 |
| 1 | <i>Expr</i> | 1 | 1 |
| 1 | <i>Plus</i> | 2 | 2 |
| 1 | <i>Term</i> | 3 | 3 |
| 1 | <i>Mult</i> | 4 | 4 |
| 1 | <i>Fact</i> | 5 | 5 |

$3 \times q \emptyset$

| iter | lhs | from | to |
|------|-------------|------|----|
| 2 | <i>Term</i> | 3 | 5 |
| 2 | <i>Prod</i> | 4 | 5 |
| 2 | <i>Expr</i> | 1 | 1 |
| 2 | <i>Plus</i> | 2 | 2 |
| 2 | <i>Term</i> | 3 | 3 |
| 2 | <i>Mult</i> | 4 | 4 |
| 2 | <i>Fact</i> | 5 | 5 |

▨ \equiv row added by re-injection \ast

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)`:
 - `[q0]`: For each `lhs → terminal`: if `terminal` is found at index `from...to` in input, add `(lhs,from,to)` to `parse`.
 - `[q∅]`: For each pair `(lhs1,from1,to1), (lhs2,from2,to2)` in `parse × parse`:¹¹ add `(lhs3,from1,to2)` if
 1. `to1 + 1 = from2` and
 2. `lhs3 → lhs1 lhs2`.

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

```

:                                     -- A re-injection code omitted
SELECT ..., g.lhs, l."from", r."to"
FROM   grammar AS g,
       parse AS l, parse AS r      -- A non-linear recursion
WHERE  l."to" + 1 = r."from"
AND    (g.rhs1, g.rhs2) = (l.lhs, r.lhs)
:
)

```