

# Advanced SQL

---

05 — Window Functions

Summer 2024

Torsten Grust  
Universität Tübingen, Germany

# 1 | Window Functions

---

With SQL:2003, the ISO SQL Standard introduced **window functions**, a new mode of row-based computation:

| SQL Feature               | Mode of Computation                 |
|---------------------------|-------------------------------------|
| function/operator         | row → row                           |
| table-generating function | row → table of rows                 |
| aggregate                 | group of rows → row (one per group) |
| window function 🖱️        | row vicinity → row (one per row)    |

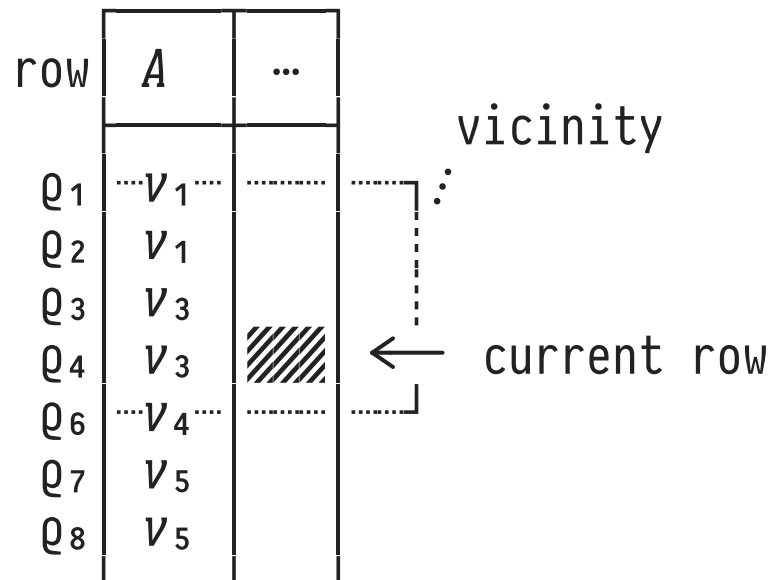
SQL Modes of Computation


## Window functions ...

- ... are **row-based**: each individual input row *r* is mapped to one result row,
- ... use the **vicinity** around *r* to compute this result row.

## Row Vicinity: Window Frames

---



- Each row is the **current row**  at one point in time.

- Row vicinity (**window**, **frame**) is based on either:

- row **position** (**ROWS** windows),
- row **values**  $v_i$  (**RANGE** windows).

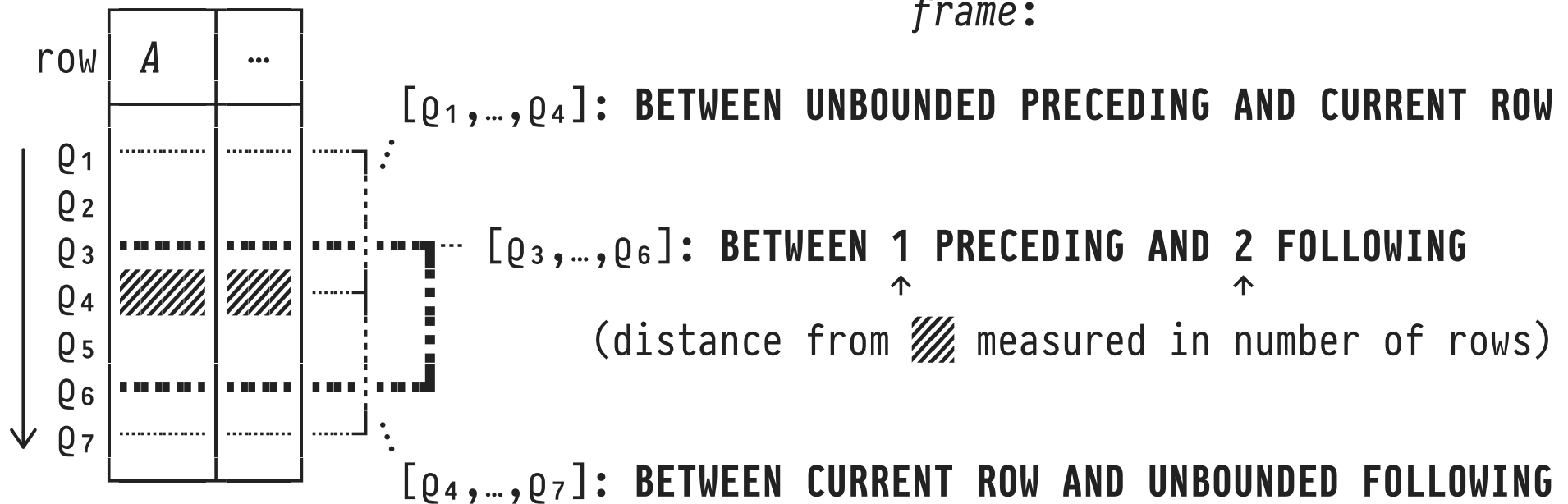
- As the current row changes, the window *slides* with it.
- ⚠ Window semantics depend on a defined **row ordering**.

# Window Frame Specifications (Variant: **ROWS**)

window function      ordering criteria      frame specification

$f$  OVER (ORDER BY  $e_1, \dots, e_n$  [ ROWS  $frame$  ])

*frame:*

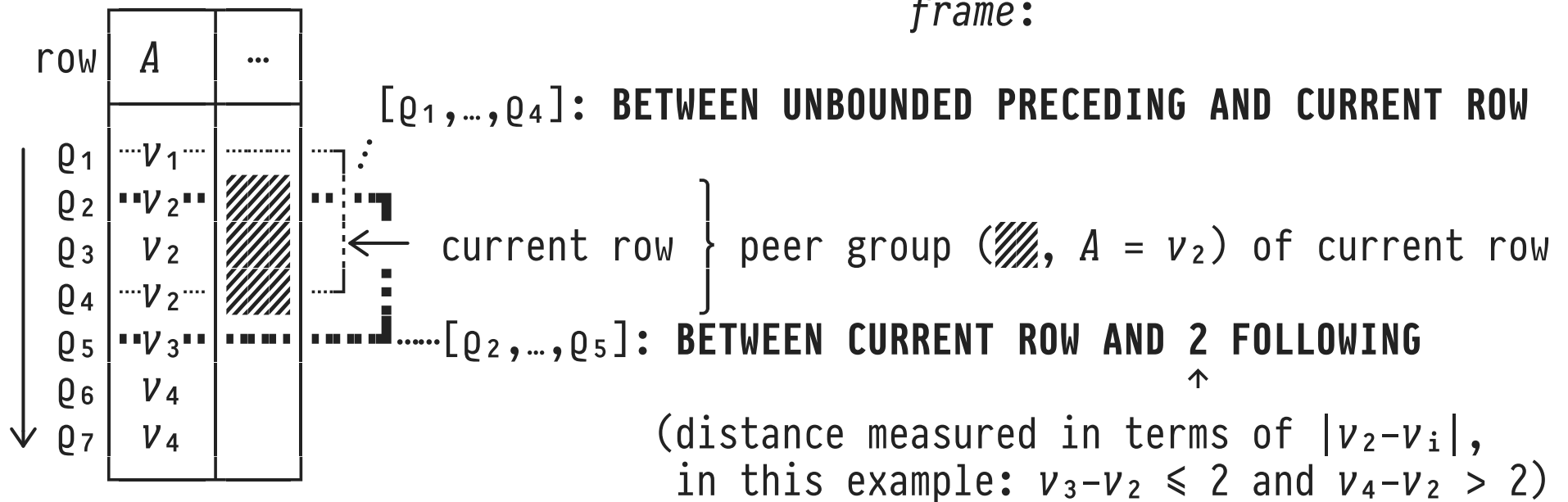


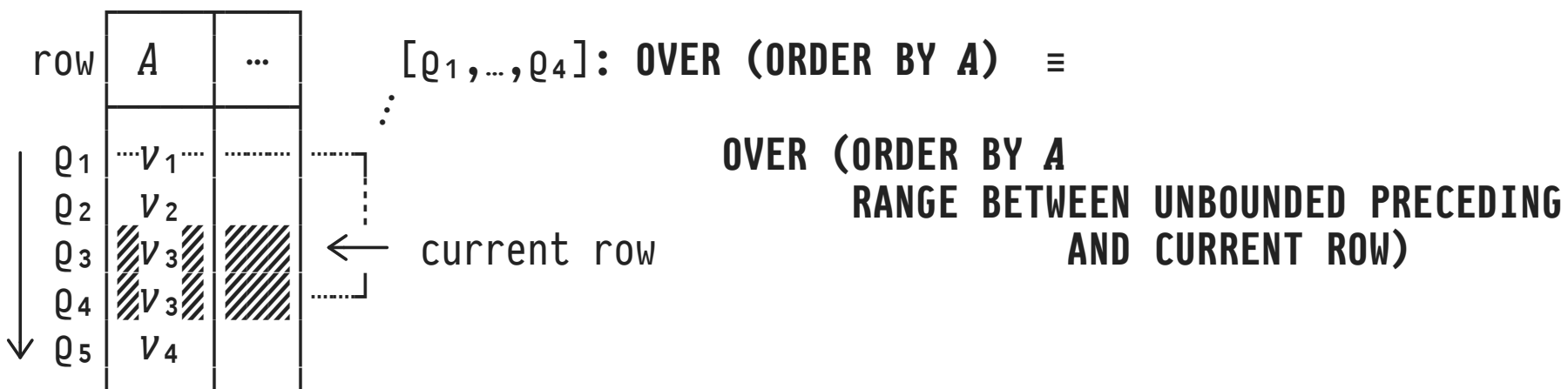
# Window Frame Specifications (Variant: **RANGE**)

window function      one criterion      frame specification

$f$  OVER (ORDER BY  $A$  [ RANGE  $frame$  ])

