Advanced SQL

05 — Window Functions

Summer 2022

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	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

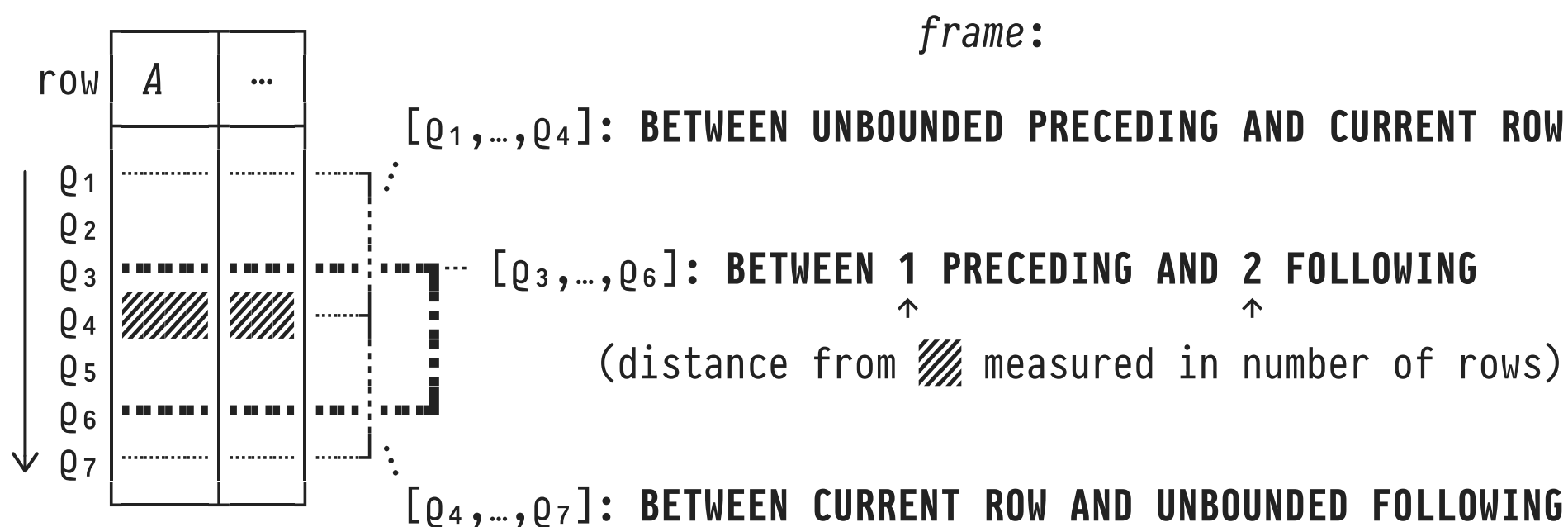| row	A	...
ϱ_1	v_1	
ϱ_2	v_1	
ϱ_3	v_3	
ϱ_4	v_3	
ϱ_5	v_3	
ϱ_6	v_4	
ϱ_7	v_5	
ϱ_8	v_5	

