

Advanced SQL

⑤

Window Functions

Winter 2025/26

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:

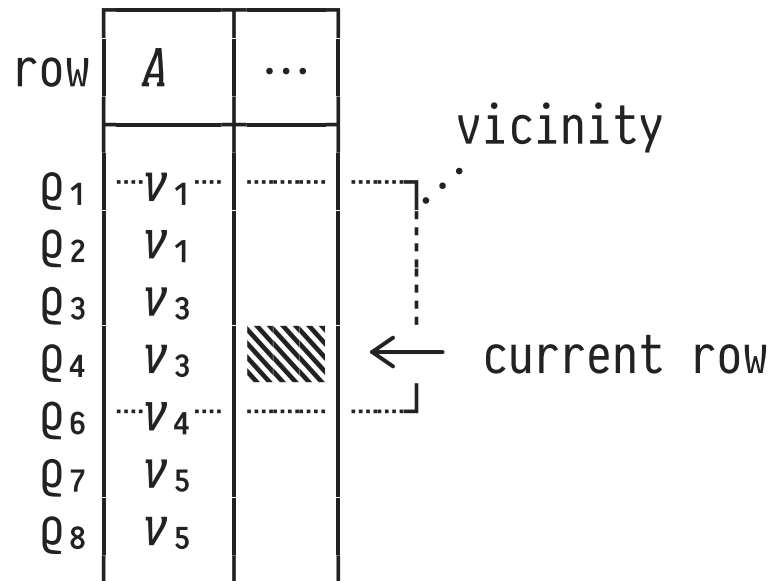
SQL Feature	Mode of Computation
function/operator	row → row
table-generating function	row → table of rows
aggregate	group of rows → row (one per group)
window function 🍷	row vicinity → row (one per row)

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.

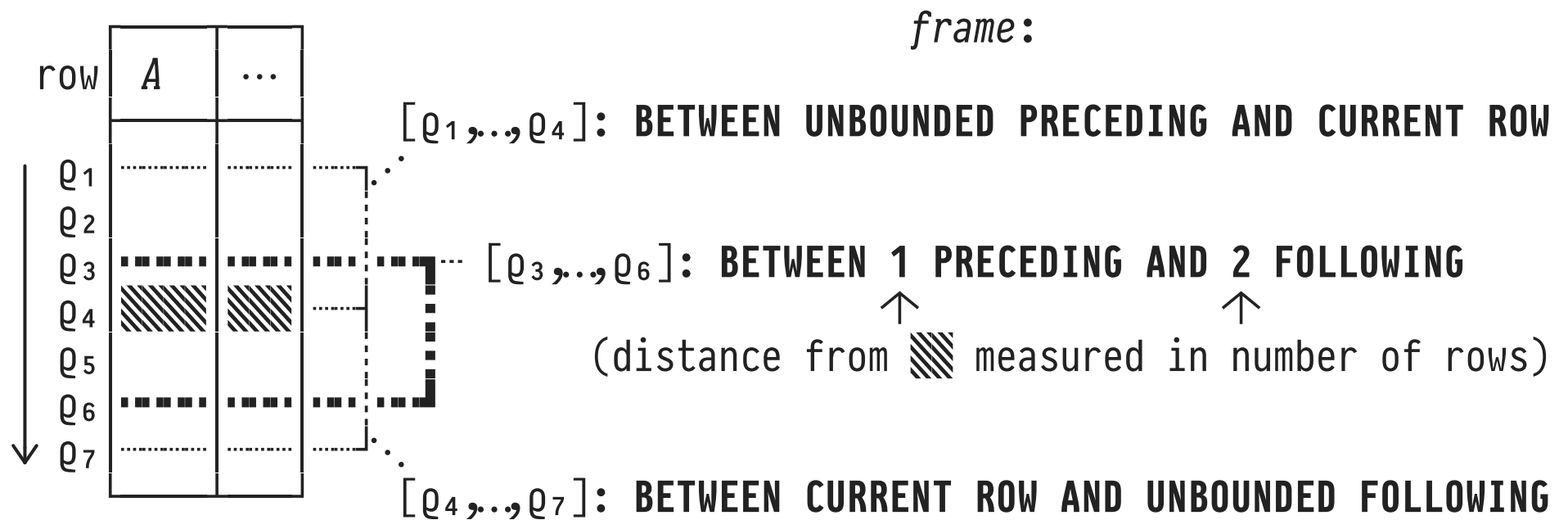
Row Vicinity: Window Frames



- Each row is the **current row** at one point in time.
- Row vicinity (**window, frame**) is based on either:
 - ① row **position** (**ROWS** windows),
 - ② row **values** v_i (**RANGE** windows),
 - ③ row **peers** (**GROUPS** 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** $frame$])

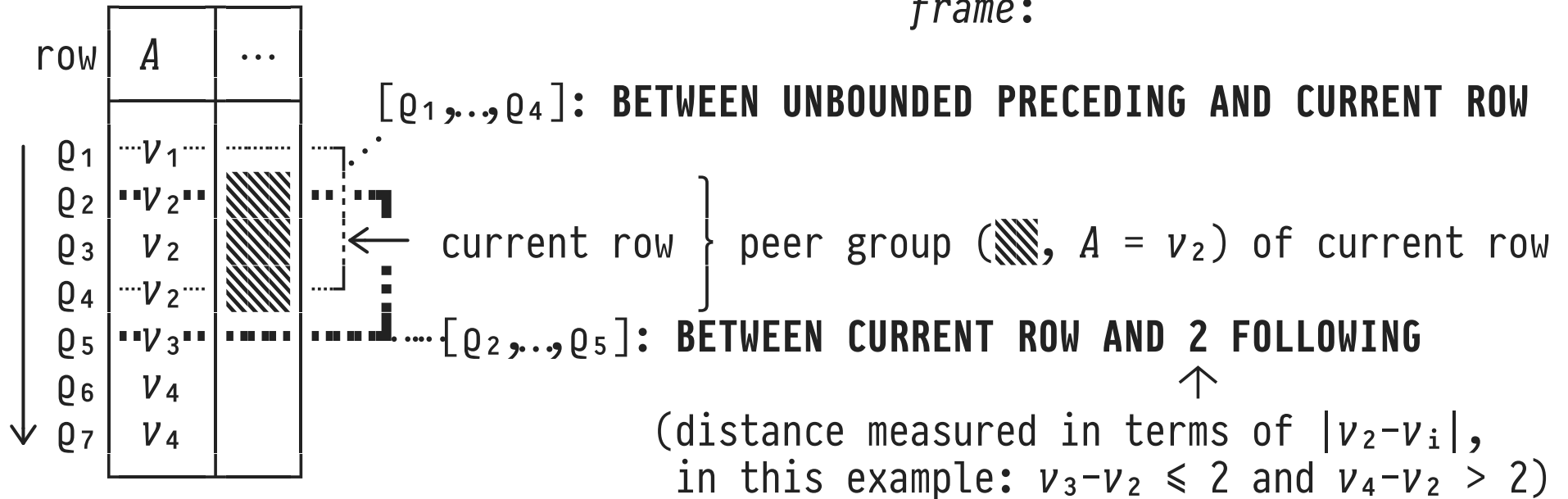


Window Frame Specifications (Variant: **RANGE**)