*frame:*





## WINDOW Frame Specifications: Exclusion<sup>1</sup>

window function

exclusion clause

$f$  OVER (ORDER BY  $A$  frame [ exclusion ])

| row            | A              | ... |
|----------------|----------------|-----|
| q <sub>1</sub> | v <sub>1</sub> |     |
| q <sub>2</sub> | v <sub>2</sub> |     |
| q <sub>3</sub> | v <sub>3</sub> |     |
| q <sub>4</sub> | v <sub>3</sub> |     |
| q <sub>5</sub> | v <sub>3</sub> |     |
| q <sub>6</sub> | v <sub>4</sub> |     |

← current row

exclusion:

**EXCLUDE NO OTHERS** [q<sub>2</sub>, q<sub>3</sub>, q<sub>4</sub>, q<sub>5</sub>]

**EXCLUDE CURRENT ROW** [q<sub>2</sub>, q<sub>3</sub>, q<sub>5</sub>]

**EXCLUDE GROUP** [q<sub>2</sub> ]

**EXCLUDE TIES** [q<sub>2</sub>, q<sub>4</sub> ]

<sup>1</sup> Q: Which *frame* would lead to a window as shown above?

## WINDOW Clause: Name the Frame

---

Syntactic ©: If window frame specifications

1. become unwieldy because of verbose SQL syntax and/or
2. one frame is used multiple times in a query,

add a **WINDOW** clause to a SFW block to **name the frame**, *e.g.*:

```
SELECT ... f OVER wi ... g OVER wj ...  
FROM ...  
WHERE ...  
⋮  
WINDOW w1 AS (frame1), ..., wn AS (framen)  
ORDER BY ...
```



## Use SQL Itself to Explain Window Frame Semantics

---

Regular **aggregates** may act as window functions *f*. All **rows in the frame will be aggregated**:

```
SELECT w.row          AS "current row",
       COUNT(*)       OVER win AS "frame size",
       list(w.row)    OVER win AS "rows in frame"
FROM   W AS w
WINDOW win AS (frame)
```

| <u>row</u>     | a | b |
|----------------|---|---|
| Q <sub>1</sub> | 1 | ● |
| Q <sub>2</sub> | 2 | ○ |
| Q <sub>3</sub> | 3 | ○ |
| Q <sub>4</sub> | 3 | ● |
| ⋮              | ⋮ | ⋮ |

Table *W*

## 🔧 Q: What is the Chance of Fine Weather on Weekends?

---

**Input:** Daily weather readings in **sensors**:

| <u>day</u> | weekday | temp | rain |
|------------|---------|------|------|
| 1          | Fri     | 10   | 800  |
| 2          | Sat     | 12   | 300  |
| ⋮          | ⋮       | ⋮    | ⋮    |

Table **sensors**

- The weather is fine on day ***d*** if—on ***d*** and the two days **prior**—the minimum temperature is above 15°C and the overall rainfall is less than 600ml/m<sup>2</sup>.
- **Expected output:**

| weekend? | % fine |
|----------|--------|
| f        | 29     |
| t        | 43     |

## 2 : PARTITION BY: Window Frames Inside Partitions

---

Optionally, we may **partition** the input table *before* rows are sorted and window frames are determined:

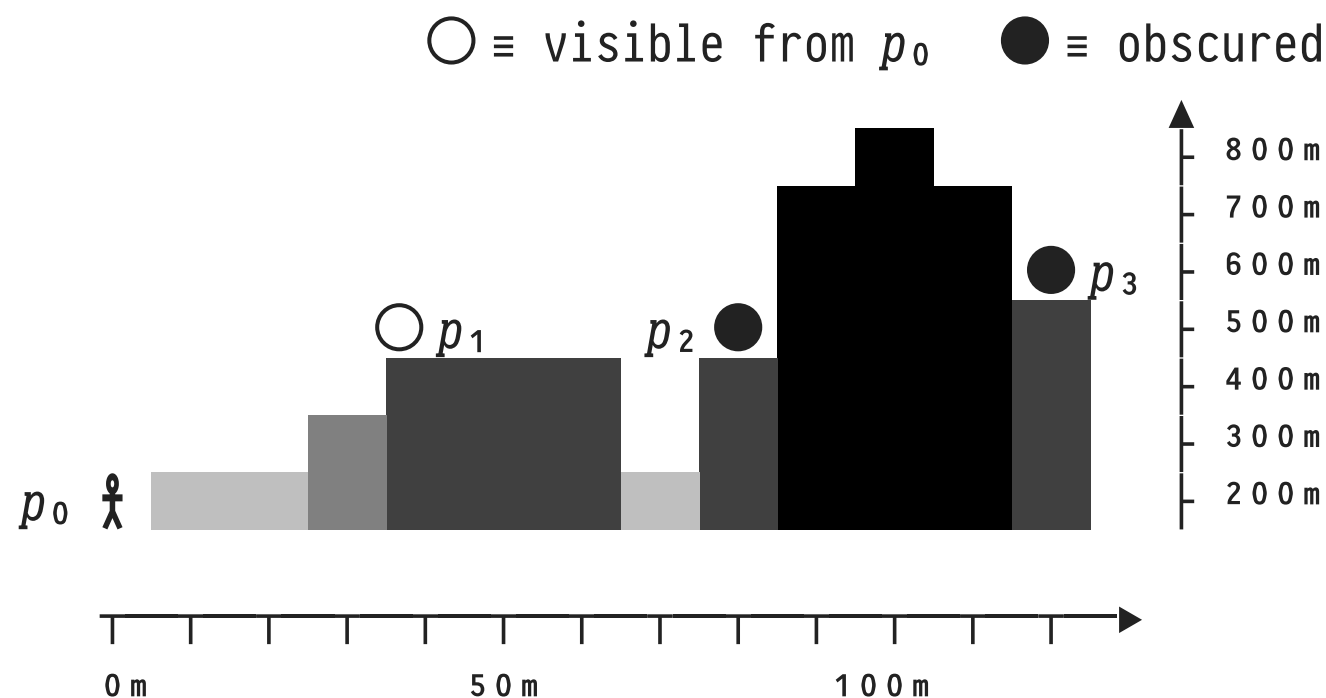
all input rows that agree on all  $p_i$  form one partition

```
f OVER ( [ PARTITION BY  $p_1, \dots, p_m$  ]  
         [ ORDER BY  $e_1, \dots, e_n$  ]  
         [ frame ] )
```

- Note:
  1. Frames **never cross partitions**.
  2. **BETWEEN ... PRECEDING AND ... FOLLOWING** respects **partition boundaries**.