vicinity
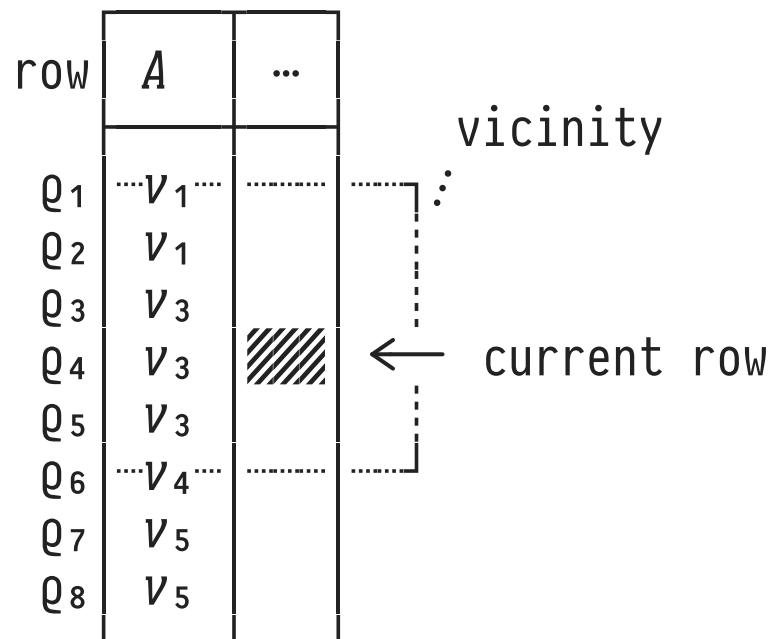
← current row

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

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

  ❶ row **position** (ROWS windows),
  ❷ row **values** v_i (RANGE windows),
  ❸ 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

$$f \;\; \textbf{OVER (ORDER BY } e_1, \ldots, e_n \; [\; \textbf{ROWS } \textit{frame} \;])$$

frame:

row | A | ...

$[\varrho_1, \ldots, \varrho_4]$: **BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**

ϱ_1
ϱ_2
ϱ_3
ϱ_4
ϱ_5
ϱ_6
ϱ_7

$[\varrho_3, \ldots, \varrho_6]$: **BETWEEN 1 PRECEDING AND 2 FOLLOWING**
↑   ↑

(distance from ▨ measured in number of rows)

$[\varrho_4, \ldots, \varrho_7]$: **BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING**
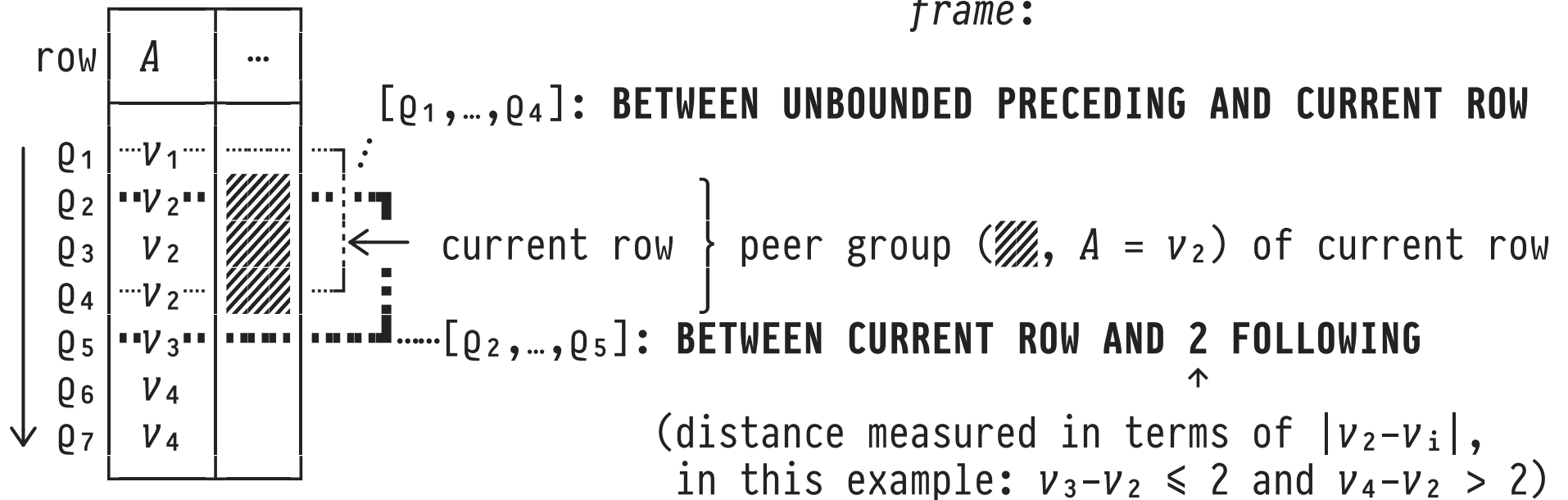
Window Frame Specifications (Variant: RANGE)