window function one criterion frame specification

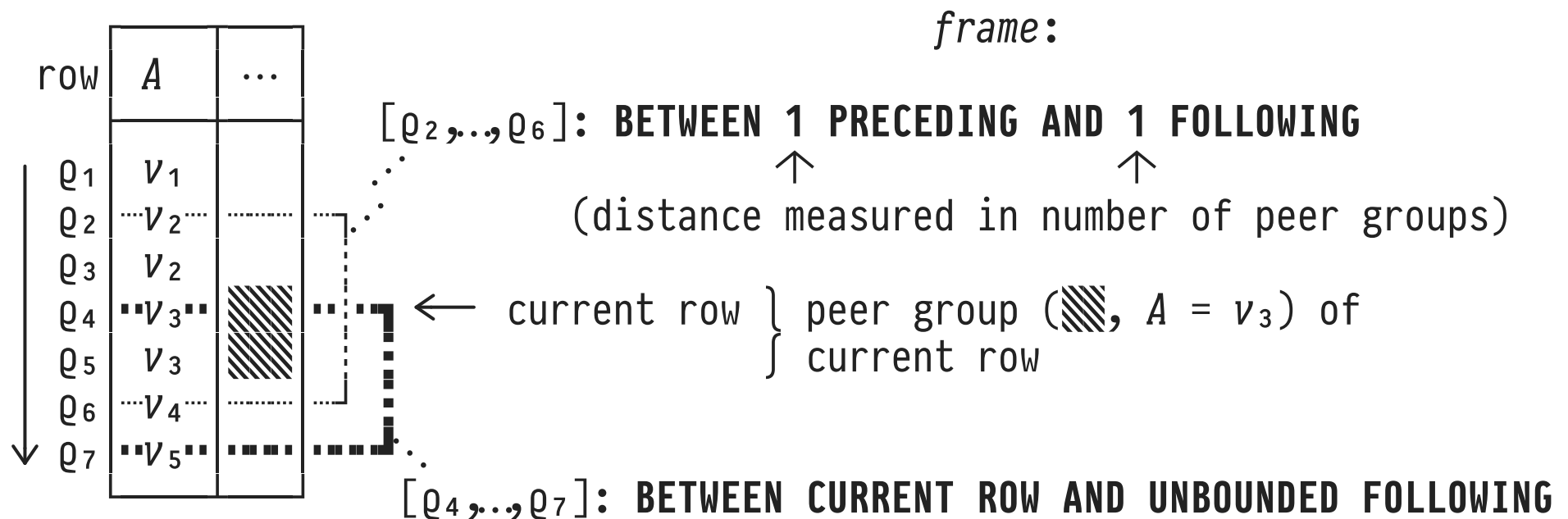
f **OVER** (ORDER BY A [**RANGE** $frame$])

frame:

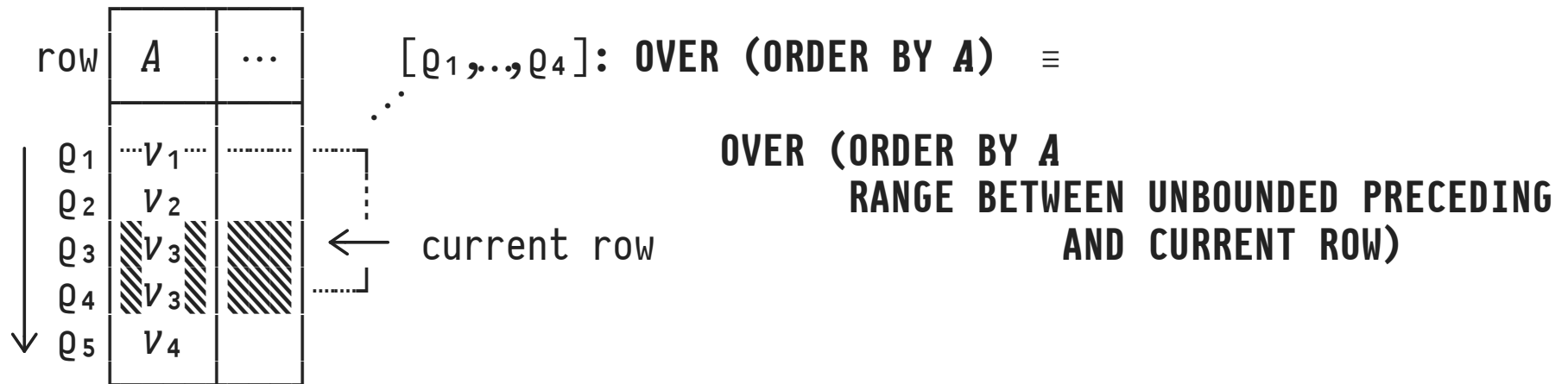
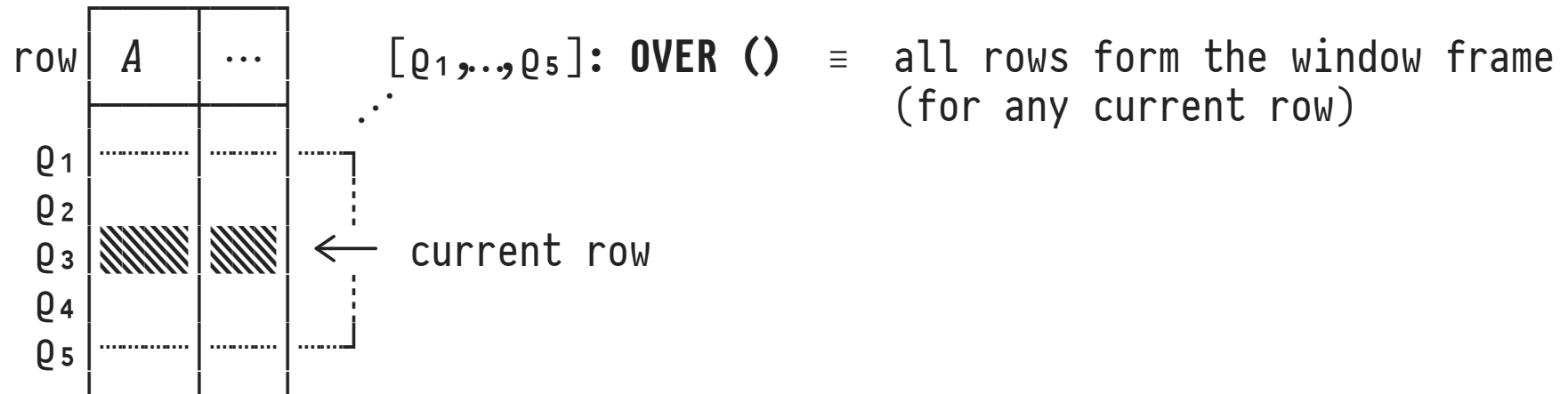


