Advanced SQL

⑥
Recursion

Winter 2025/26

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 (*i.e.*, the powerset) 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
  T₁(…) AS          ⎫
    (q₁),           ⎪
   ⋮                ⎬ Queries qⱼ may refer to all Tᵢ
  Tₙ(…) AS          ⎪
    (qₙ)            ⎭
q                   } Top-level q may refer to all Tᵢ
```

- In particular, any q_j may refer to itself (↻)! 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
  T(c₁,…,cₖ) AS            -- common schema of q₀ and q⟲(•)
   (q₀                      -- base case query, evaluated once

     UNION [ ALL ]          -- either UNION or UNION ALL

     q⟲(T)                  -- recursive query refers to T
   )                        --   itself, evaluated repeatedly
q(T)                        -- top-level post-processing query
```

- Semantics in a nutshell:

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

Semantics of a Self-Referential Query (UNION Variant)

Iterative and recursive semantics—both are equivalent:

iterate(q^{ϑ}, q_0):
  $u \leftarrow q_0$
  $i \leftarrow u$
  while $i \neq \emptyset$
    $i \leftarrow q^{\vartheta}(i) \setminus u$
    $u \leftarrow u \uplus i$
  return u
	recurse(q^{ϑ}, u):
	  if $u \neq \emptyset$ **then**
	    return $u \uplus$ recurse(q^{ϑ}, $q^{\vartheta}(u) \setminus u$)
	  else
	    return $\emptyset$

- Invoke the recursive variant with recurse(q^{ϑ}, 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 $i \in \{from, from+1, \ldots, to\}$:

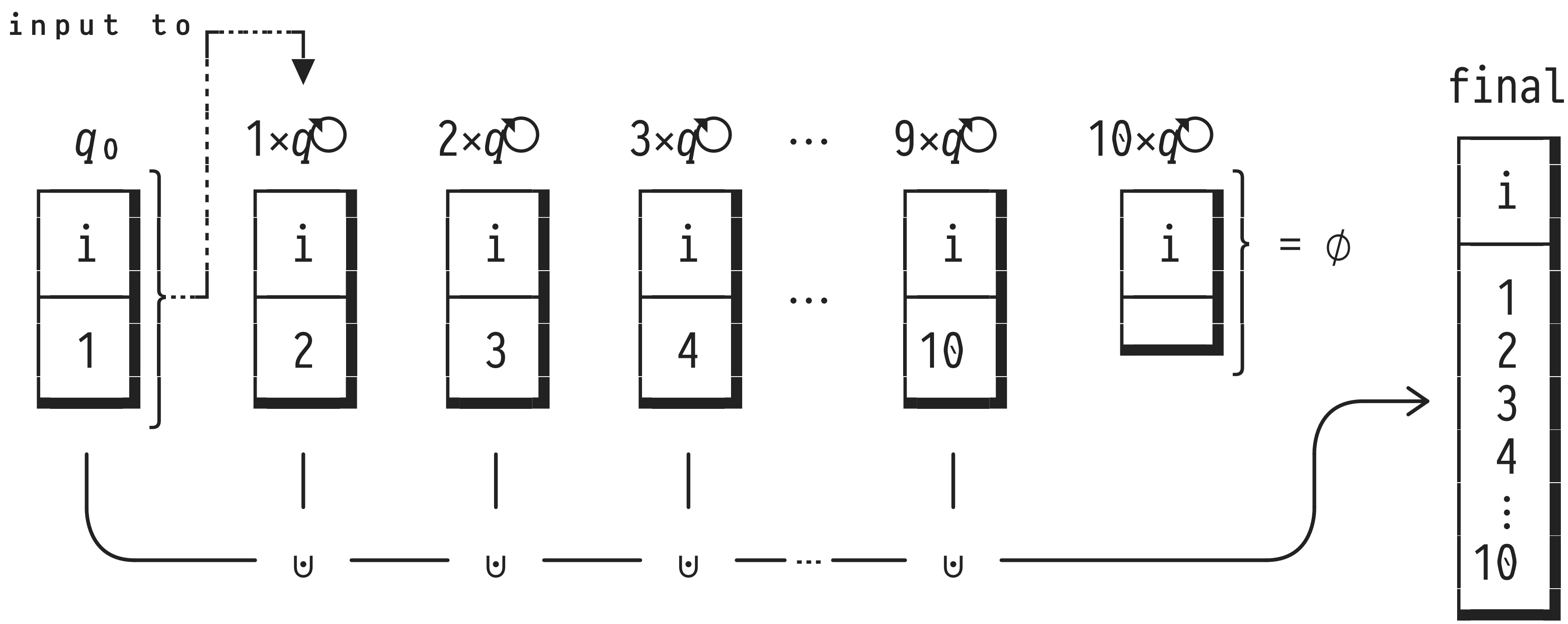
```
WITH RECURSIVE
  series(i) AS (
  ▲ VALUES (from)             -- q₀
  ┆    UNION
  ┆ SELECT s.i + 1 AS i       -- ┐                     ▲  self-
  └╌FROM╌╌╌╌▶series AS s      -- ├ q⟲(series)          ┆  reference
    WHERE  s.i < to           -- ┘                     └╌▶
)
TABLE series
```

- **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:

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



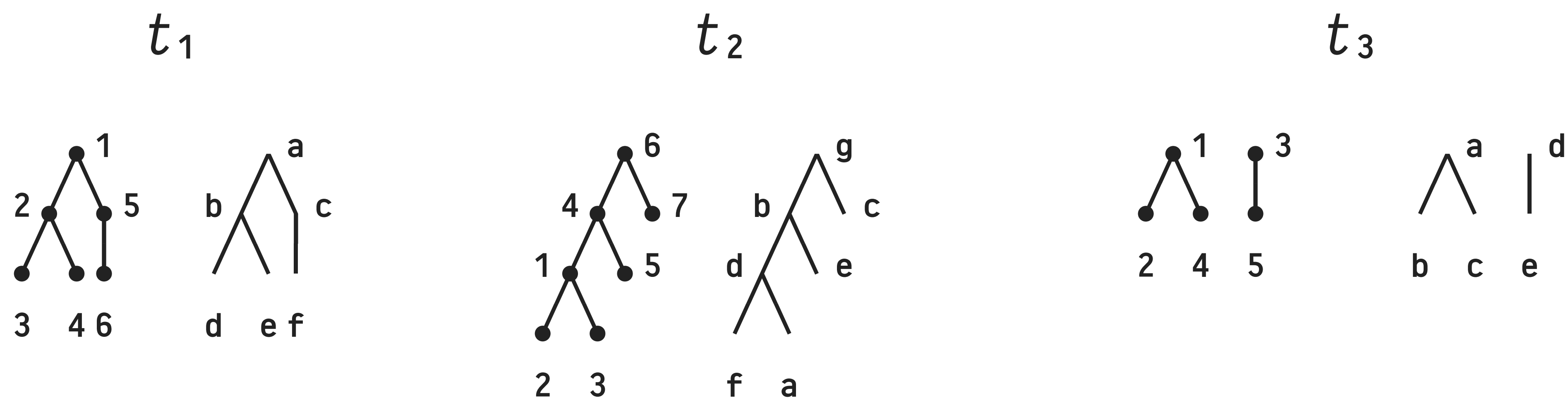
Semantics of a Self-Referential Query (UNION ALL Variant)