window function   one criterion   frame specification

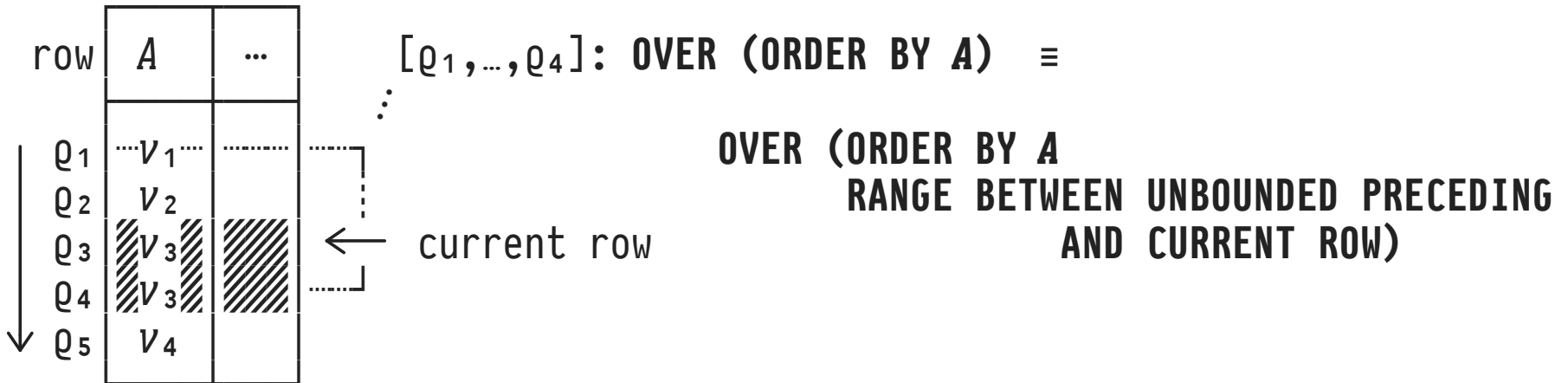f **OVER** (**ORDER BY** A [**RANGE** *frame*])

frame:

row | A | ⋯

ϱ_1 v_1
ϱ_2 v_2
ϱ_3 v_2
ϱ_4 v_2
ϱ_5 v_3
ϱ_6 v_4
ϱ_7 v_4

$[\varrho_1,…,\varrho_4]$: **BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**

← current row } peer group (▨, $A = v_2$) of current row

$[\varrho_2,…,\varrho_5]$: **BETWEEN CURRENT ROW AND 2 FOLLOWING**

↑

(distance measured in terms of $|v_2-v_i|$,
in this example: $v_3-v_2 \leqslant 2$ and $v_4-v_2 > 2$)

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

window function   ordering criteria   frame specification

$$f \;\; \textbf{OVER} \; (\textbf{ORDER BY} \; e_1,\ldots,e_n \; [\; \textbf{GROUPS} \; \textit{frame} \;])$$

frame:

$[\varrho_2,\ldots,\varrho_6]$: **BETWEEN 1 PRECEDING AND 1 FOLLOWING**

↑   ↑

(distance measured in number of peer groups)

← current row } peer group (▨, $A = v_3$) of current row

$[\varrho_4,\ldots,\varrho_7]$: **BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING**

row	A	...
ϱ_1	v_1	
ϱ_2	v_2	
ϱ_3	v_2	
ϱ_4	v_3	▨
ϱ_5	v_3	▨
ϱ_6	v_4	
ϱ_7	v_5	

Window Frame Specifications: Abbreviations

$[\varrho_1,\ldots,\varrho_5]$: **OVER ()** ≡ all rows form the window frame
(for any current row)