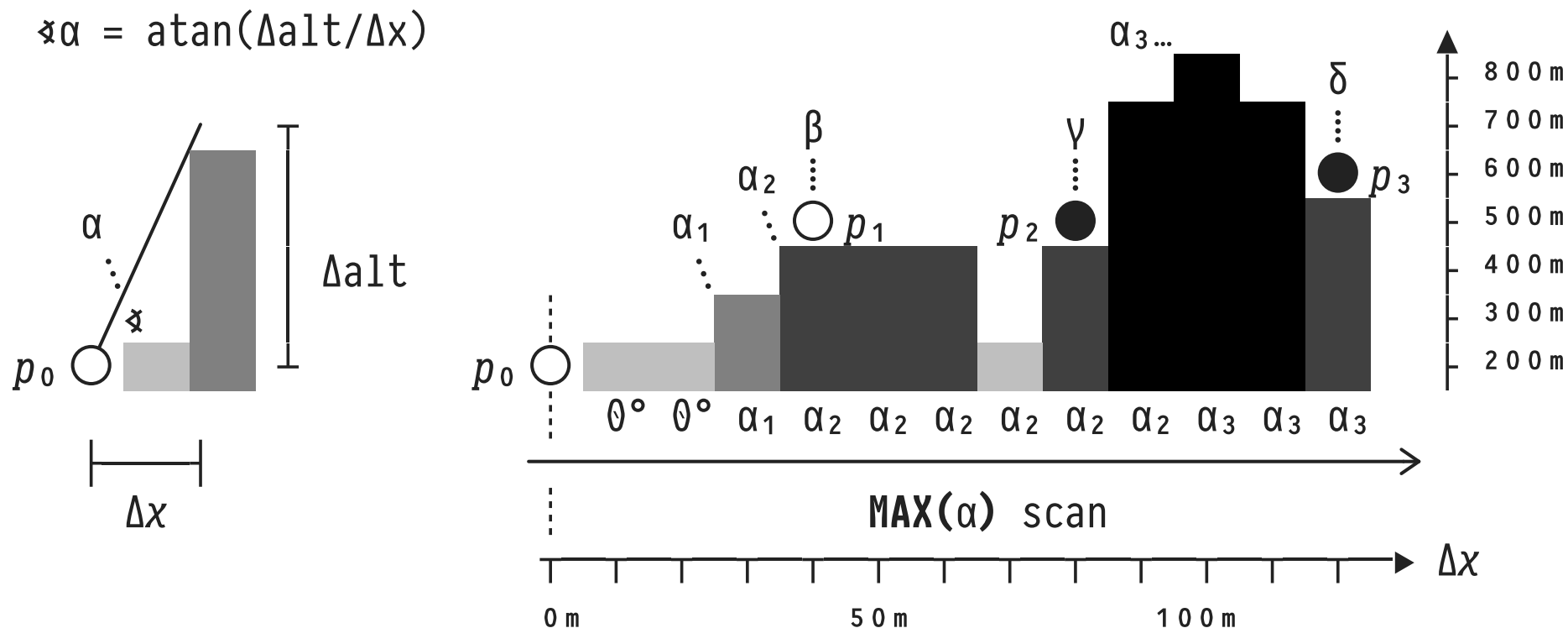
# 🔧 Q: Which Spots are Visible in a Hilly Landscape?

---



- From the viewpoint of  $p_0$  (🧑) we can see  $p_1$ , but...
  - ...  $p_2$  is **obscured** (no straight-line view from  $p_0$ ),
  - ...  $p_3$  is **obscured** (lies behind the 800m peak).

# 🔧 Q: Visible Spots in a Hilly Landscape? — A: MAX Scan!



- We have  $0^\circ < \alpha_1 < \alpha_2 < \alpha_3$  and  $\beta \geq \alpha_2$ ,  $\gamma < \alpha_2$ ,  $\delta < \alpha_3$ .  
 $p_1$  visible     $p_{2,3}$  obscured

## 🔧 Q: Visible Spots in a Hilly Landscape? — A: MAX Scan!

---

- **Input:** Location of  $p_0$  (here:  $x = 0$ ) and 1D-map of hills:

| $x$ | alt |
|-----|-----|
| 0   | 200 |
| 10  | 200 |
| ⋮   | ⋮   |
| 120 | 500 |

Table `map`

- **Output:** Can  $p_0$  see the point on the hilltop at  $x$ ?

| $x$ | visible? |
|-----|----------|
| 0   | true     |
| 10  | true     |
| ⋮   | ⋮        |
| 120 | false    |

## Q: Visible Spots in a Hilly Landscape? — A: MAX Scan!

---

**WITH**

-- ❶ Angles  $\alpha$  (in  $^\circ$ ) between  $p_0$  and the hilltop at  $x$   
 angles( $x$ , angle) **AS** (  
     **SELECT**  $m.x$ ,  
         **degrees**(**atan**(( $m.alt - p_0.alt$ ) /  
                     **abs**( $p_0.x - m.x$ ))) **AS** angle

**FROM** map **AS**  $m$

**WHERE**  $m.x > p_0.x$

),

-- ❷  $\text{MAX}(\alpha)$  scan (to the right of  $p_0$ )

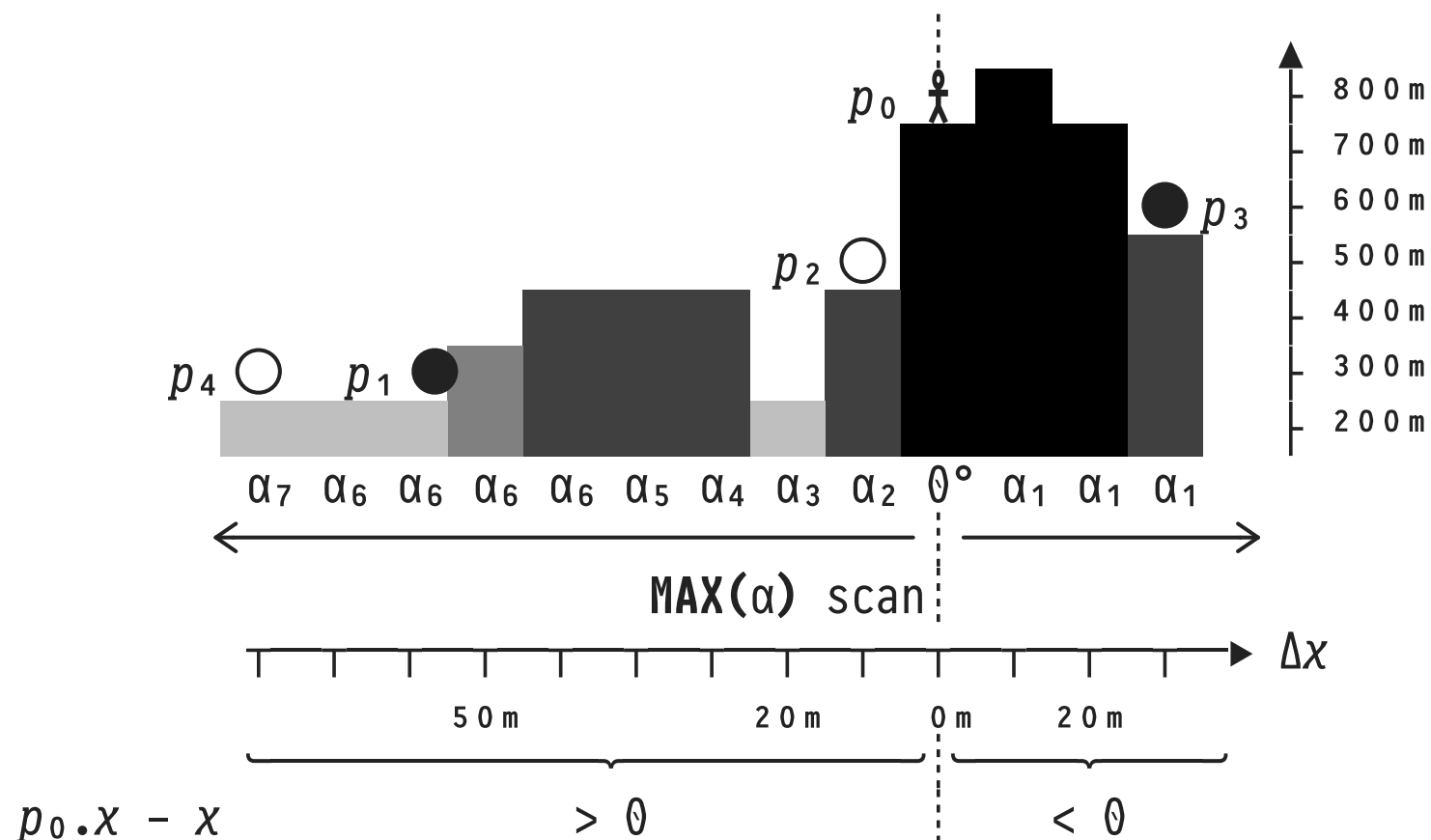
max\_scan( $x$ , max\_angle) **AS** (  
     **SELECT**  $a.x$ ,  
         **MAX**( $a.angle$ )  
         **OVER** (**ORDER BY**  $\text{abs}(p_0.x - a.x)$ ) **AS** max\_angle

**FROM** angles **AS**  $a$

),

⋮

# Looking Left *and* Right: PARTITION BY



- Need **MAX** scans left *and* right of  $p_0 \Rightarrow$  use **PARTITION BY**.



## Looking Left *and* Right: **PARTITION BY**

---

```

WITH
:
-- 2 MAX( $\alpha$ ) scan (left/right of  $p_0$ )
max_scan(x, max_angle) AS (
  SELECT a.x,          --  $\in \{-1, 0, 1\}$ 
         MAX(a.angle)   --  $\underbrace{\hspace{1.5cm}}$ 
         OVER (PARTITION BY sign( $p_0.x - a.x$ )
              ORDER BY abs( $p_0.x - a.x$ )) AS max_angle
  FROM   angles AS a    --  $\underbrace{\hspace{1.5cm}}$ 
                      --  $\Delta x > 0$ 
),
:

```

- $\forall a \in \text{angles}: a.x \neq p_0.x \Rightarrow$  We end up with **two** partitions.