Window Frame Specifications (Variant: **GROUPS**)

window function ordering criteria frame specification
 f **OVER** (**ORDER BY** e_1, \dots, e_n [**GROUPS** $frame$])



Window Frame Specifications: Abbreviations



WINDOW Frame Specifications: Exclusion¹

window function

exclusion clause

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

row	A	...
Q ₁	V ₁	
Q ₂	V ₂	
Q ₃	V ₃	
Q ₄	V ₃	
Q ₅	V ₃	
Q ₆	V ₄	

← current row

exclusion:

EXCLUDE NO OTHERS [Q₂, Q₃, Q₄, Q₅]

EXCLUDE CURRENT ROW [Q₂, Q₃, Q₅]

EXCLUDE GROUP [Q₂]

EXCLUDE TIES [Q₂, Q₄]

¹ Q: Which *frame* would lead to a window as shown above?

WINDOW Clause: Name the Frame

Syntactic \P : 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 ₁	1	●
Q ₂	2	○
Q ₃	3	○
Q ₄	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/m².
- **Expected output:**

weekend?	% fine
f	29
t	43

2 : PARTITION BY: Window Frames Inside Partitions

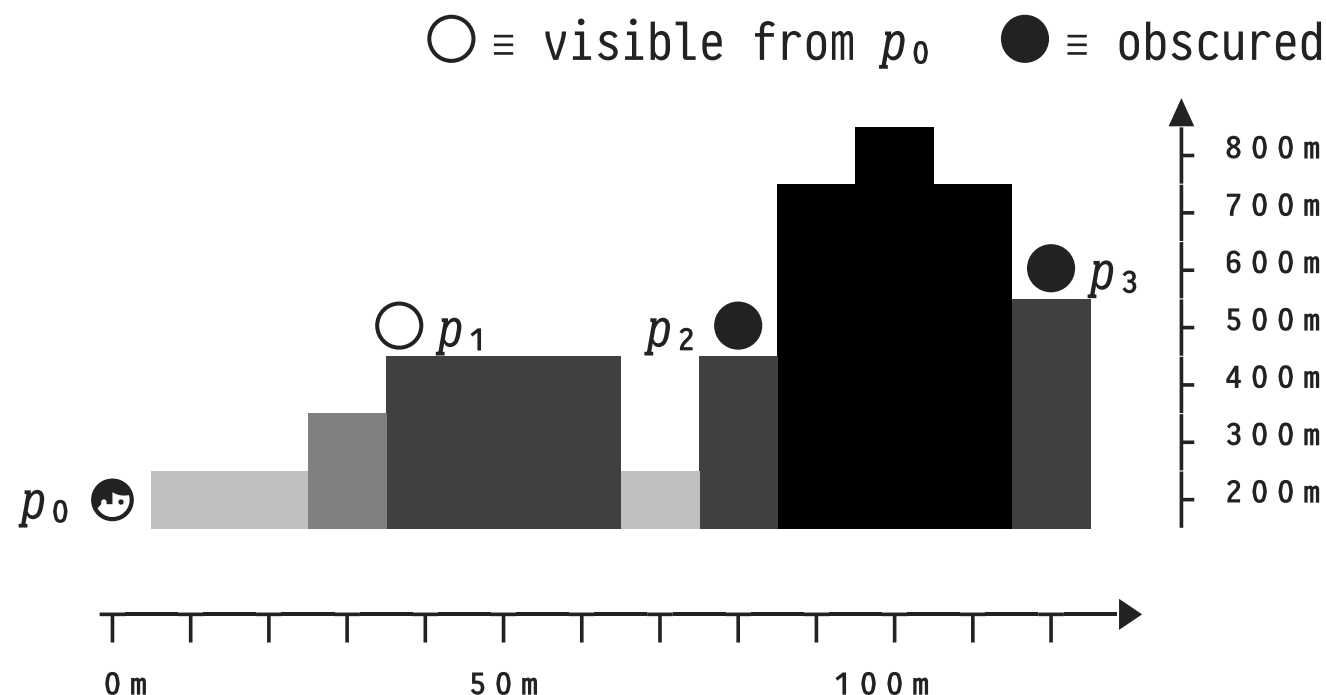
Optionally, we may **partition** the input table *before* rows are sorted and window frames are determined:

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 ] )
```

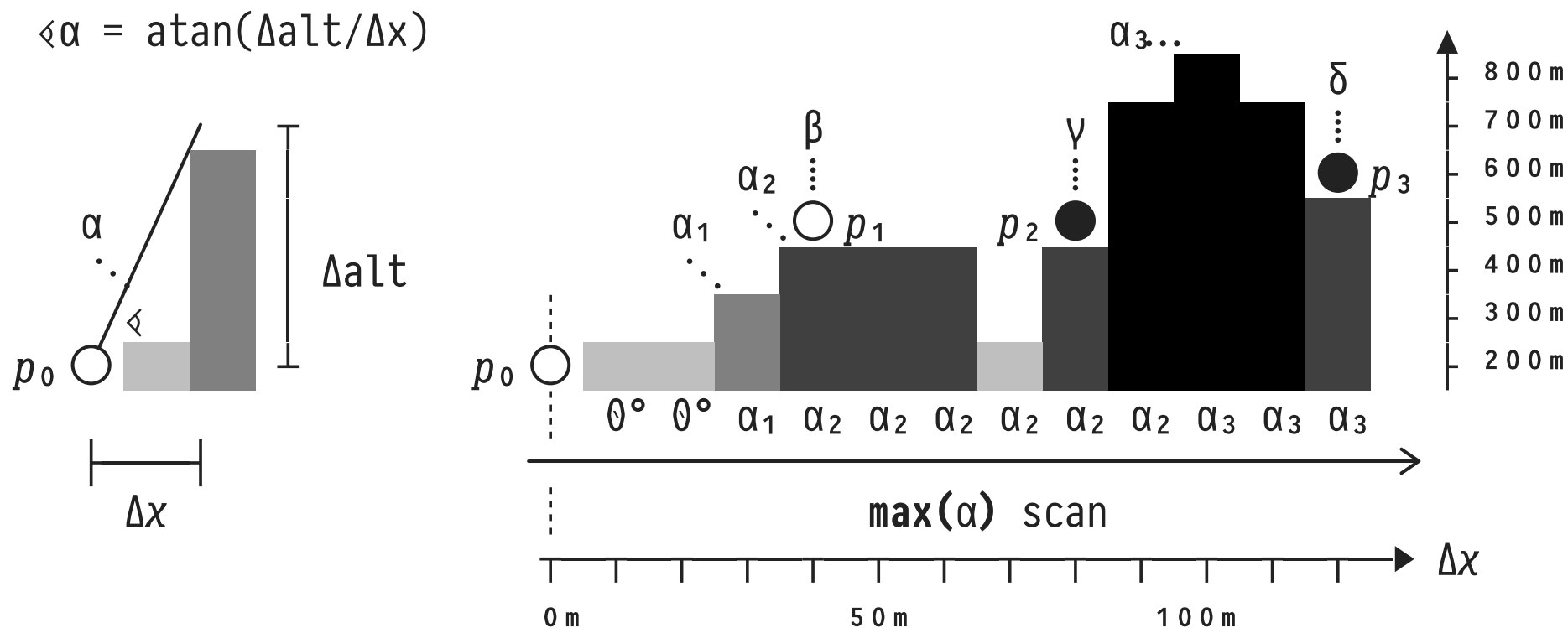
- Note:
 1. Frames **never cross partitions**.
 2. **BETWEEN ... PRECEDING AND ... FOLLOWING** respects **partition boundaries**.

🔧 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 $\beta \geq \alpha_2$, $\gamma < \alpha_2$, $\delta < \alpha_3$.
 \uparrow \uparrow \uparrow
 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
\vdots	\vdots
120	500

Table `map`

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

x	visible?
0	true
10	true
\vdots	\vdots
120	false

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

WITH

-- ❶ Angles α (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 $\max(\alpha)$ scan (to the right of p_0)

max_scan(x , max_angle) **AS** (

SELECT $a.x$,

max($a.angle$)

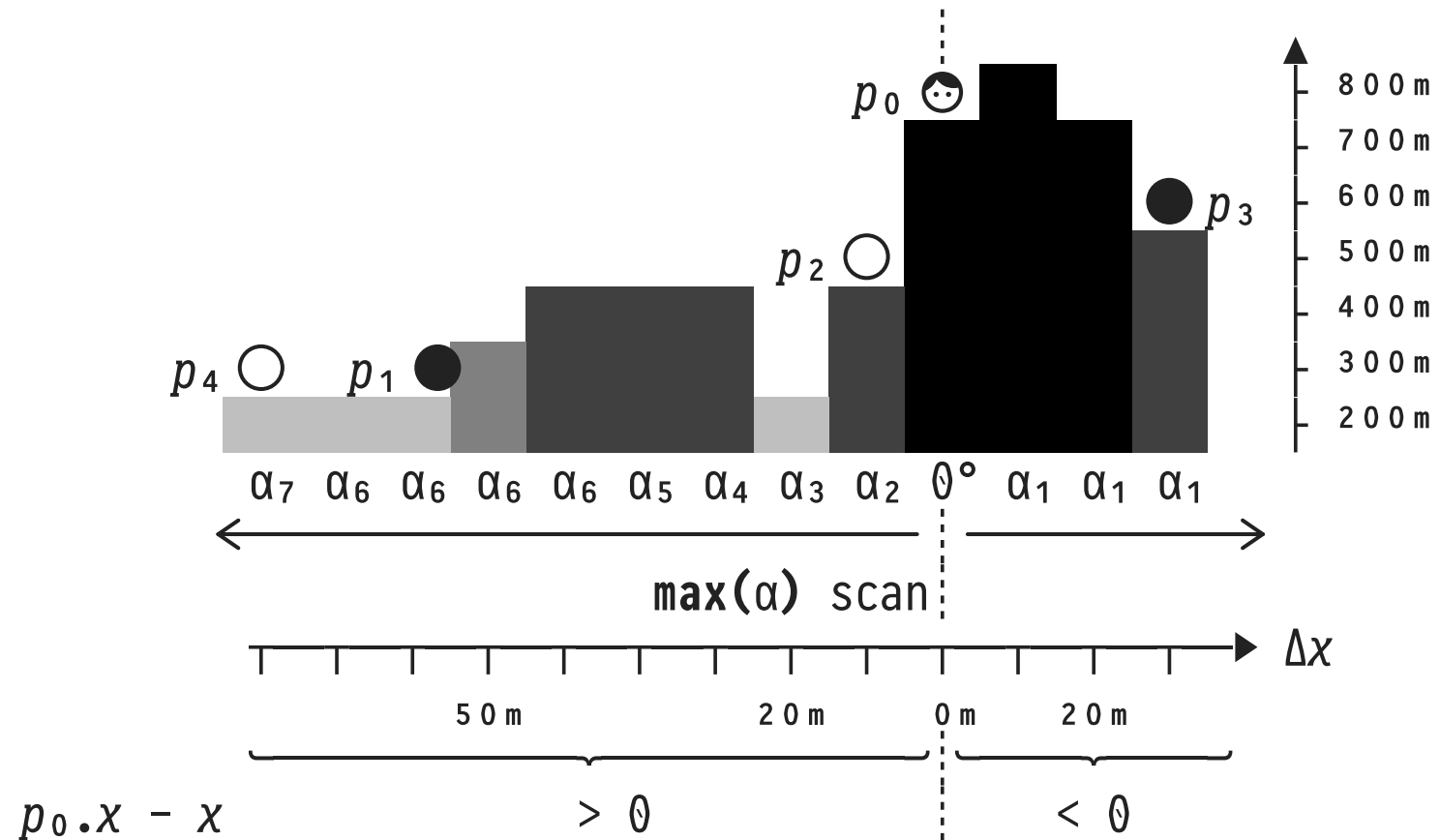
OVER (**ORDER BY** **abs**($p_0.x - a.x$)) **AS** max_angle

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)   --  $\underbrace{\hspace{2cm}}$ 
         OVER (PARTITION BY sign( $p_0.x - a.x$ )
              ORDER BY abs( $p_0.x - a.x$ )) AS max_angle
  FROM   angles AS a    --  $\underbrace{\hspace{2cm}}$ 
                      --  $\Delta x > 0$ 
),
:

```

- $\forall a \in \text{angles}: a.x \neq p_0.x \Rightarrow$ 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([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))) 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:** $\text{sum}(\cdot)$, $\text{avg}(\cdot)$, $\text{max}(\cdot)$, $\text{list}(\cdot)$, ... 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