← current row

$[\varrho_1,\ldots,\varrho_4]$: **OVER (ORDER BY A)** ≡

OVER (ORDER BY A
RANGE BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)

← current row

row | A | ...
ϱ_1
ϱ_2
ϱ_3
ϱ_4
ϱ_5

row | A | ...
ϱ_1 | v_1
ϱ_2 | v_2
ϱ_3 | v_3
ϱ_4 | v_3
ϱ_5 | v_4

WINDOW Frame Specifications: Exclusion[1]

window function   exclusion clause

$$f \quad \textbf{OVER} \ (\textbf{ORDER BY} \ A \ frame \ [\ exclusion \ ])$$

row	A	...
ϱ_1	v_1	
ϱ_2	v_2	
ϱ_3	v_3	
ϱ_4	v_3	
ϱ_5	v_3	
ϱ_6	v_4	

← current row

exclusion:

EXCLUDE NO OTHERS $[\varrho_2, \varrho_3, \varrho_4, \varrho_5]$

EXCLUDE CURRENT ROW $[\varrho_2, \varrho_3, \quad \varrho_5]$

EXCLUDE GROUP $[\varrho_2 \qquad\qquad]$

EXCLUDE TIES $[\varrho_2, \quad \varrho_4 \quad]$

[1] **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",
       array_agg(w.row) OVER win AS "rows in frame"
FROM   W AS w
WINDOW win AS (frame)
```

row	a	b
ϱ₁	1	●
ϱ₂	2	○
ϱ₃	3	○
ϱ₄	3	●
⋮	⋮	⋮

Table W

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

Input: Daily weather readings in sensors:

day	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 pᵢ form one partition
                  ⌒‾‾‾‾‾‾‾‾‾‾‾‾‾‾⌒
  f OVER ([ PARTITION BY p₁,…,pₘ ]
          [ ORDER BY e₁,…,eₙ ]
          [ frame ])
```

- Note:

1. Frames **never cross partitions.**
2. BETWEEN ⋯ PRECEDING AND ⋯ FOLLOWING **respects partition boundaries.**

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

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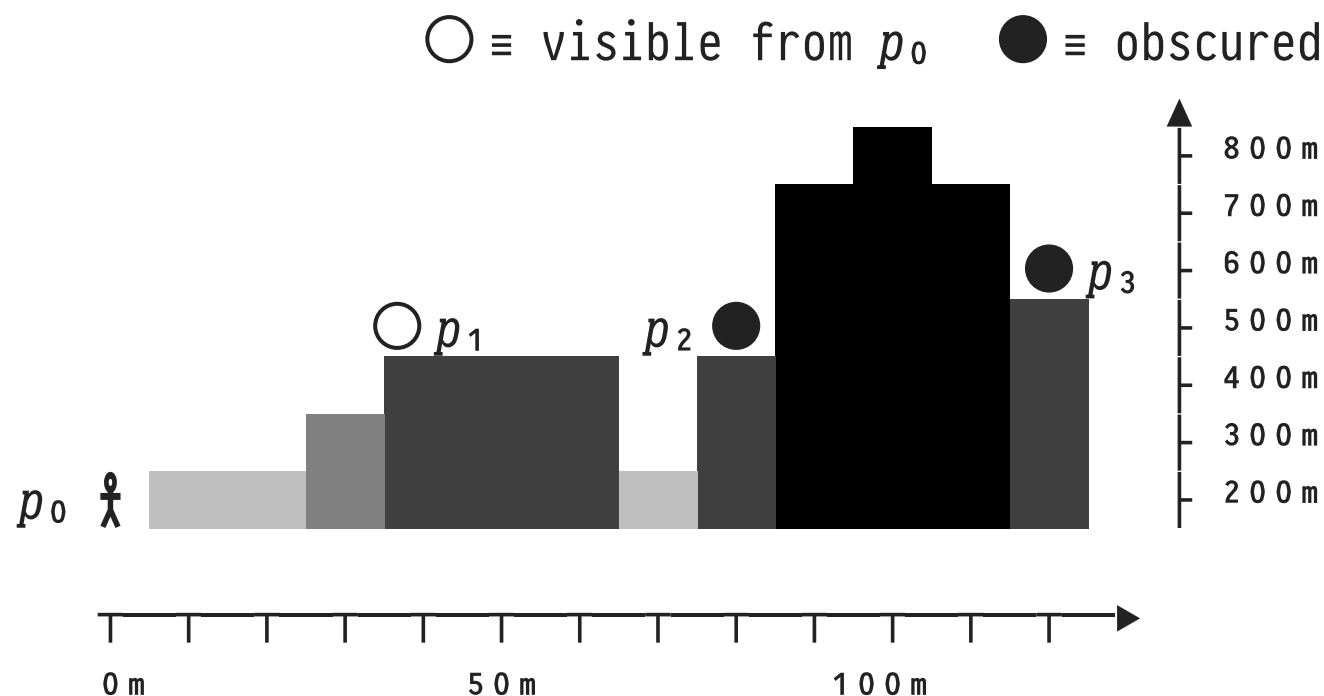
- From the viewpoint of p_0 (🧍) we can see p_1, but...
  - ... p_2 is **obscured** (no straight-line view from p_0),
  - ... p_3 is **obscured** (lies behind the 800m peak).