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

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

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

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

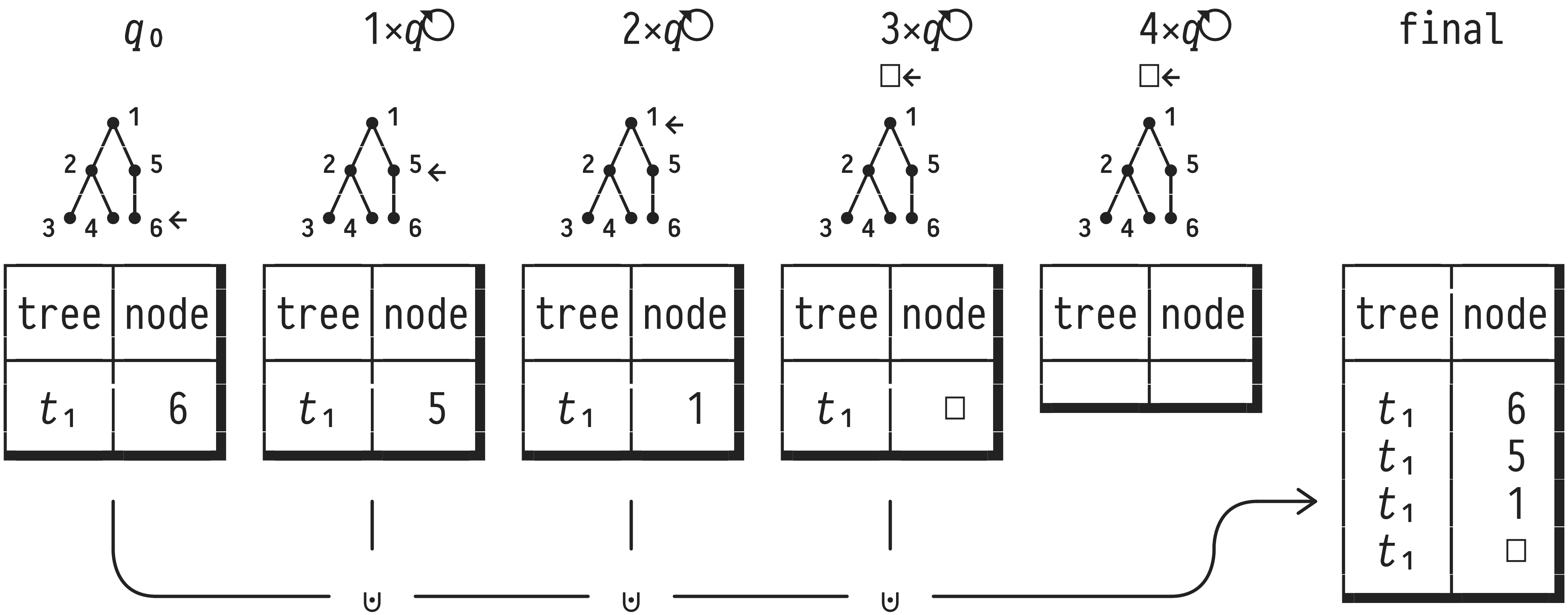
Traverse the Paths from Nodes 'f' to their Root

```
WITH RECURSIVE
  paths(tree, node) AS (
    SELECT t.tree, list_position(t.labels, 'f') AS node
    FROM   Trees AS t
    WHERE  'f' = ANY(t.labels)

      UNION

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

$(t,n) \in$ 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…



q_0:

tree	node
t_1	6

$1\times q^{\circlearrowright}$:

tree	node
t_1	5

$2\times q^{\circlearrowright}$:

tree	node
t_1	1

$3\times q^{\circlearrowright}$:

tree	node
t_1	□

$4\times q^{\circlearrowright}$:

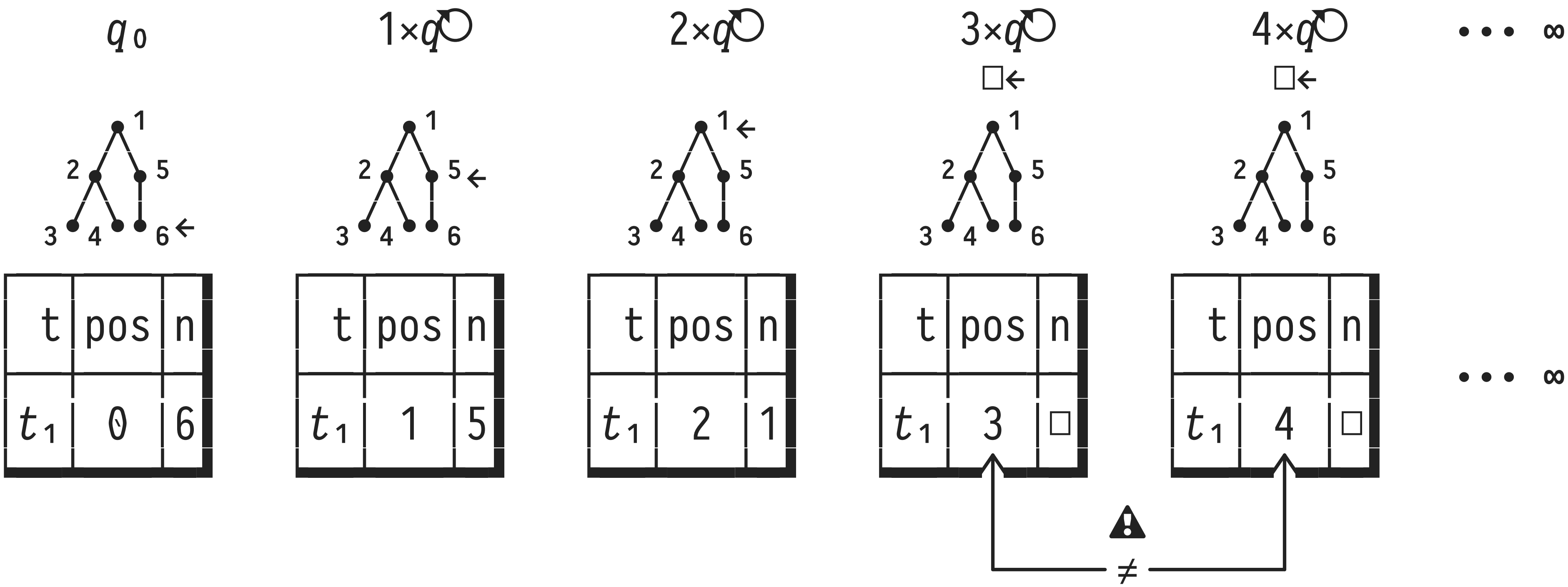
tree	node

final:

tree	node
t_1	6
t_1	5
t_1	1
t_1	□

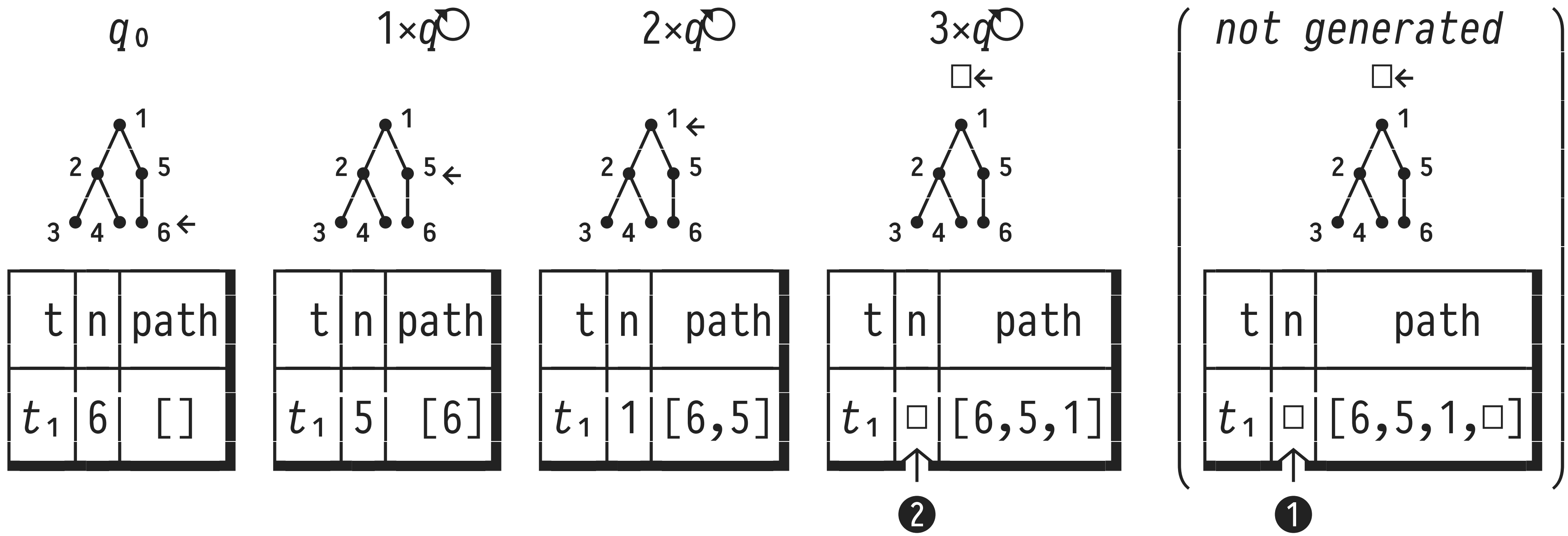
- $4\times q^{\circlearrowright}$ yields no new rows (recall: `t.parents[NULL] ≡ NULL`).

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 bugs.

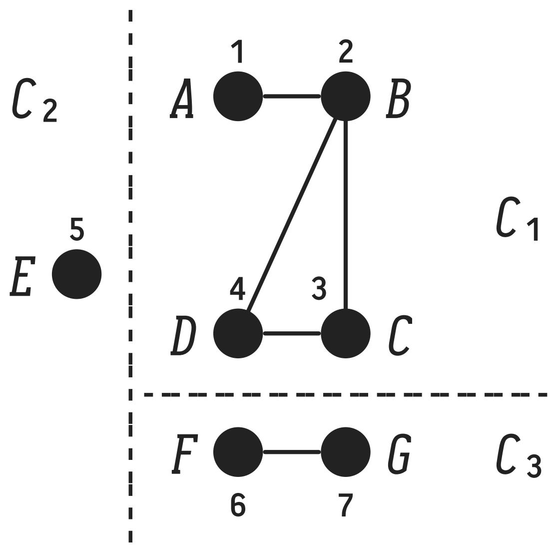
Path as Array in Tree t_1 (New Rows Trace)



❶ Ensure termination (enforce ϕ): filter on $n \neq \square$ in $q^{\circlearrowleft}$.

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

2 ¦ Connected Components in a Graph



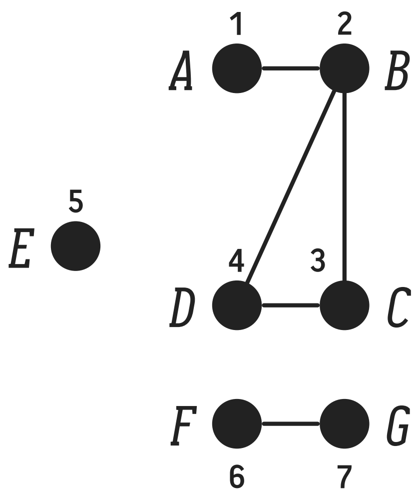
- Given an undirected graph G, find its **connected components** C_i:

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

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

Graphs are *relations* (edges). 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

$A \rightarrow B$

$\xrightarrow[\text{derive}]{}$

graph

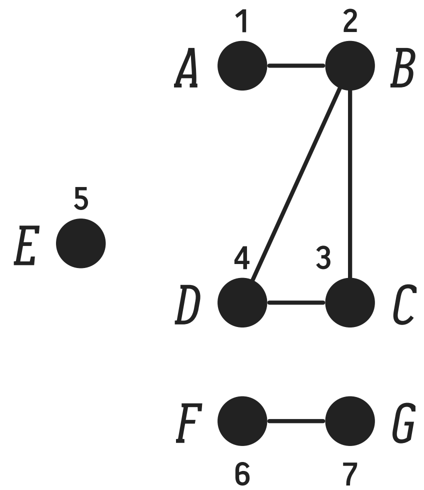
from	to
1	2
2	1
2	3
3	2
2	4
4	2
3	4
4	3
6	7
7	6

$\left.\begin{array}{l} A \leftrightarrows B \end{array}\right.$

$A — B$

- 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.

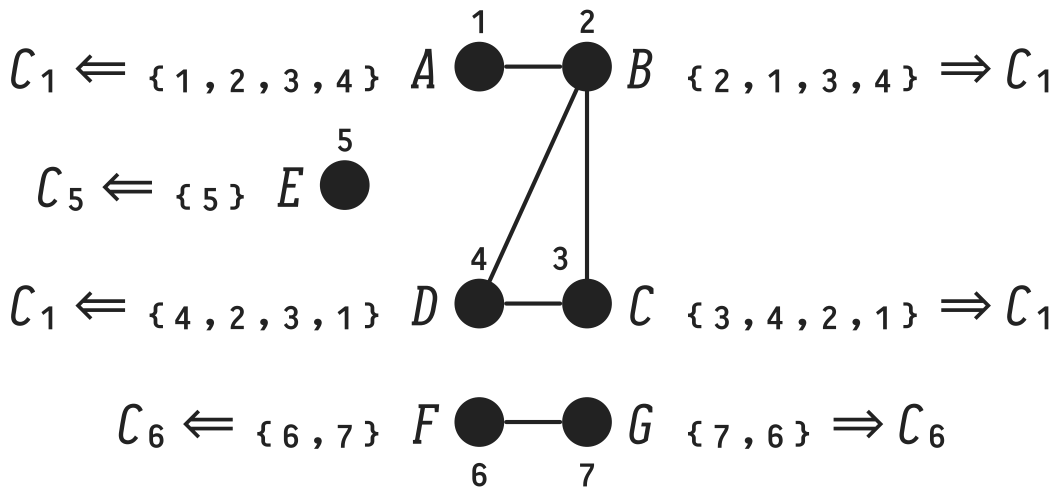
2. For each n, assign the **minimum ID** i of all front nodes as n's component C_i.

   ⇒ 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:



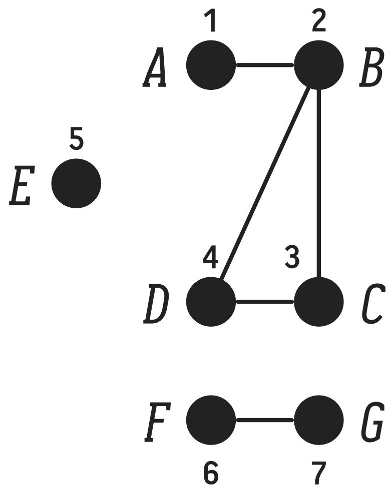
- Tasks for further post-processing:
  - Assign sane component IDs (like C_1,C_2,C_3).
  - Extract subgraphs based on components' node sets.

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

$1 \times q^{\circlearrowright}$

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

$2 \times q^{\circlearrowright}$

node	front
1	3
1	4
3	1
4	1

$3 \times q^{\circlearrowright}$

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 iteratively explore.

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

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

Set-Oriented (Bulk) Regular Expression Matching

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

input
s_1
s_2
⋮
s_n

$\longrightarrow$

input	parse?
s_1	✓
s_2	✗
⋮	⋮
s_n	✓

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

[2] Later on, we consider parsing given a context-free grammar.

Breaking Bad

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 (character), $[c_1c_2…c_n]$, 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:**

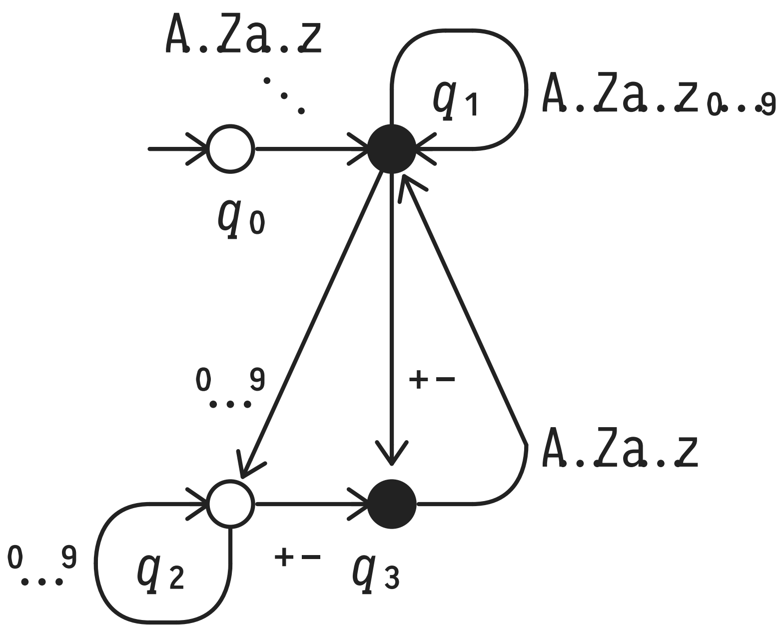


Table **fsm**

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:**

compound	step	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)