### 3 | Scans: Not Only in the Hills

---

**Scans** are a general and expressive computational pattern:

|  |
|--|
| $\underbrace{\text{agg}(e)}_{(\phi, z, \oplus)} \text{ OVER } (\text{ORDER BY } e_1, \dots, e_n \text{ } \{\text{ROWS}, \text{RANGE}\} \text{ BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW})$ |
|--|

- Available in a variety of forms in programming languages
  - Haskell: `scanl z ⊕ xs`, APL: `⊕\xs`, Python: `accumulate`.  
`scanl ⊕ z [x1, x2, ...] = [z, z ⊕ x1, (z ⊕ x1) ⊕ x2, ...]`
- In parallel programming: *prefix sums* (👉 Guy Blelloch)
  - Sorting, lexical analysis, tree operations, reg.exp. search, drawing operations, image processing, ...

## 4 | Interlude: Quiz

---

**Q:** Assume  $xs \equiv '((b*2)-4*a*c)*0.5'$ . What is computed below?

```
SELECT inp.pos, inp.c,
       SUM([0,1,-1][p.oc]) OVER (ORDER BY inp.pos) AS d
FROM   unnest(string_split(xs(), ''))
       WITH ORDINALITY AS inp(c,pos),
       LATERAL (VALUES (list_position(['(', ')'],
                                     inp.c) + 1)) AS p(oc)
ORDER BY inp.pos;
```

💡 **Hint** (this is the same query expressed in APL):

```
xs ← '((b*2)-4*a*c)*0.5'
+ \ (1 -1 0) ['(') ⌊ xs]
```

## 5 | Beyond Aggregation: Window Functions

---

window function

↓  
 $f$  OVER ([ PARTITION BY  $p_1, \dots, p_m$  ]  
 [ ORDER BY  $e_1, \dots, e_n$  ]  
 [ *frame* ])

Three kinds of **window functions**  $f$ :

1. **Aggregates:**  $SUM(\cdot)$ ,  $AVG(\cdot)$ ,  $MAX(\cdot)$ ,  $list(\cdot)$ , ... process all rows in the frame. ✓
2. **Row Access:** access row by *absolute/relative position* in ordered frame or partition: first/last/ $n^{th}$ / $n$  rows away.
3. **Row Ranking:** assign numeric *rank of row* in partition.

## 6 : LAG/LEAD: Access Rows of the Past and Future

---

Row access at offset  $\pm n$ , relative to the current row:

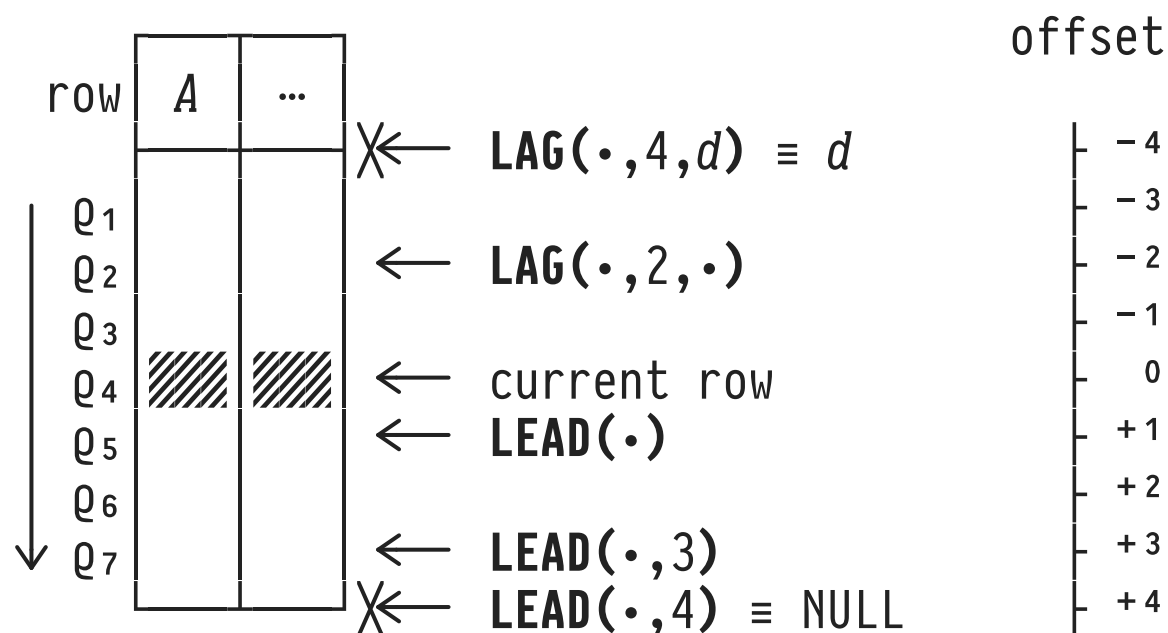
```
-- evaluate e as if we were
-- n rows before the current row
--       $\overbrace{\text{LAG}(e, n, d)}$  OVER ([ PARTITION BY  $p_1, \dots, p_m$  ]
--                                     ORDER BY  $e_1, \dots, e_n$ )
```

### Note:

- **LEAD( $e, n, d$ )**: ...  $n$  rows **after** the current row ...
- Scope is partition—no row access outside the partition.
- If there is no row at offset  $\pm n \Rightarrow$  return value  $d$ .

## LAG/LEAD: Row Offsets (Assume: No Partitioning, ORDER BY A)

---



- The frame of the current row is irrelevant for **LAG/LEAD**.
- Default parameters if absent:  $n \equiv 1$ ,  $d \equiv \text{NULL}$ .

## 🔧 A March Through the Hills: Ascent or Descent?

```

SELECT m.x, m.alt,
       CASE sign(LEAD(m.alt, 1) OVER rightwards - m.alt)
         WHEN -1 THEN '↘' WHEN 1 THEN '↗'
         WHEN 0 THEN '→' ELSE '?'
       END AS climb,
       LEAD(m.alt, 1) OVER rightwards - m.alt AS "by [m]"
FROM   map AS m
WINDOW rightwards AS (ORDER BY m.x) -- marching right

```

| x   | alt | climb | by [m] |
|-----|-----|-------|--------|
| 0   | 200 | →     | 0      |
| ⋮   | ⋮   | ⋮     | ⋮      |
| 90  | 700 | ↗     | 100    |
| 100 | 800 | ↘     | -100   |
| 110 | 700 | ↘     | -200   |
| 120 | 500 | ?     | NULL   |

## 🔧 Crime Scene: Sessionization

---

A spy broke into the Police HQ computer system. A **log** records keyboard activity of user **uid** at time **ts**:

| <u>uid</u> | <u>ts</u>           |
|------------|---------------------|
| 👤          | 2024-05-25 07:25:12 |
| 👤          | 2024-05-25 07:25:18 |
| 👤          | 2024-05-25 08:01:55 |
| 👤          | 2024-05-25 08:05:07 |
| 👤          | 2024-05-25 08:05:30 |
| 👤          | 2024-05-25 08:05:39 |
| 👤          | 2024-05-25 08:05:46 |

Table **log**

- **Q:** Can we **sessionize** the log so that investigators can identify *sessions* ( $\equiv$  streaks of uninterrupted activity)?

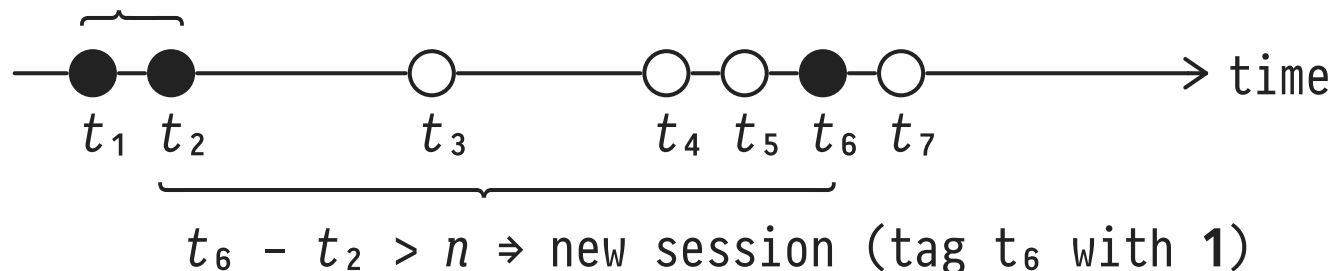


## 🔧 Sessionization (Query Plan)

---

1. Cop and spy sessions happen independently (even if interleaved): partition **log** into 🕵️/● and 🚔/○ rows.
2. **Tag** keyboard activities (below: tagging for ●):

$t_2 - t_1 \leq \text{threshold } n \Rightarrow \text{continue session (tag } t_2 \text{ with 0)}$



3. **Scan** the tagged table and derive session IDs by maintaining a **running sum** of *start of session* tags.