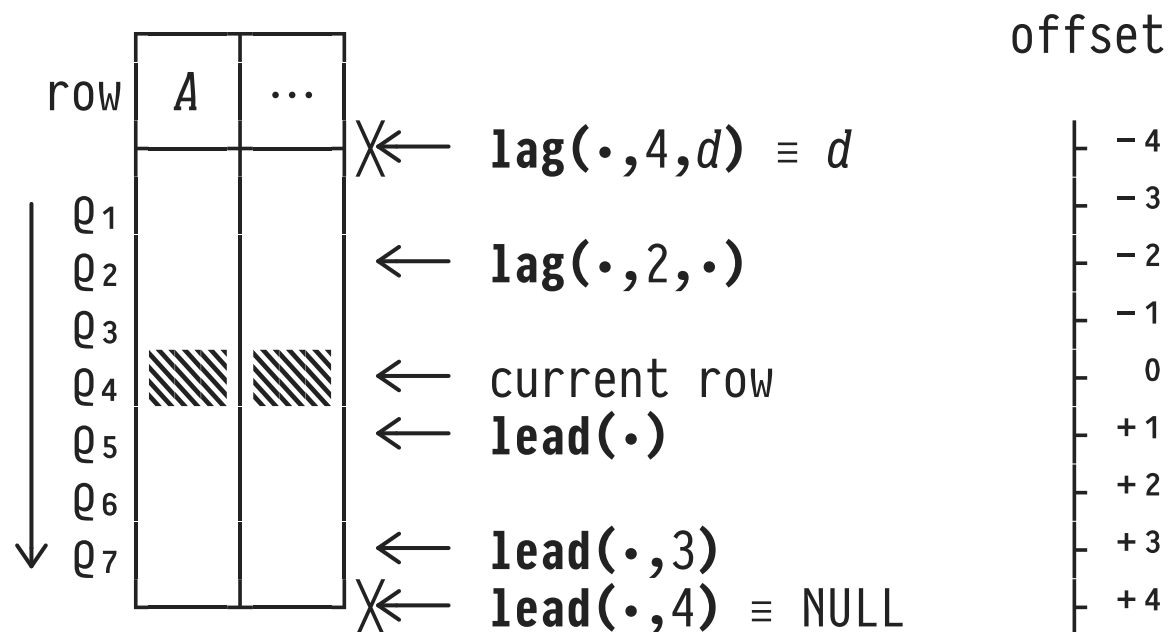
Row access at offset $\pm n$, relative to the current row:

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

Note:

- $\text{lead}(e, n, d)$: ... n rows **after** the current row ...
- Scope is partition—no row access outside the partition.
- No row at offset $\pm n$? \Rightarrow return default 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>
0:0:0	2025-12-11 07:25:12
0:0:0	2025-12-11 07:25:18
0:0:0	2025-12-11 08:01:55
0:0:0	2025-12-11 08:05:07
0:0:0	2025-12-11 08:05:30
0:0:0	2025-12-11 08:05:39
0:0:0	2025-12-11 08:05:46

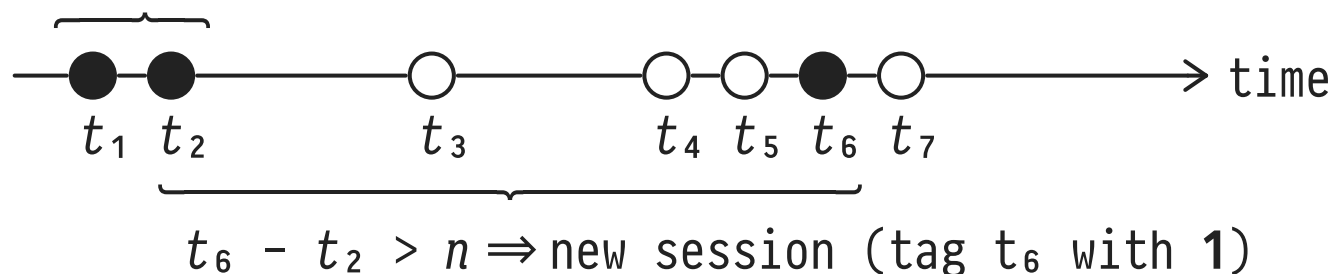
Table **log**

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

🔧 Sessionization (Query Plan)

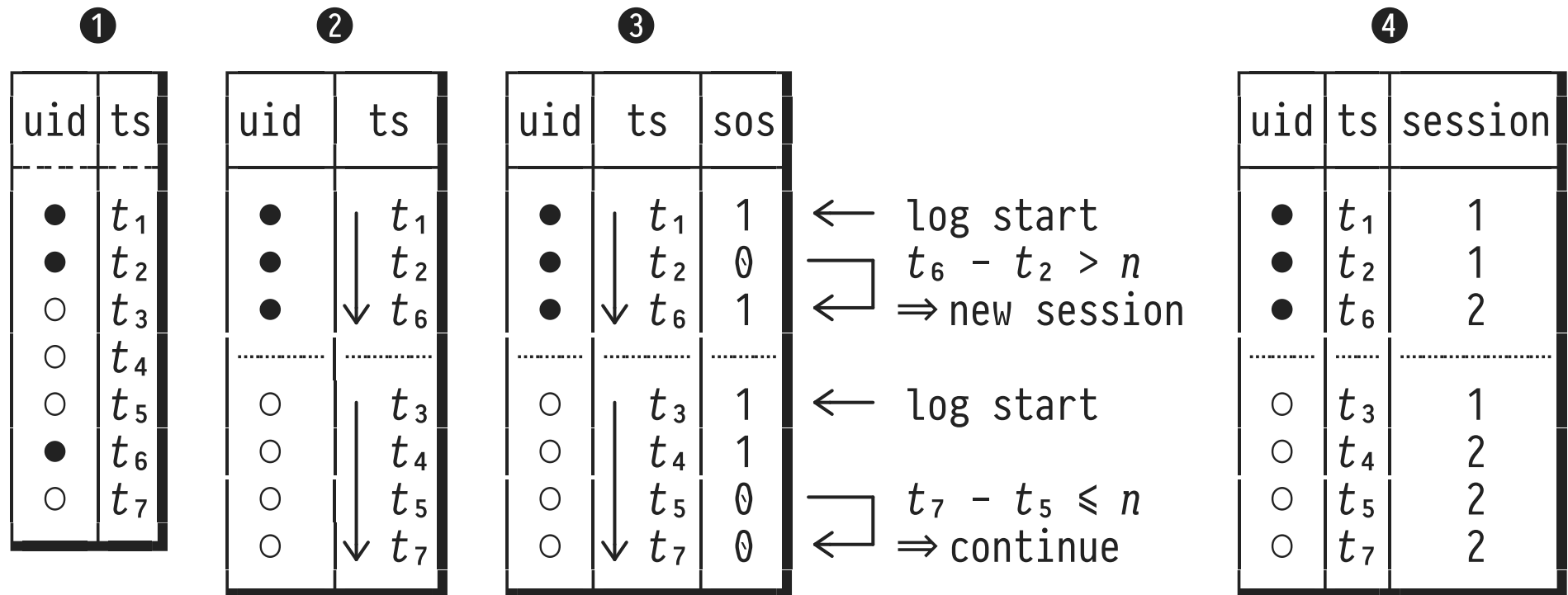
1. Cop and spy sessions happen independently (even if interleaved): partition **log** into 🕵️/● and 🕵️/○ rows.
2. **Tag** keyboard activities (below: tagging for ●):

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



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

🔧 Sessionization (Query Plan)



- 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:

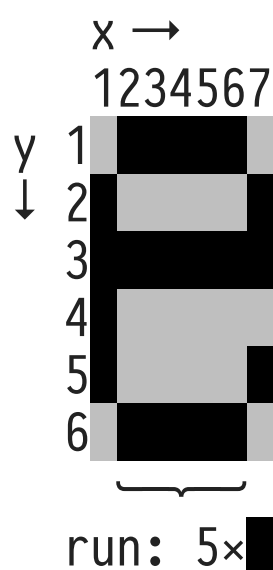


Table **original**

x	y	pixel
1	1	gray
2	1	black
⋮	⋮	⋮
6	6	black
7	6	gray



Table **encoding**

run	length	pixel
1	1	gray
2	5	black
⋮	⋮	⋮
12	5	black
13	1	gray

- 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
⋮	⋮	⋮	⋮

Diagram: A vertical bar with a light gray top and dark gray bottom. A bracket on the right side of the bar spans from the first row to the seventh row, with a question mark below it. An arrow points from the first row to the second row.