```sql
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      --   state 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    contains(f.labels, left(m.input, 1))
)
```

Matching Progess (by Compound / by Step)

❶ 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

=

❷ 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 $q^{\circlearrowleft}(i) \setminus u$ in iterate($q^{\circlearrowleft}$, q_0): $q^{\circlearrowleft}(i) \cap u = \emptyset$.

- $q^{\circlearrowleft}$ **is guaranteed to evaluate to $\emptyset$ 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 (contains(f.labels, left(m.input, 1)) yields false).

4 Recursive Array Processing: Solving Sudoku[3] 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 $\in \{1,\ldots,9\}$ such that

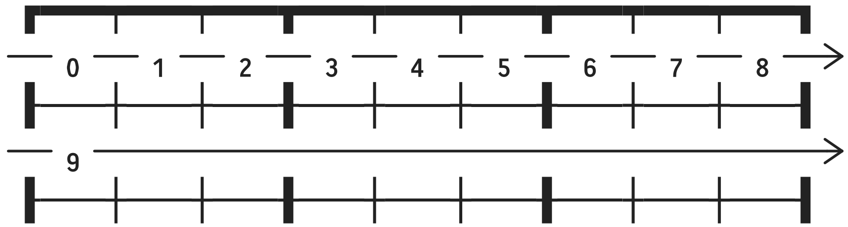
  1. no 3×3 box and
  2. no row or column

  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 ∈ {0,…,9}, with 0 ≡ blank.

- Derive **row/column/box index** of a cell from its array index 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}
  - Box of c: `((c // 3) % 3) * 3 + (c // 27) * 27` ∈ {0,3,6,27,30,33,54,57,60} (top-right cell)
- (Clunky—But: relational encodings of grids upcoming.)

Finding All Puzzle Solutions (Query Plan)

board	blank
[5,3,0,0,7,…] ↑ 2	b = 2 ∈ {0,…,80} ∪ {NULL}

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 is NULL.
2. In each iteration, **fill in all digits** ∈ {1,…,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, list_position(i.board, 0)-1 AS blank
 FROM   input AS i               --       ↑
                                 -- encodes blank

   UNION ALL

 SELECT s.bd[1:s.b] || [fill] || s.bd[s.b+2:81] AS board,
        list_position(board, 0)-1                AS blank
 FROM  sudoku AS s(bd, b), generate_series(1,9) AS _(fill)
                         --
                         --   try to fill in all 9 digits
 WHERE s.b IS NOT NULL AND NOT EXISTS (
   SELECT NULL
   FROM   generate_series(1,9) AS __(o)
   --
   --  there are 9 cells in the row/column/box of s.b
   WHERE  fill IN (‹digits in row/column/box of s.b at offset o›))
)
```