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

∡α = atan(Δalt/Δx)
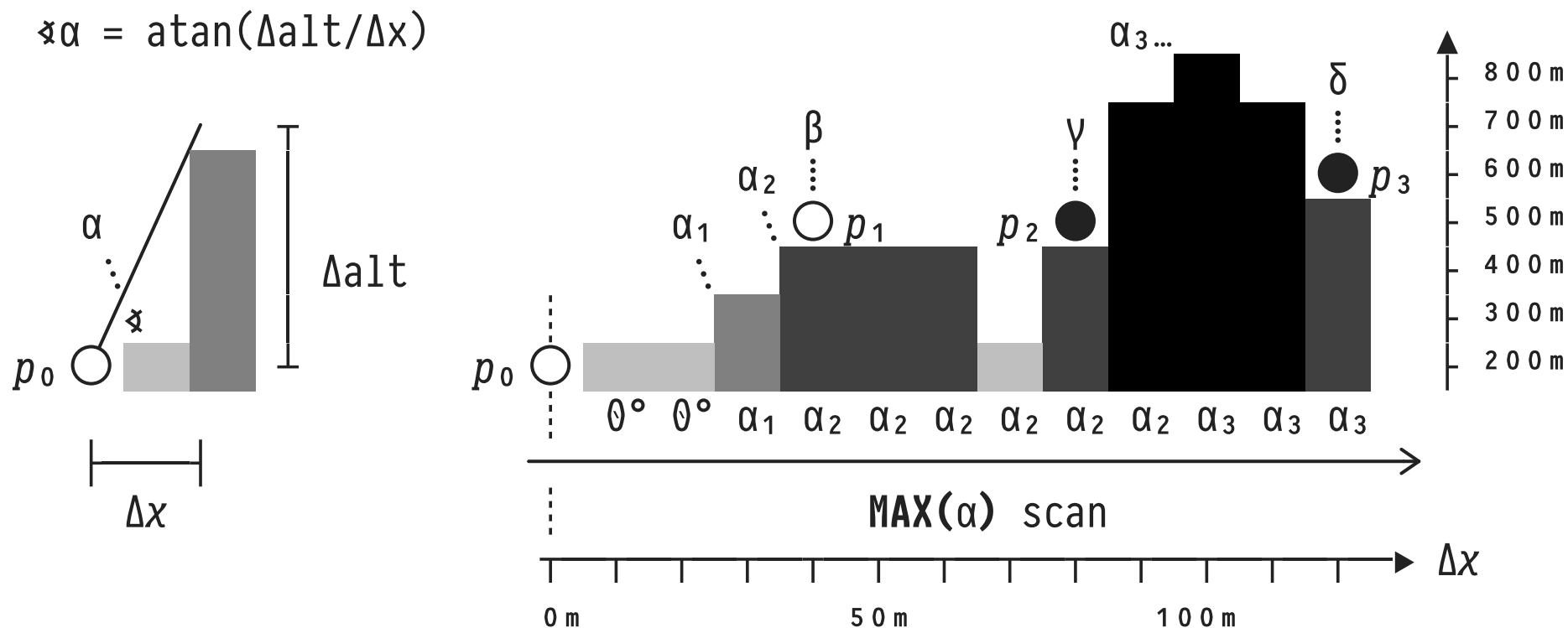
- We have $0° < α_1 < α_2 < α_3$ and $β \geqslant α_2$, $γ < α_2$, $δ < α_3$.