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
⋮	⋮	⋮	⋮	⋮

Diagram: A vertical bar with a light gray top and dark gray bottom. A bracket on the right side of the bar spans from the second row to the sixth row, labeled "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:

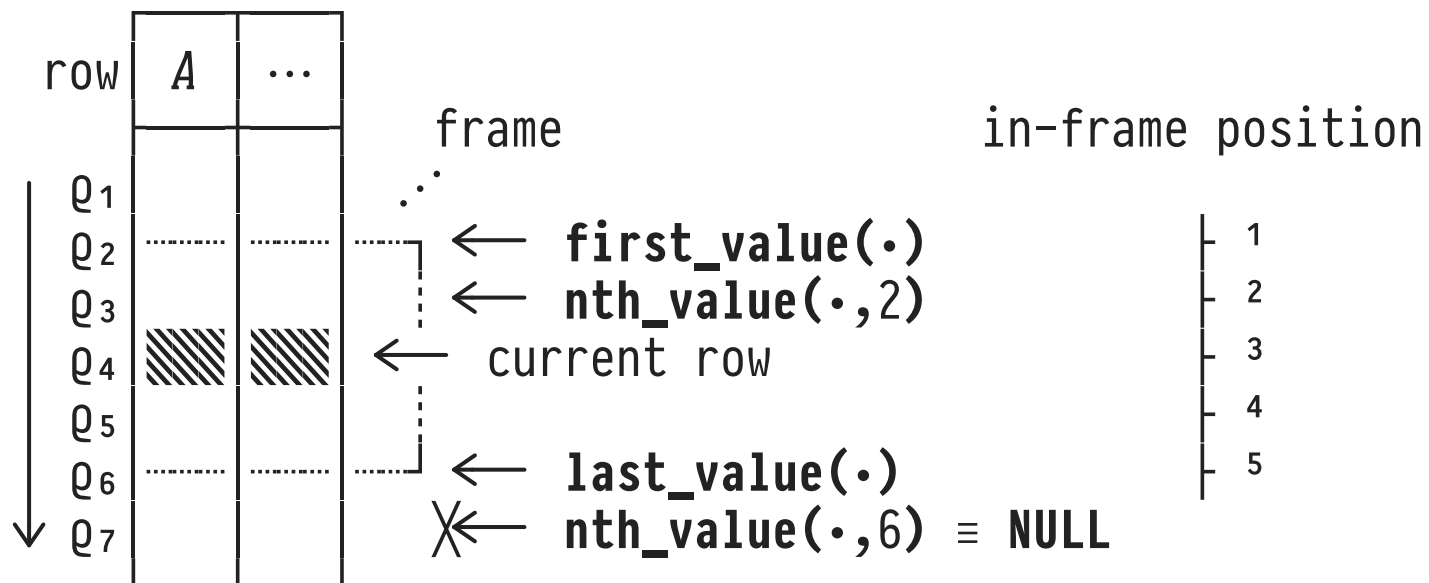
- **Positional access to individual rows** inside a frame is provided by three window functions:

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

$\left. \begin{array}{l} \text{first_value}(e) \\ \text{last_value}(e) \\ \text{nth_value}(e,n) \end{array} \right\}$	OVER (...)
--	-------------------

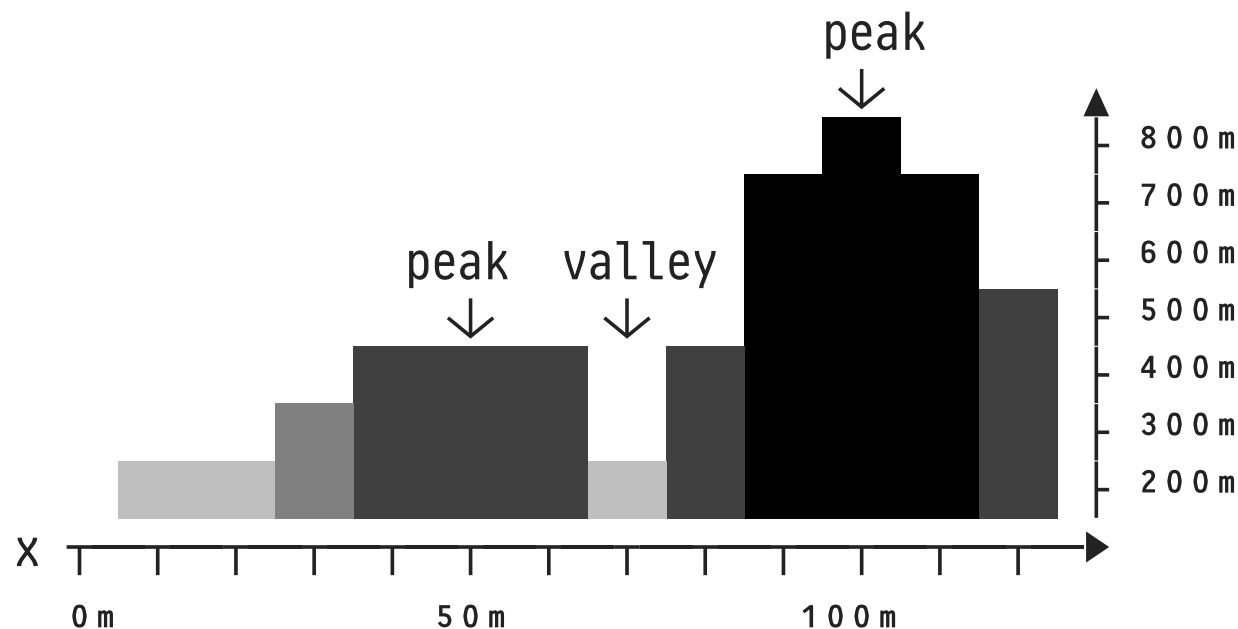
- **`nth_value(e,n)`**: No n^{th} row in frame \Rightarrow return **NULL**.

In-Frame Row Access



- We have `first_value(e) ≡ nth_value(e,1)`.