5 ¦ How SQL Can Tackle Problems in Machine Learning[4]

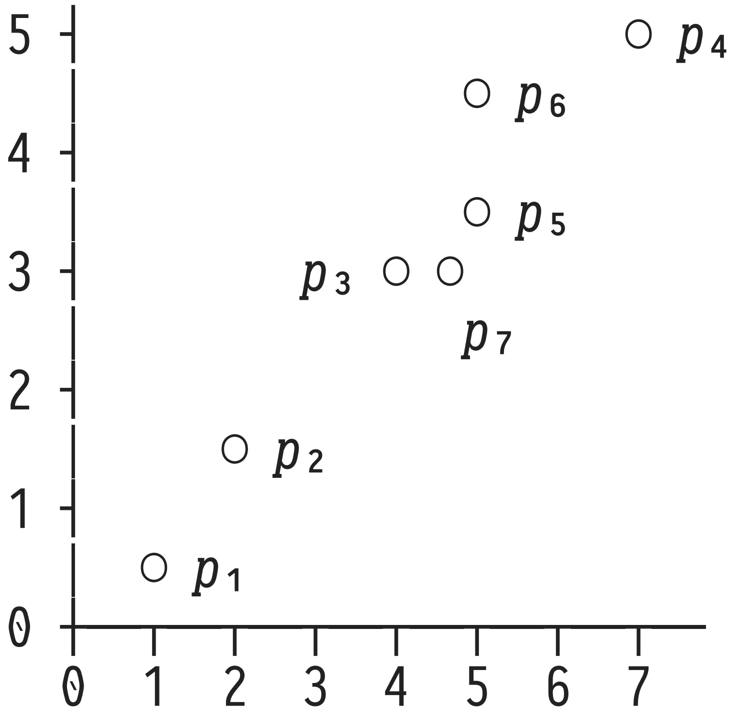
Most sizable *source data* for Machine Learning (ML) problems reside **inside** database systems. Actual *ML algorithms* are predominantly implemented **outside** the DBMS—Python, R, MatLab—however:

- Involves data serialization, transfer, and parsing. 👎
- Main-memory based ML libraries and programming frameworks may be 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.

[4] I apologize for the hype vocabulary.

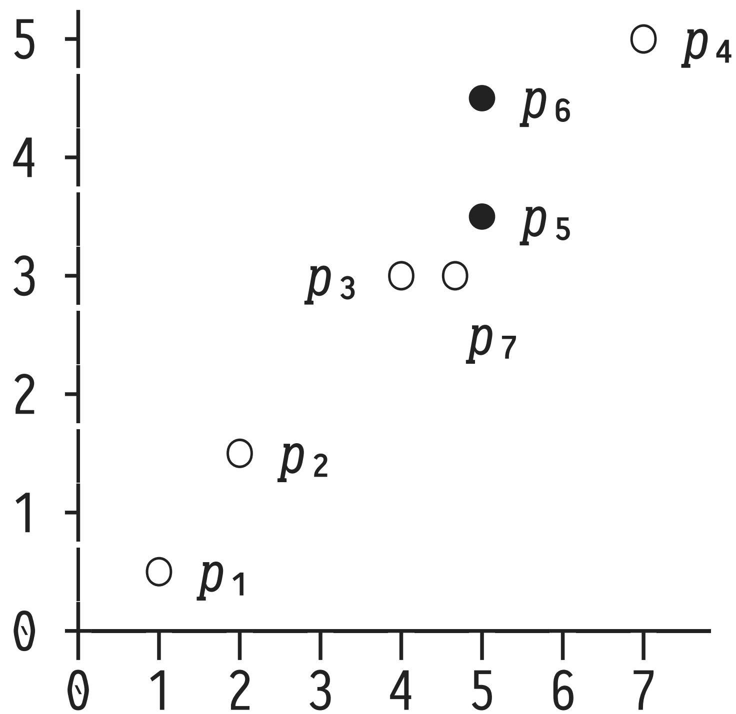
6 K-Means Clustering



- **Goal:** Assign each n-dimensional point p_i to one of ***k* clusters** ($k \geqslant 2$ 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 points p_i to the cluster with the 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	(1.0, 1.0) :: point
2	(2.0, 1.5) :: point
⋮	⋮

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
USING SAMPLE [k ROWS | n%]  -- pick k rows | n% of all rows
```

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

Invariant:

<u>iter</u>	<u>point</u>	loc	cluster
i	p	l	c

Table k_means

- In iteration i, point p (at location l) has been assigned to cluster c.
  - Iteration 0 will use a sample table of k random points as cluster centroids. Iteration $i > 0$ will determine the centroids based on the point-to-cluster assignment found in iteration i-1.

K-Means: Core of the SQL Code