## 🔧 Sessionization (Query Plan)

1

| uid | ts    |
|-----|-------|
| ●   | $t_1$ |
| ●   | $t_2$ |
| ○   | $t_3$ |
| ○   | $t_4$ |
| ○   | $t_5$ |
| ●   | $t_6$ |
| ○   | $t_7$ |

2

| uid   | ts    |
|-------|-------|
| ●     | $t_1$ |
| ●     | $t_2$ |
| ●     | $t_6$ |
| ----- |       |
| ○     | $t_3$ |
| ○     | $t_4$ |
| ○     | $t_5$ |
| ○     | $t_7$ |

3

| uid   | ts    | sos |
|-------|-------|-----|
| ●     | $t_1$ | 1   |
| ●     | $t_2$ | 0   |
| ●     | $t_6$ | 1   |
| ----- |       |     |
| ○     | $t_3$ | 1   |
| ○     | $t_4$ | 1   |
| ○     | $t_5$ | 0   |
| ○     | $t_7$ | 0   |

← log start

└─  $t_6 - t_2 > n$

└─  $\Rightarrow$  new session

← log start

└─  $t_7 - t_5 \leq n$

└─  $\Rightarrow$  continue

4

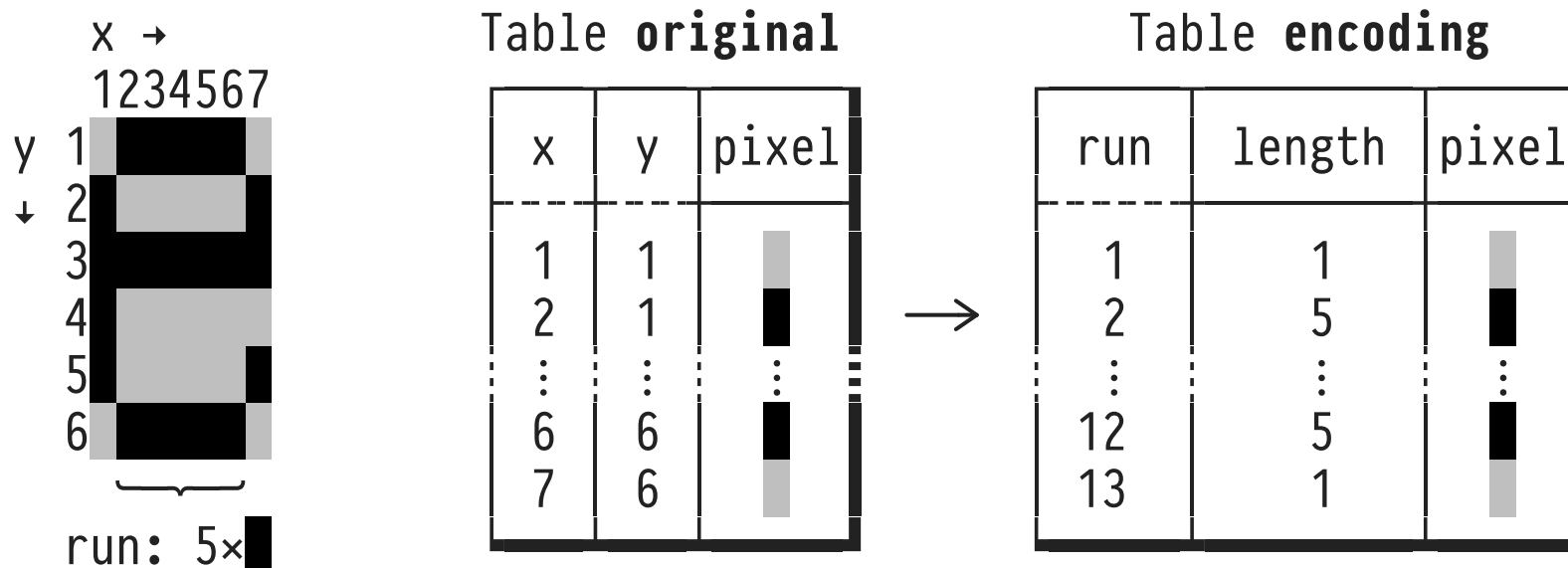
| uid   | ts    | session |
|-------|-------|---------|
| ●     | $t_1$ | 1       |
| ●     | $t_2$ | 1       |
| ●     | $t_6$ | 2       |
| ----- |       |         |
| ○     | $t_3$ | 1       |
| ○     | $t_4$ | 2       |
| ○     | $t_5$ | 2       |
| ○     | $t_7$ | 2       |

- At log start, always begin a new session (**sos = 1**).
- How to assign *global session IDs* (○'s sessions: 3, 4)?

## 🔧 Image Compression by Run-Length Encoding

---

Compress image by identifying pixel **runs** of the same color:



- Here: assumes a row-wise linearization of the pixel map.
- In b/w images we may omit column **pixel** in table **encoding**.

## 🔧 Run-Length Encoding (Query Plan)

1

| x | y | pixel | change? |
|---|---|-------|---------|
| 1 | 1 | ⬜     | t 1     |
| 2 | 1 | ⬛     | t 1     |
| 3 | 1 | ⬛     | f 0     |
| 4 | 1 | ⬛     | f 0     |
| 5 | 1 | ⬛     | f 0     |
| 6 | 1 | ⬛     | f 0     |
| 7 | 1 | ⬜     | t 1     |
| ⋮ | ⋮ | ⋮     | ⋮       |

2

| x | y | pixel | change? | Σ change? |
|---|---|-------|---------|-----------|
| 1 | 1 | ⬜     | 1       | 1         |
| 2 | 1 | ⬛     | 1       | 2         |
| 3 | 1 | ⬛     | 0       | 2         |
| 4 | 1 | ⬛     | 0       | 2         |
| 5 | 1 | ⬛     | 0       | 2         |
| 6 | 1 | ⬛     | 0       | 2         |
| 7 | 1 | ⬜     | 1       | 3         |
| ⋮ | ⋮ | ⋮     | ⋮       | ⋮         |

run #2 of  
length 5

- ①: `LAG(pixel,1,undefined)`: pixel @ (1,1) always “changes.”
- ②: A `SUM()` scan of `change?` yields run identifiers.

## 7 : **FIRST\_VALUE, LAST\_VALUE, NTH\_VALUE**: In-Frame Row Access

---

Aggregates reduce *all rows* inside a frame to a single value.  
Now for something different:

- **Positional access to individual rows** inside a frame is provided by three window functions:

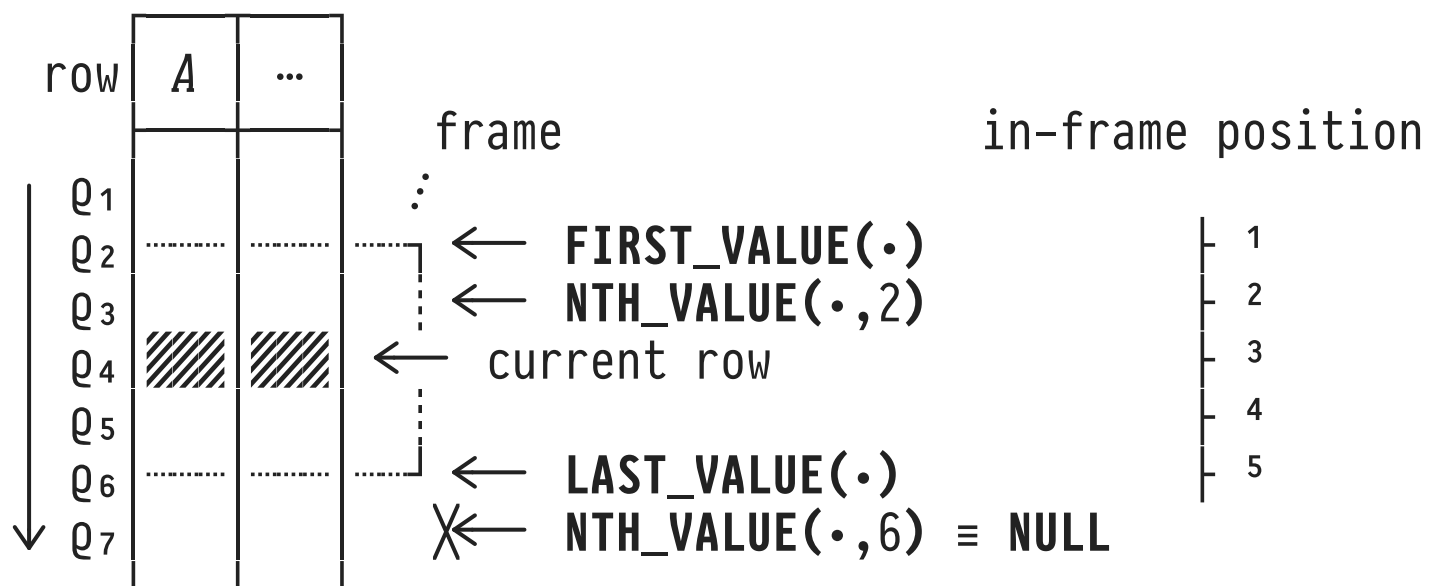
```
-- evaluate expression e as if we were at  
-- the first/last/ $n^{\text{th}}$  row in the frame  
--
```

```
    FIRST_VALUE(e)  
    LAST_VALUE(e)  
    NTH_VALUE(e,n)  } OVER (...)
```

- **NTH\_VALUE(*e*,*n*)**: No  $n^{\text{th}}$  row in frame  $\Rightarrow$  return **NULL**.