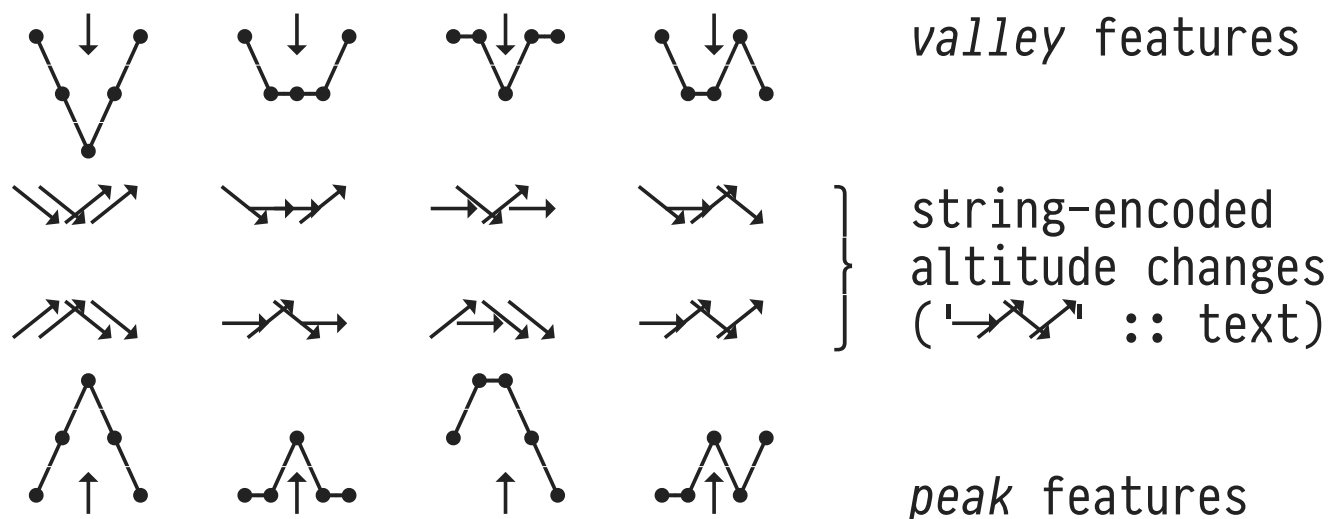
🔧 Detecting Landscape Features



- 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)

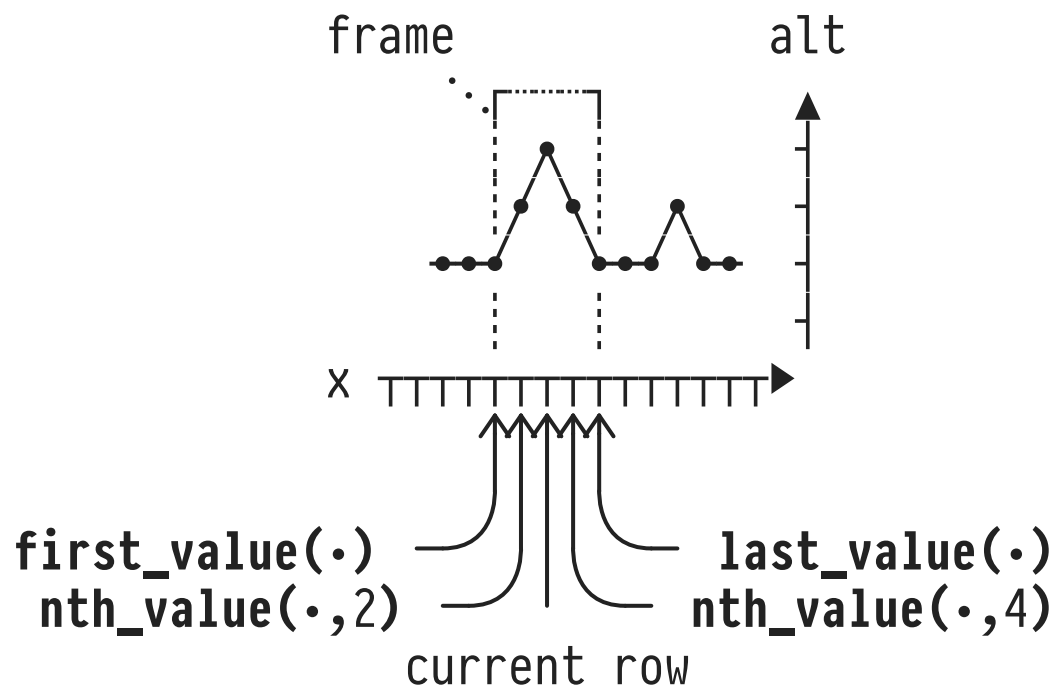
1. Track relative altitude changes in a sliding **x-window** of size 5:



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):



- `first_value(alt) < nth_value(alt,2) \Rightarrow 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 alt 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 December 2025, not supported by .
(Work is underway here in Tübingen to rectify this.)