```
WITH RECURSIVE
⋮
k_means(iter,point,loc,cluster) AS (
  ⋮ -- ‹iteration 0 (initialization)›
    UNION ALL
  (WITH clusters(cluster,centroid) AS (
    -- 2. Update: find new cluster centers
    SELECT k.cluster, (avg(k.loc.x), avg(k.loc.y)) AS centroid
    FROM   k_means AS k
    GROUP BY k.cluster
   )
   -- 1. Assignment: (re-)assign points to clusters
   SELECT k.iter + 1 AS iter, k.point, k.loc,
          (SELECT arg_min(c.cluster, dist(k.loc, c.centroid))
           FROM   clusters AS c) AS cluster
   FROM   k_means AS k
   WHERE  k.iter < ‹iterations›
  )
)
```

SQL Notes and Grievance (1)

- We first deconstruct and later reconstruct the 2D point locations `k.loc :: point` for centroid computation:

```
… (avg(k.loc.x), avg(k.loc.y)) :: point …
```

- Wanted: aggregate `avg() :: bag(point) → point`.

  In some RDBMSs, we can build **user-defined aggregates.**[5]

[5] For PostgreSQL, 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?
  - To find the point-to-cluster assignment in iteration *i*, need information about *all* points in *all* clusters found in iteration *i*-1 (to determine the clusters' centroid). UNION semantics only provides the most recently changed assignments in table k_means.
  - Strictly increasing iter counter guarantees that all rows in k_means change in every iteration ⇒ good 👍, but endless recursion 👎.

💡 **Iteration constructs** that provide access to *all* rows of the previous iteration, changed or not. (U TÜ research 👍)

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

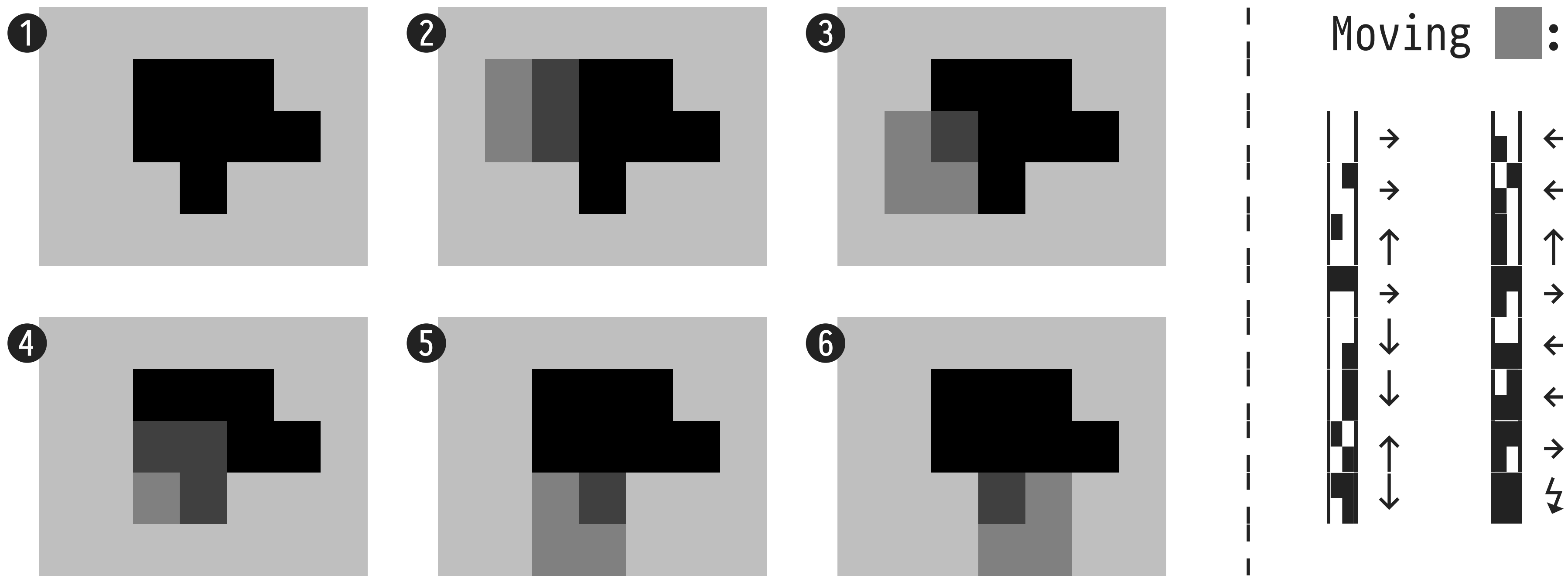
SQL provides a family of constructs to encode the **logic** (in the sense of **control flow**) of algorithms:

1. Obviously: WHERE p, HAVING p, QUALIFY p
2. q_1 UNION ALL q_2 UNION ALL ⋯ UNION ALL q_n in which the q_i contain guards (mutually exclusive predicates p_i) 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 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) →, ▘ 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 map of 2×2 squares [table squares].

3. Walk around shape and iteratively fill table march(x,y):
   - [q_0]: Start with $(x,y) = (1,1) \in$ march.
   - [$q_{\circlearrowright}$]: 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 m.x + (d.dir).Δx AS x, m.y + (d.dir).Δy AS y
 FROM   march AS m, squares AS s,
        directions AS d,                                  } *
 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)[6] 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.

[6] 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)

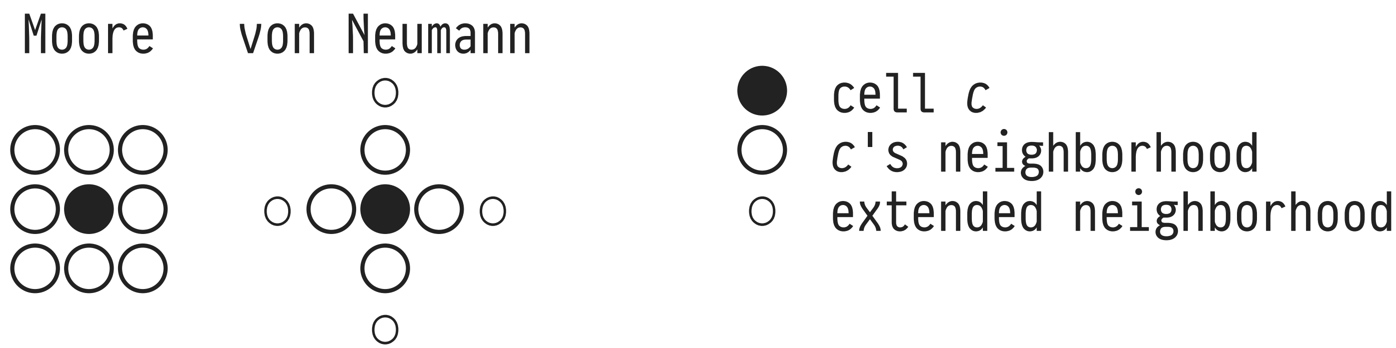
[Fluid simulation]

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 — Variant 1

- Excerpt of code in *q*⟲ (computes next generation of grid), access the 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  neighbors(c,n)   -- ← e.g. Moore neighborhood
  GROUP BY c.x, c.y, c.cell
  ⋮
)
```

- This a suitable CA core (*f*, *agg* encode CA rules).
- **NB.** *q*⟲ refers to recursive table ca *more than once*.