  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
-- 1 Angles α (in °) between p₀ and the hilltop at x
angles(x, angle) AS (
  SELECT m.x,
         degrees(atan((m.alt - p0.alt) /
                      abs(p0.x - m.x))) AS angle
  FROM   map AS m
  WHERE  m.x > p0.x
),
-- 2 MAX(α) scan (to the right of p₀)
max_scan(x, max_angle) AS (
  SELECT a.x,
         MAX(a.angle)
           OVER (ORDER BY abs(p0.x - a.x)) AS max_angle
  FROM   angles AS a
),
⋮
```

🔧 Looking Left *and* Right: PARTITION BY
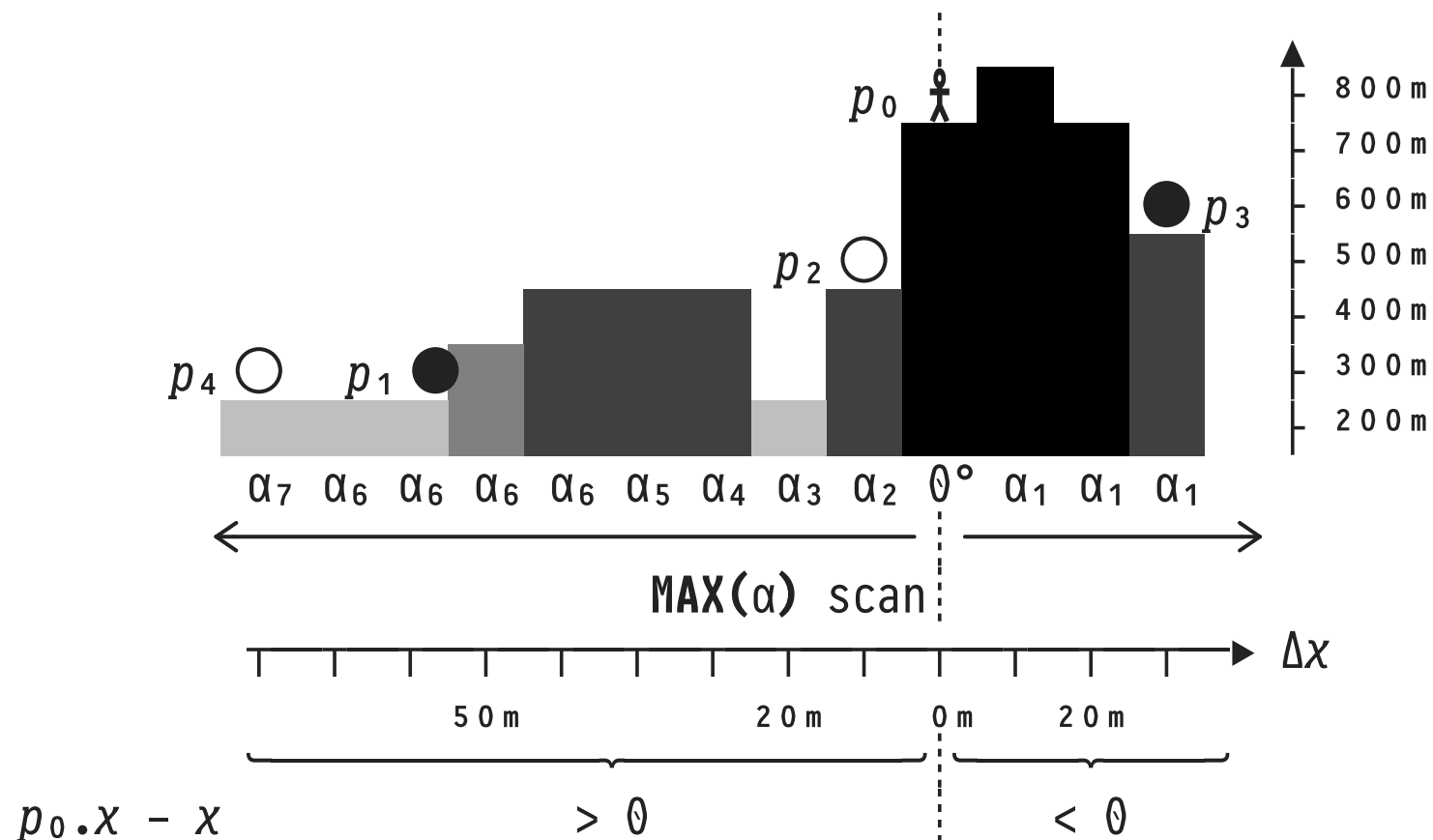
- Need MAX scans left *and* right of p_0 ⟹ use PARTITION BY.

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