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), -- } row tags
         EVEN    AS EVEN.alt = PREV(EVEN.alt) --
)

```

Output

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

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()	}	-- intra-partition ranking 👍
dense_rank()		--
rank()		OVER ([PARTITION BY p_1, \dots, p_m]
		[ORDER BY e_1, \dots, e_n])
percent_rank()		--
cume_dist()		-- ranking w/o ORDER BY 👎
ntile(n)		

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

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

Table W f

row	A	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` (but see SQL filtering clause `QUALIFY`).

🔧 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

**Output**

ref	range
1	r_0
2	r_0
⋮	⋮
42	r_4

← 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	
		⋮				
		subtract				

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

row	A	percent_rank	cume_dist	ntile(3)
q1	1	0	1/9	1
q2	2	1/8	2/9	1
q3	3	2/8	5/9	1
q4	3	2/8	5/9	2
q5	3	2/8	5/9	2
q6	4	5/8	6/9	2
q7	6	6/8	8/9	3
q8	6	6/8	8/9	3
q9	7	8/8	9/9	3

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

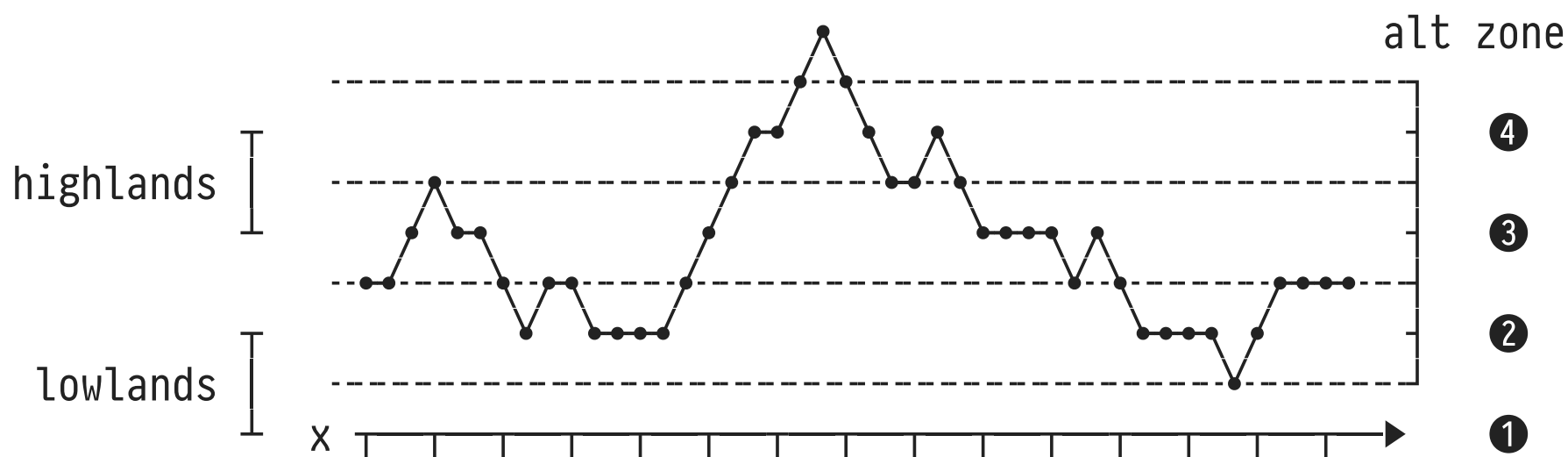
← current row is in the n^{th}
of 3 chunks of rows

$n\%$ 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

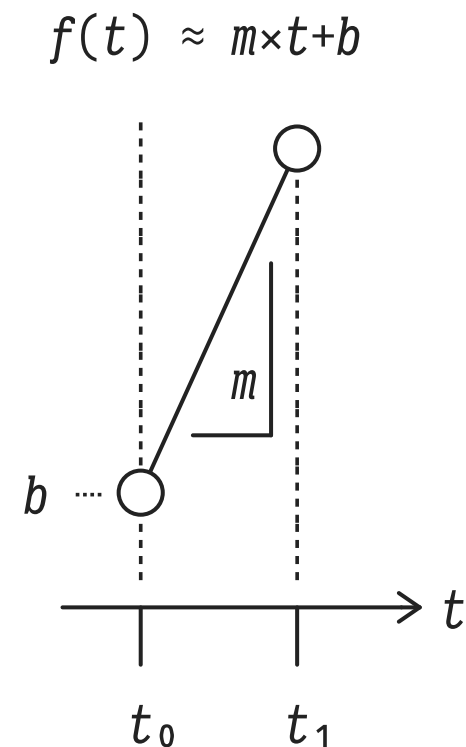
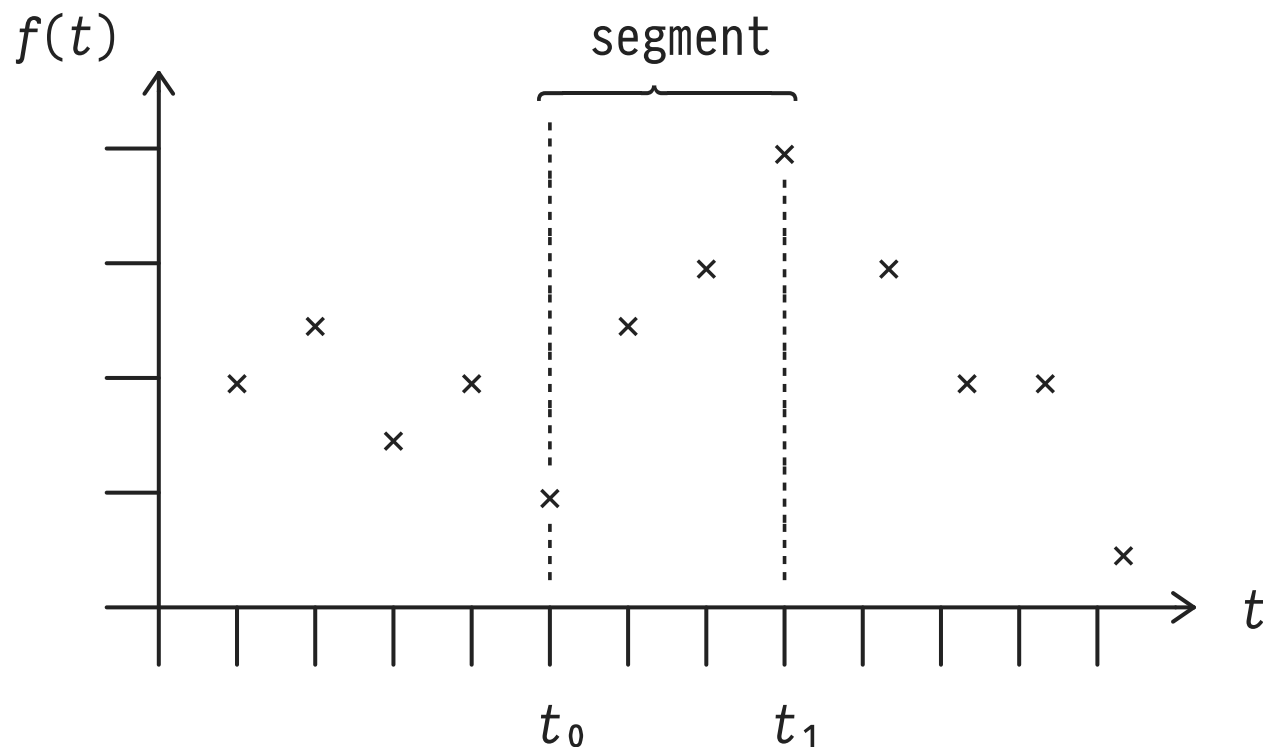
- Classify the altitudes of a mountain range into
 - equal-sized** vegetation **zones** and
 - lowlands (altitude in the lowest **20%**) and highlands (between **60%-80%** of maximum altitude).



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 Semantics²

Scope	Computation	Function	Description
frame	aggregation row access	(aggregates) <code>first_value(e)</code> <code>last_value(e)</code> <code>nth_value(e,n)</code>	<code>sum</code> , <code>avg</code> , <code>max</code> , <code>list</code> , ... <i>e</i> at first row in frame <i>e</i> at last row in frame <i>e</i> at n^{th} row in frame
partition	row access ranking	<code>lag(e,n,d)</code> <code>lead(e,n,d)</code> <code>row_number()</code> <code>rank()</code> <code>dense_rank()</code> <code>percent_rank()</code> <code>cume_dist()</code> <code>ntile(n)</code>	<i>e</i> at <i>n</i> rows <i>before</i> current row <i>e</i> at <i>n</i> rows <i>after</i> current row number of current row rank with gaps (“Olympics”) rank without gaps relative rank of current row ratio of rows up to “—” rank on a scale $\{1,2,\dots,n\}$

² `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.