Life — SQL Encoding of Rules (Variant 1)

- $q^{\circlearrowleft}$ uses non-linear recursion over table life:

```
WITH RECURSIVE
life(gen,x,y,cell) AS (
 ⋮
 SELECT l.gen + 1 AS gen, l.x, l.y,
        CASE (l.cell, sum(n.cell))
           --   (c, p): c ≡ state of cell, p ≡ # of live neighbors
           WHEN (1, 2) THEN 1 -- c lives on
           WHEN (1, 3) THEN 1 -- c lives on
           WHEN (0, 3) THEN 1 -- reproduction
           ELSE             0 -- under/overpopulation
        END AS cell
 FROM   life AS l, life AS n
 WHERE  abs(l.x - n.x) <= 1         --    neighbors(l,n)
 AND    abs(l.y - n.y) <= 1         --    (Moore neighborhood)
 AND    (l.x, l.y) <> (n.x, n.y)    --
 GROUP BY l.gen, l.x, l.y, l.cell
)
```

Interlude: WITH RECURSIVE — Syntactic Restrictions

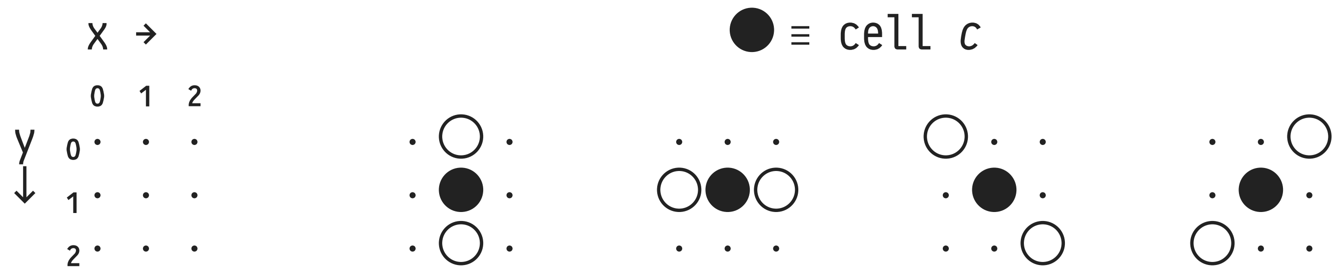
Some RDBMSs **syntactically restrict** WITH RECURSIVE queries, in particular the **references to the recursive table** T:

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 — Variant 2

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



frame ≡

	c neighbors above/below	c neighbors left/right	diagonal ↘	diagonal ↙
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
EXCLUDE CURRENT ROW -- ← omit center *c*

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

Conway's Game of Life

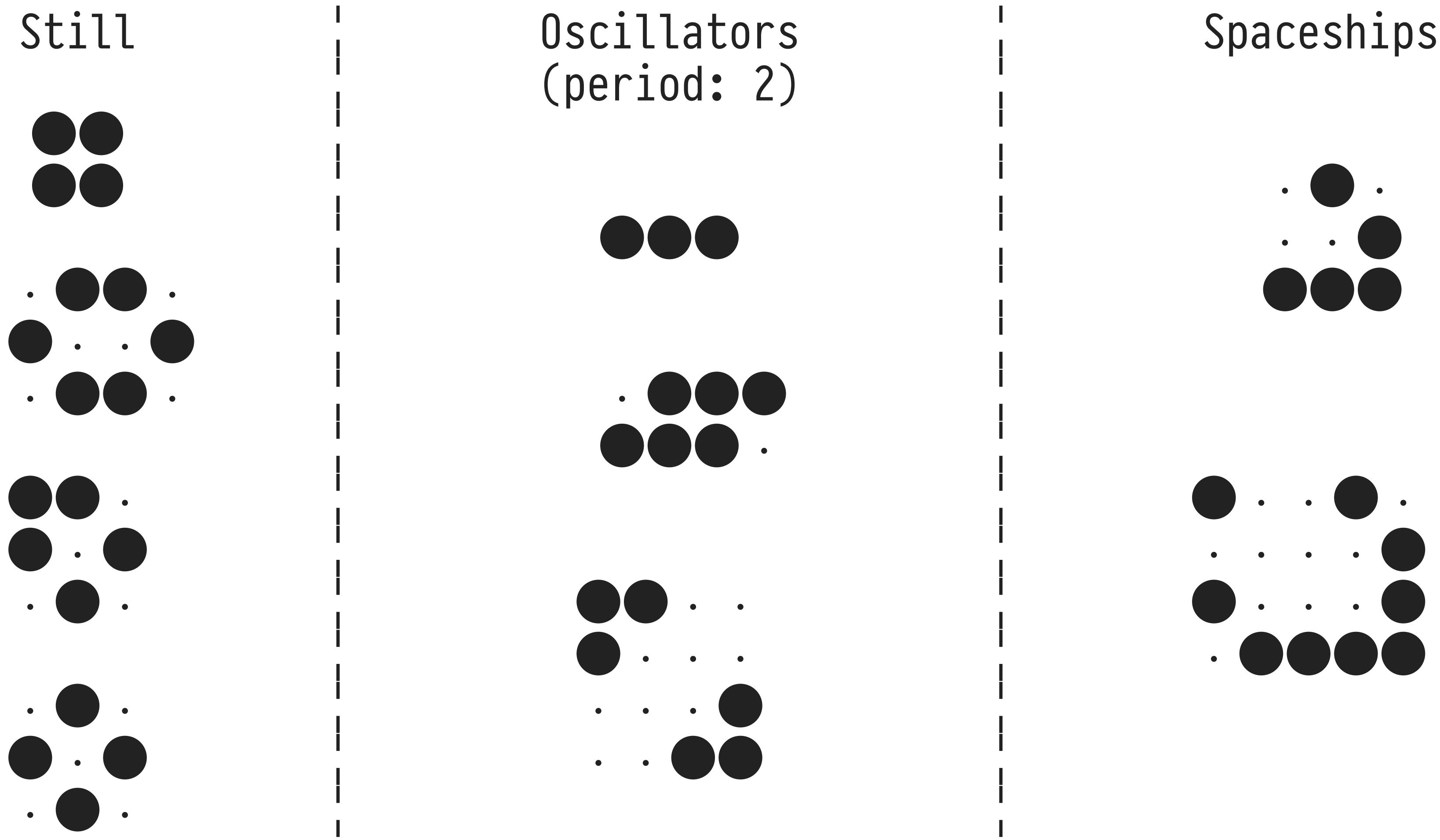
Life[7] simulates the evolution of cells c (state: either *alive* or *dead*) based on the population count $0 \leqslant p \leqslant 8$ of c's Moore neighborhood:

1. If c is alive and $p < 2$, c dies (underpopulation).
2. If c is alive and $2 \leqslant p \leqslant 3$, 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.

[7] John H. Conway (✝ April 2020), column *Mathematical Games* in *Scientific American* (October 1970).

Life — A Few Notable Cell Patterns



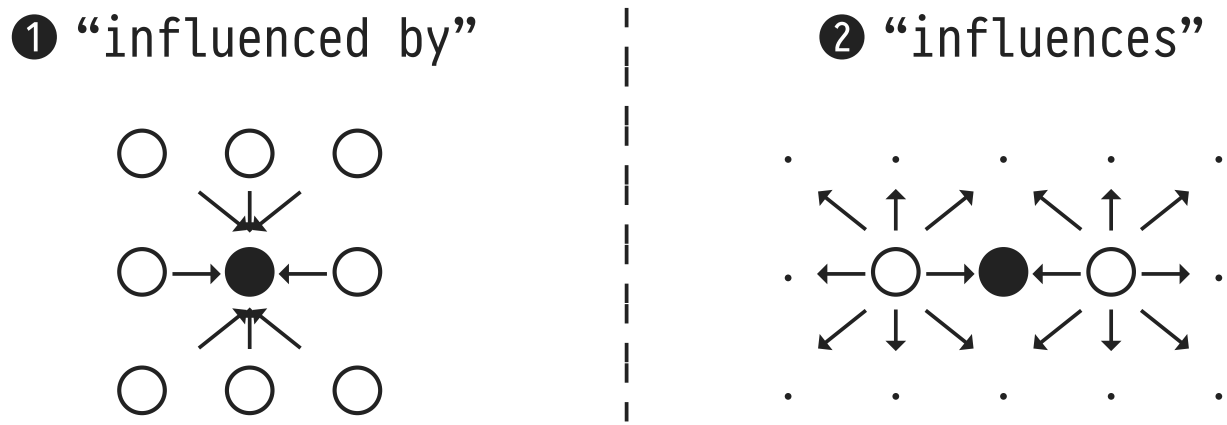
Life — SQL Encoding of Rules (Variant 2)