## In-Frame Row Access

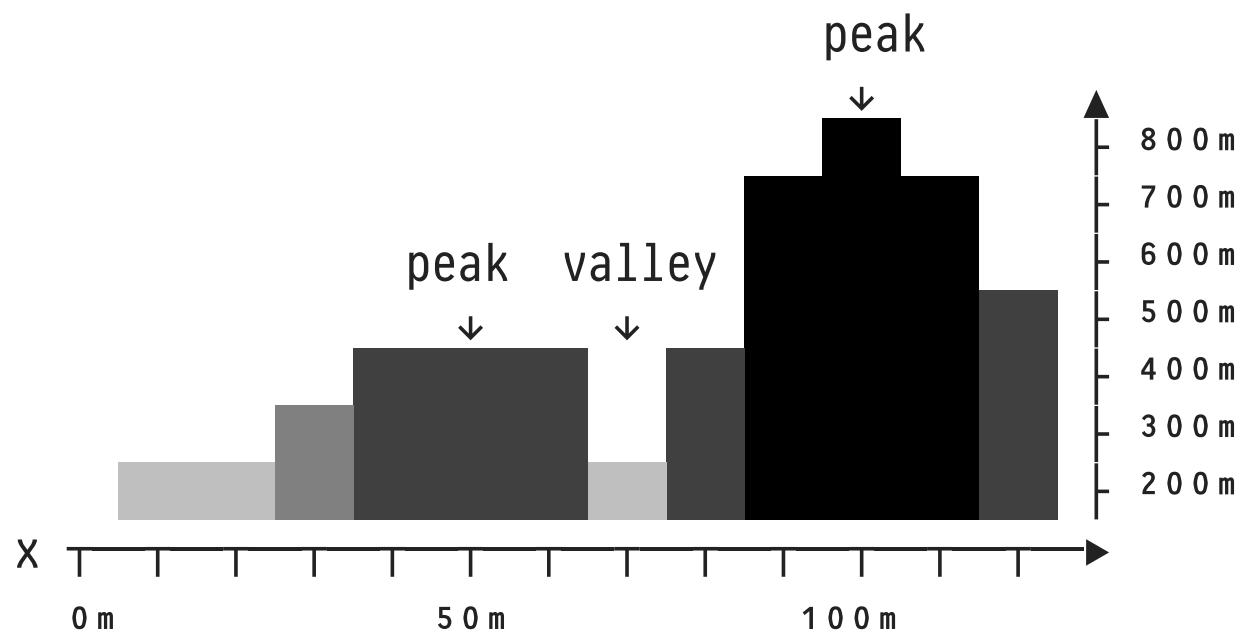
---



- We have  $\text{FIRST\_VALUE}(e) \equiv \text{NTH\_VALUE}(e,1)$ .

## 🔧 Detecting Landscape Features

---

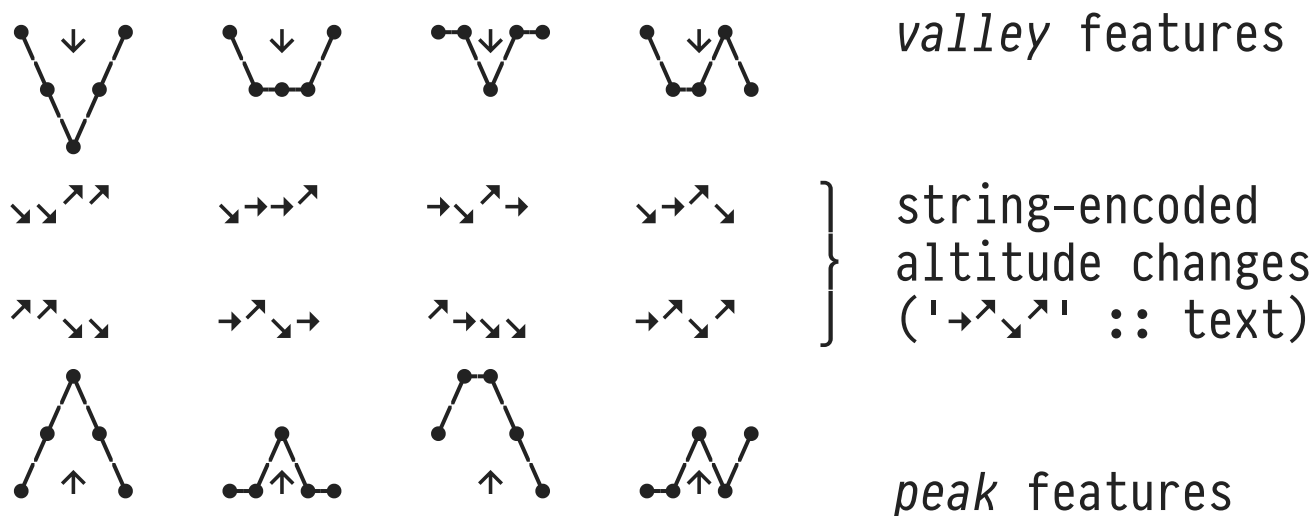


- Detect features in a hilly landscape. Attach label  $\in \{\text{peak}, \text{valley}, -\}$  to every location  $x$ .
- Feature defined by relative altitude change **in vicinity**.

## 🔧 Detecting Landscape Features (Query Plan)

---

1. Track relative altitude changes in a sliding **x-window** of size 5:

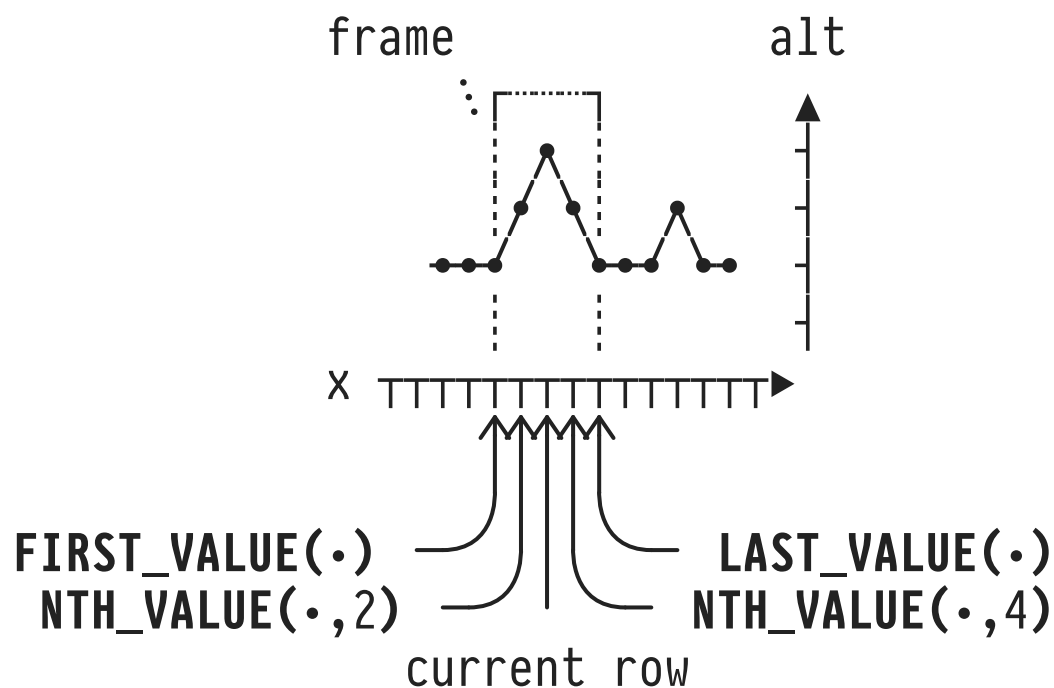


2. **Pattern match** on change strings to detect features.