- q_2 uses window functions to explore vicinity of c:

```
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 ⋰›
                      )
             )
          --   (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
 ⋮
)
```

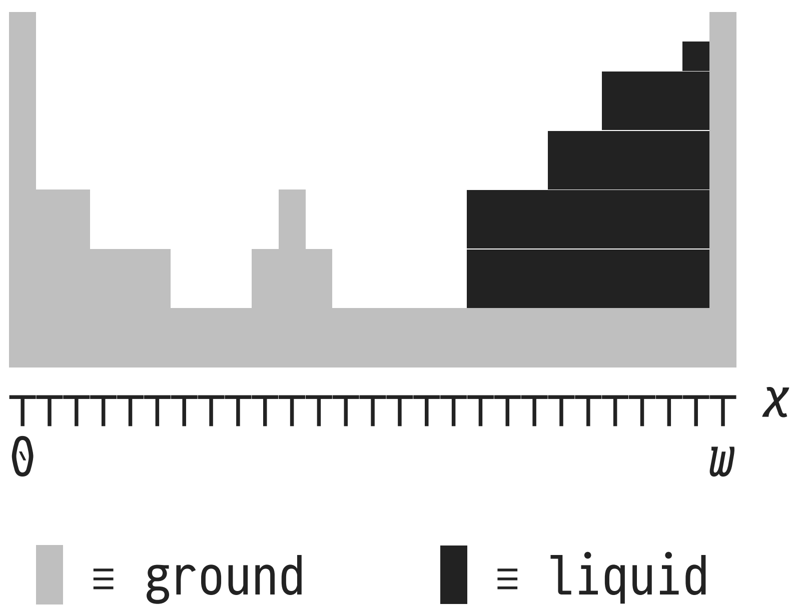
9 ┆ CA with Cells That Influence Their Neighborhood

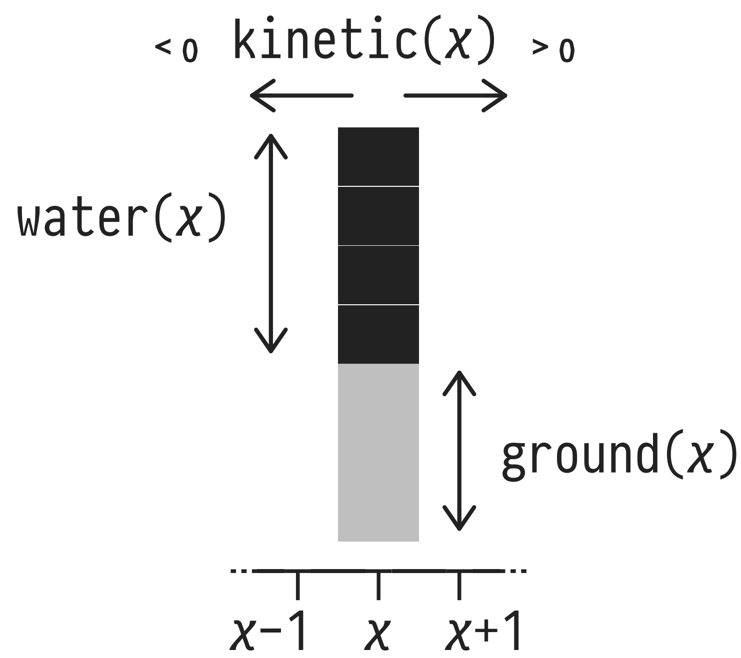
If cells assume an **active role** in influencing the next generation, this suggests a different SQL implementation.



- 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 (**kin**etic(x))

Liquid Flow: Cellular Automaton[8]

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

[8] 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
 LATERAL (                                   -- ⎫      encodes rules
          ) AS agg(x,y,Δstate)               -- ⎭      of the CA
          -- extract all influences on cell c0 (NULL if none)
          ON (c0.x, c0.y) = (agg.x, agg.y)
 WHERE    c0.iter < ‹iterations›
)
```

- No row agg(x,y,_) if cell @ x,y doesn't change state.
- Assume that z is neutral element for ⊕: $s \oplus z = s$.

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   (        
                        ) AS infs(x,y,Δstate)
        GROUP BY infs.x, infs.y
       ) AS agg(x,y,Δstate)
       -- extract all influences on cell c0 (□ if none)
       ON (c0.x, c0.y) = (agg.x, agg.y)
⋮
```

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

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 a **list 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…
              [(c1.x-1, c1.y, ▒),             -- influence on ← cell
               (c1.x, c1.y+1, ▓)]             -- influence on ↓ cell
              ELSE []
         END
      || CASE WHEN p₂ THEN
              [(c1.x, c1.y, █)]               -- influence on c1 itself
              --   ↑      ↑    ↑
              --   x      y   Δstate
              ELSE []
         END
      || …
        ) 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 **agg**

x	y	Δstate
1	3	+2
1	4	0
2	2	-5

apply to current cell
states using ⊕ to
find next generation

Liquid Flow (SQL Code)

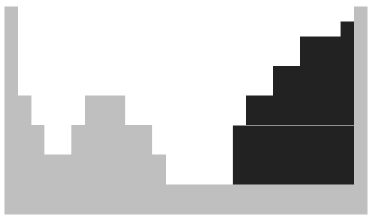
```
WITH RECURSIVE
sim(iter,x,ground,water,kinetic) AS (
  SELECT 0 AS iter, f.x, f.ground, f.water, 0.0 AS kinetic
  FROM   fluid AS f

    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   sim AS s0
          LEFT OUTER JOIN
        LATERAL (SELECT infs.xwk.x, sum(infs.xwk.Δwater) AS Δwater, sum(infs.xwk.Δkinetic) AS Δkinetic
                 FROM   (SELECT (-- flow to the left                                 --
                                 CASE WHEN ‹p1›                                       --
                                 THEN [{x:s1.x-1, Δwater:…, Δkinetic:…},              --
                                       {x:s1.x  , Δwater:…, Δkinetic:…},              --
                                       {x:s1.x-1, Δwater:…, Δkinetic:…}               --
                                      ]                                               --
                                 ELSE []                                              --
                                 END                                                  --
                                 ||                                                   --
                                 -- flow to the right                                 --
                                 CASE WHEN ‹p2›                                       --
                                 THEN [{x:s1.x+1, Δwater:…, Δkinetic:…},              --
                                       {x:s1.x  , Δwater:…, Δkinetic:…},              --
                                       {x:s1.x+1, Δwater:…, Δkinetic:…}               --
                                      ]                                               --
                                 ELSE []                                              --
                                 END                                                  --
                               ) AS influence
                         FROM   sim AS s1
                         WINDOW horizontal AS (ORDER BY s1.x)
                        ) AS inf(influence),
                        LATERAL unnest(inf.influence) AS infs(xwk)
                 GROUP BY infs.xwk.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** … **ELSE** [] **END** to implement conditional rules.
- Use windows to access cell neighborhood.
- Use list concatenation (||) to implement sequences of rules.

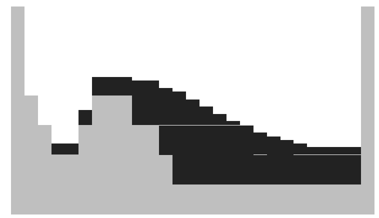
Liquid Flow (First 275 Intermediate Simulation States)

iteration #0



iteration #25

iteration #50



iteration #75

iteration #100

iteration #125

iteration #150

iteration #175

iteration #200

iteration #225

iteration #250

iteration #275

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 → [0-9]+
Plus → [+]
Mult → [×]
```

choice

non-terminal *terminal* (reg.exp.)