## Altitude Changes in a Sliding Window

- Frame: ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING (5 rows):



- $\text{FIRST\_VALUE}(\text{alt}) < \text{NTH\_VALUE}(\text{alt}, 2) \Rightarrow \text{ascent ('↗')}.$

## 🔧 Altitude Changes in a Sliding Window

---

```
-- Find slopes in ±2 vicinity around point x
```

```
SELECT m.x,  
       slope(sign(FIRST_VALUE(m.alt) OVER w-NTH_VALUE(m.alt,2) OVER w)) ||  
       slope(sign(NTH_VALUE(m.alt,2) OVER w-m.alt                      )) ||  
       slope(sign(m.alt -NTH_VALUE(m.alt,4) OVER w)) ||  
       slope(sign(NTH_VALUE(m.alt,4) OVER w-LAST_VALUE(m.alt) OVER w))  
FROM   map AS m  
WINDOW w AS (ORDER BY m.x ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)
```


- Recall: 1D landscape represented as table `map(x,alt)`.
- Encode altitude changes: `slope(-1) ≡ '↗'`, `slope(0) ≡ '→'`  
(here: `slope(·)` implemented as a DuckDB macro).

## Row Pattern Matching (SQL:2016)

---

SQL:2016 introduced an entirely new SQL construct, **row pattern matching** (`MATCH_RECOGNIZE`):

1. `ORDER BY`: Order the rows of a table.
2. `DEFINE`: Tag rows that satisfy given predicates.
3. `PATTERN`: Specify a **regular expression over row tags**, find matches in the ordered sequence of rows.
4. `MEASURES`: For each match, evaluate expressions that measure its features (matched rows, length, ...).

⚠ As of June 2022, not supported by . Currently implemented by Oracle® only.

## Row Pattern Matching (SQL:2016)

---

```

SELECT *
FROM   map
MATCH_RECOGNIZE (
  ORDER BY x
  MEASURES FIRST(x,1)      AS x,
             MATCH_NUMBER() AS feature,
             CLASSIFIER()  AS slope
  ONE ROW PER MATCH
  AFTER MATCH SKIP TO NEXT ROW
  PATTERN ((DOWN DOWN|DOWN EVEN|UP DOWN|EVEN DOWN)...)
  DEFINE UP      AS UP.alt > PREV(UP.alt),  --
         DOWN    AS DOWN.alt < PREV(DOWN.alt), --
         EVEN    AS EVEN.alt = PREV(EVEN.alt) -- } row tags
)

```

### Output

| x   | feature | slope |
|-----|---------|-------|
| 50  | 1       | DOWN  |
| 70  | 2       | UP    |
| 100 | 3       | DOWN  |

[todo] below

## 8 : Numbering and Ranking Rows

---

Countless problem scenarios involve the **number** (position) or **rank** of the current row in an *ordered sequence* of rows.

- Family of **window functions to number/rank rows**:

|                |   |   |                              |
|----------------|---|---|------------------------------|
| ROW_NUMBER()   | } | OVER ( [ PARTITION BY $p_1, \dots, p_m$ ] | -- intra-partition ranking ✓ |
| DENSE_RANK()   |   |   | --                           |
| RANK()         |   |   | --                           |
| PERCENT_RANK() |   |   | --                           |
| CUME_DIST()    |   |   | --                           |
| NTILE( $n$ )   |   | [ ORDER BY $e_1, \dots, e_n$ ] )          | -- ranking w/o ORDER BY ⚡    |

- Scope is partition (if present)—*frame* is irrelevant.

# Numbering and Ranking Rows — $f()$ OVER (ORDER BY $A$ )

---

Table W  $f$

| row | <b>A</b> | ROW_NUMBER | DENSE_RANK | RANK |
|-----|----------|------------|------------|------|
| q1  | 1        | 1          | 1          | 1    |
| q2  | 2        | 2          | 2          | 2    |
| q3  | 3        | 3          | 3          | 3    |
| q4  | 3        | 4          | 3          | 3    |
| q5  | 3        | 5          | 3          | 3    |
| q6  | 4        | 6          | 4          | 6    |
| q7  | 6        | 7          | 5          | 7    |
| q8  | 6        | 8          | 5          | 7    |
| q9  | 7        | 9          | 6          | 9    |

- ... Rows that agree on
- ... the sort criterion
- ... (here: **A**) ...
- ... number randomly
- ... rank equally

⊥ Mind the ranking gap  
(think Olympics)

- In general:  $\text{DENSE\_RANK}() \leq \text{RANK}() \leq \text{ROW\_NUMBER}()$

## 🔧 Once More: Find the Top $n$ Rows in a Group

---

| species | length | height | legs                        |
|---------|--------|--------|-----------------------------|
| :       | :      | :      | $\in \{2, 4, \text{NULL}\}$ |

Table `dinosaurs`

```

SELECT tallest.legs, tallest.species, tallest.height
FROM (SELECT d.legs, d.species, d.height,
ROW_NUMBER()...RANK() OVER (PARTITION BY d.legs
                                ORDER BY d.height DESC) AS n
      FROM dinosaurs AS d
      WHERE d.legs IS NOT NULL) AS tallest
WHERE tallest.n <= 3

```

- `RANK()` vs `ROW_NUMBER()`: both OK, but different semantics!
- Need a subquery: window functions *not* allowed in `WHERE`.

## 🔧 Identify Consecutive Ranges

---

- What you often encounter in scientific papers 🙄:  
 “... as Knuth has shown in  $[5, 2, 14, 3, 1, 42, 6, 10, 7, 13]$  ...”
- What you want to see 😊:  
 “... as Knuth has shown in  $[1-3, 5-7, 10, 13\&14, 42]$  ...”

Table **citations**

| ref |
|-----|
| 5   |
| 2   |
| ⋮   |
| 13  |



| ref | range |
|-----|-------|
| 1   | $r_0$ |
| 2   | $r_0$ |
| ⋮   | ⋮     |
| 42  | $r_4$ |

**Output**

← references belong  
 ← to the same range



# Identify Consecutive Ranges (Query Plan)

---

| 1   | 2   | ROW_NUMBER() |    |   |    |                        |
|-----|-----|--------------|----|---|----|------------------------|
| ref | ref |              |    |   |    |                        |
| 5   | 1   | -            | 1  | = | 0  | } range 0 $\equiv r_0$ |
| 2   | 2   | -            | 2  | = | 0  |                        |
| 14  | 3   | -            | 3  | = | 0  |                        |
| 3   | 5   | -            | 4  | = | 1  | } range 1 $\equiv r_1$ |
| 1   | 6   | -            | 5  | = | 1  |                        |
| 42  | 7   | -            | 6  | = | 1  |                        |
| 6   | 10  | -            | 7  | = | 3  | } range 3 $\equiv r_2$ |
| 10  | 13  | -            | 8  | = | 5  |                        |
| 7   | 14  | -            | 9  | = | 5  | } range 5 $\equiv r_3$ |
| 13  | 42  | -            | 10 | = | 32 |                        |
|     |     | ...          |    |   |    | range 32 $\equiv r_4$  |
|     |     | subtract     |    |   |    |                        |

# Numbering and Ranking Rows — $f$ OVER (ORDER BY $A$ )

---

| row | <b>A</b> | PERCENT_RANK | CUME_DIST | NTILE(3) |   |
|-----|----------|--------------|-----------|----------|---|
| q1  | 1        | 0            | 1/9       | 1        | ...                                     |
| q2  | 2        | 1/8          | 2/9       | 1        | ...                                     |
| q3  | <b>3</b> | 2/8          | 5/9       | 1        | ...                                     |
| q4  | <b>3</b> | 2/8          | 5/9       | 2        | Rows that agree on                      |
| q5  | <b>3</b> | 2/8          | 5/9       | 2        | the sort criterion                      |
| q6  | 4        | 5/8          | 6/9       | 2        | (here: <b>A</b> ) rank equally          |
| q7  | <b>6</b> | 6/8          | 8/9       | 3        |   |
| q8  | <b>6</b> | 6/8          | 8/9       | 3        | ← current row is in the $n^{\text{th}}$ |
| q9  | 7        | 8/8          | 9/9       | 3        | of 3 chunks of rows                     |