Grammar for simple arithmetic expressions:

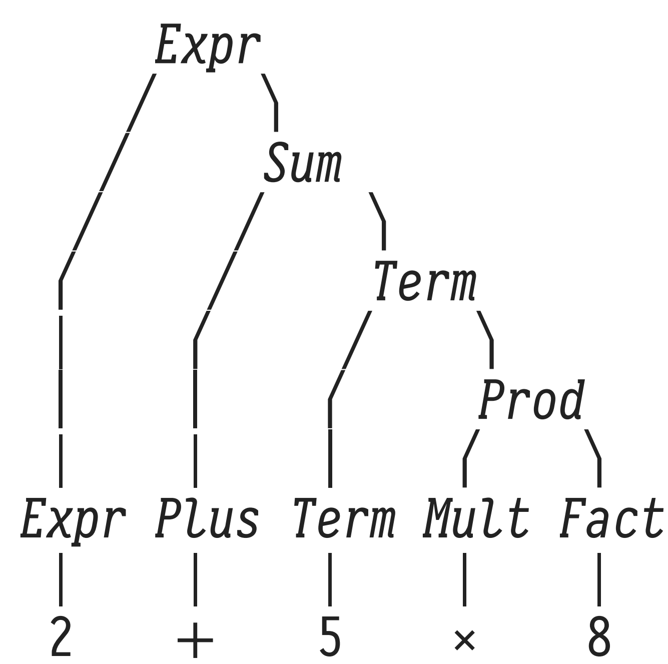
- operators +/×, literals,
- +/× left-associative,
- op precedence: × > +.

Chomsky Normal Form and Parse Trees

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

Expr → *Expr Sum*
Expr → *Term* Prod
Expr → [0-9]+
Term → *Term Prod*
Term → [0-9]+
Sum → *Plus Term*
Prod → *Mult Fact*
Fact → [0-9]+
Plus → [+]
Mult → [×]

Parse tree for input 2+5×8:



A Tabular Encoding of Chomsky Grammars

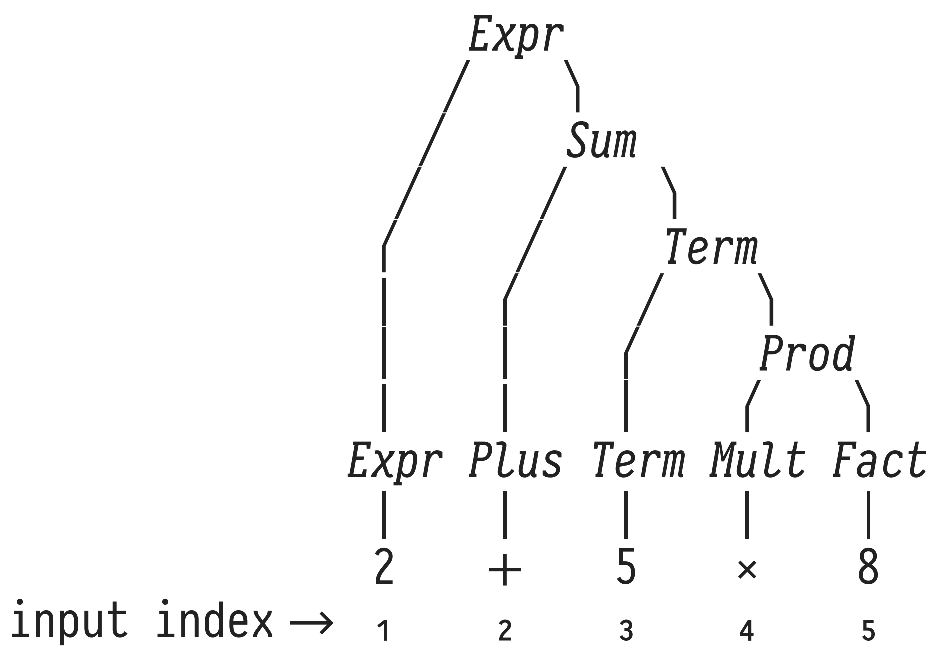
Simple encoding of the sample arithmetic expression grammar:

lhs	sym	rhs_1	rhs_2	start?
Expr	□	*Expr*	*Sum*	true
Expr	□	*Term*	*Prod*	true
Expr	[0-9]+	□	□	true
Term	□	*Term*	*Prod*	false
Term	[0-9]+	□	□	false
Sum	□	*Plus*	*Term*	false
Prod	□	*Mult*	*Fact*	false
Fact	[0-9]+	□	□	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 (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:



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

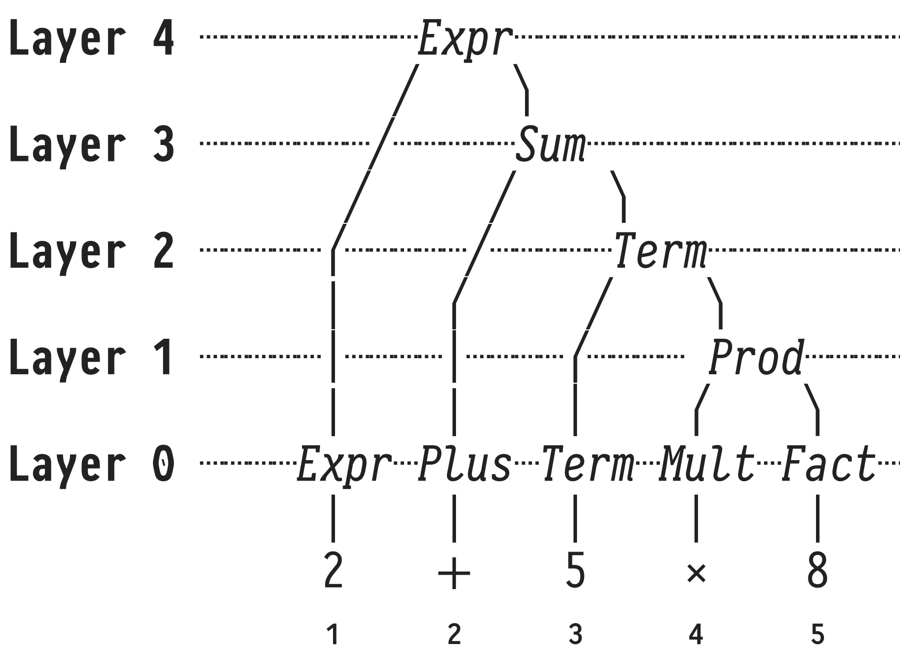


Table **parse**

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	found in iteration #0
Term	3	3	found in iteration #0
Mult	4	4	found in iteration #0
Fact	5	5	found in iteration #0

- 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…

1×q↻

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

Prod → *Mult Fact* ✓

2×q↻

lhs	from	to
Prod	4	5

Term → *Term Prod* ✗

3×q↻

lhs	from	to
Term	3	5

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

          q⟲                        -- original q⟲ (refers to 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 re-injection ⁂

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 $\mathit{lhs} \rightarrow \mathit{terminal}$: if *terminal* is found at indices *from…to* in input, add $(\mathit{lhs},\mathit{from},\mathit{to})$ to parse.
  - [$q_{\circlearrowleft}$]: For each pair $(\mathit{lhs}_1,\mathit{from}_1,\mathit{to}_1)$, $(\mathit{lhs}_2,\mathit{from}_2,\mathit{to}_2)$ in parse × parse:[9] add $(\mathit{lhs}_3,\mathit{from}_1,\mathit{to}_2)$ if
    1. $\mathit{to}_1 + 1 = \mathit{from}_2$ and
    2. $\mathit{lhs}_3 \rightarrow \mathit{lhs}_1\ \mathit{lhs}_2$.

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

Parsing Using CYK (Core SQL Code)

```
WITH RECURSIVE
tokens(tok, i) AS (
  -- row (tok,i): token tok occurs at index i in input
),
parse(…, lhs, "from", "to") AS (
  SELECT …, g.lhs, t.i AS "from", t.i AS "to"
  FROM   tokens AS t, grammar AS g
  WHERE  t.tok ~ g.sym

    UNION ALL

  ⋮                                       -- ⚠ re-injection of parse omitted
  SELECT …, g.lhs, l."from", r."to"
  FROM   grammar AS g,
         parse AS l, parse AS r           -- identify two partial parses…
  WHERE  l."to" + 1 = r."from"            -- …that are adjacent in the input
  AND    (g.rhs1, g.rhs2) = (l.lhs, r.lhs)
  ⋮                                       -- ⚠ stop condition omitted
)
```