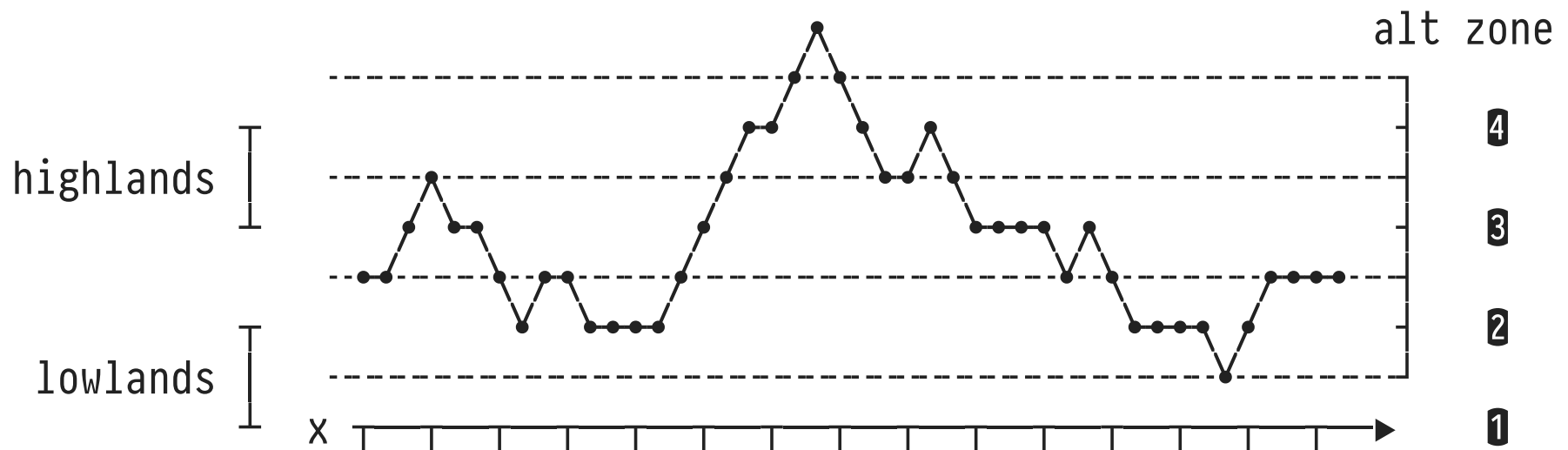
$n\%$  of the other rows rank lower than the current row

the current row and lower ranked rows make up  $n\%$  of all rows

## 🔧 Altitudinal Mountain Zones

---

- Classify the altitudes of a mountain range into
  - equal-sized** vegetation **zones** and
  - lowlands (altitude in the lowest **20%**) and highlands (between **60%-80%** of maximum altitude).

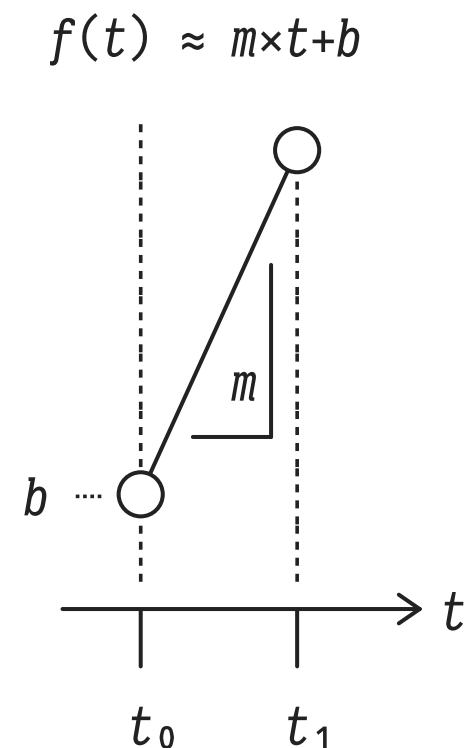
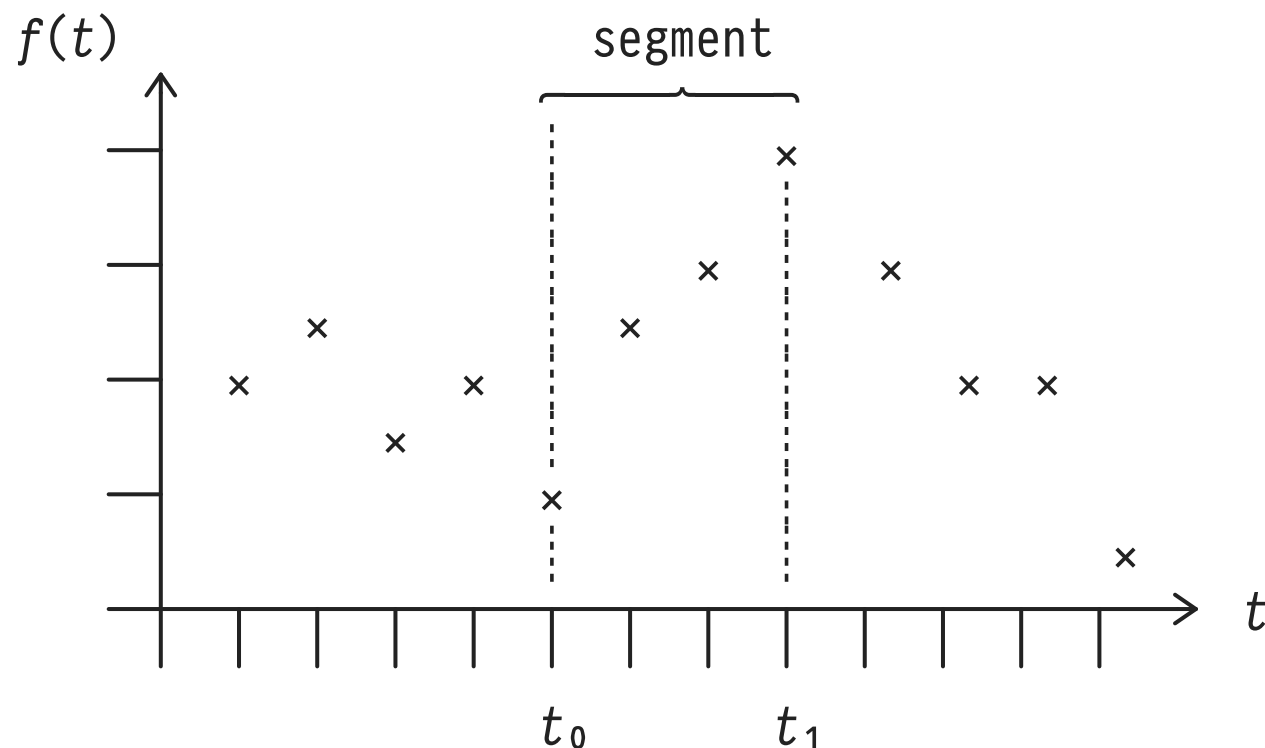


## Altitudinal Mountain Zones

---

```
-- Classify altitudinal zones in table mountains(x, alt)
--
SELECT
  m.x, m.alt,
  NTILE(4) OVER altitude AS zone,
  CASE
    WHEN PERCENT_RANK() OVER altitude BETWEEN 0.6 AND 0.8
      THEN 'highlands'
    WHEN PERCENT_RANK() OVER altitude < 0.2
      THEN 'lowlands'
    ELSE '-'
  END AS region
FROM mountains AS m
WINDOW altitude AS (ORDER BY m.alt)
ORDER BY m.x;
```

## ⚙ Linear Approximation of a Time Series



1. **NTILE( $n$ )** segments time series at desired granularity.
2. Compute  $m$ ,  $b$  in each **segment**  $\equiv$  **window frame**.

## 9 | Summary: Window Function Semantics<sup>2</sup>

---

| Scope     | Computation | Function  | Description  |
|-----------|-------------|---|--|
| frame     | aggregation | (aggregates)  | <code>SUM</code> , <code>AVG</code> , <code>MAX</code> , <code>array_agg</code> , ...  |
|           | row access  | <code>FIRST_VALUE(e)</code><br><code>LAST_VALUE(e)</code><br><code>NTH_VALUE(e,n)</code>  | <i>e</i> at first row in frame<br><i>e</i> at last row in frame<br><i>e</i> at $n^{\text{th}}$ row in frame  |
| partition | row access  | <code>LAG(e,n,d)</code><br><code>LEAD(e,n,d)</code>   | <i>e</i> at <i>n</i> rows <i>before</i> current row<br><i>e</i> at <i>n</i> rows <i>after</i> current row  |
|           | ranking     | <code>ROW_NUMBER()</code><br><code>RANK()</code><br><code>DENSE_RANK()</code><br><code>PERCENT_RANK()</code><br><code>CUME_DIST()</code><br><code>NTILE(n)</code> | number of current row<br>rank with gaps ("Olympics")<br>rank without gaps<br>relative rank of current row<br>ratio of rows up to —"<br>rank on a scale $\{1,2,\dots,n\}$ |

<sup>2</sup> `FIRST_VALUE(e)`: expression *e* will be evaluated as if we are at the first row in the frame.

`LAG(e,n,d)`: default expression *d* is returned if there is no row at offset *n* before the current row.