```
WITH
  ⋮
-- ❷ MAX(α) scan (left/right of p₀)
max_scan(x, max_angle) AS (
  SELECT a.x,                  --    ∈ {-1, 0 ,1}
         MAX(a.angle)          --   ⌒‾‾‾‾‾⌄‾‾‾‾‾⌒
           OVER (PARTITION BY sign(p0.x - a.x)
                 ORDER BY abs(p0.x - a.x)) AS max_angle
  FROM   angles AS a  --   ⌒‾‾‾‾‾⌄‾‾‾‾‾⌒
                      --        Δx > 0
),
  ⋮
```

- $\forall a \in$ angles: a.x ≠ p0.x ⟹ We end up with **two** partitions.

3 ┊ Scans: Not Only in the Hills

Scans are a general and expressive computational pattern:

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

- Available in a variety of forms in programming languages
  - Haskell: scanl $z \oplus xs$, APL: $\oplus\backslash xs$, Python: accumulate.
    scanl $\oplus$ z $[x_1,x_2,\dots]$ = $[z,\ z \oplus x_1,\ (z \oplus x_1) \oplus x_2,\ \dots]$

- 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* ≡ '((b*2)−4×a×c)*0.5'. What is computed below?

```
SELECT inp.pos, inp.c,
       SUM((array[1,-1,0])[COALESCE(p.oc, 3)])
         OVER (ORDER BY inp.pos) AS d
FROM   string_to_table(:'xs', NULL)
         WITH ORDINALITY AS inp(c,pos),
       LATERAL (VALUES (array_position(array['(',')'],
                            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 $p_1,…,p_m$ ]
              [ ORDER BY $e_1,…,e_n$ ]
              [ frame ])
```

Three kinds of **window functions** f:

1. **Aggregates:** SUM(•), AVG(•), MAX(•), array_agg(•), …
   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
--
        LAG(e,n,d) OVER ([ PARTITION BY p₁,…,pₘ ]
                          ORDER BY e₁,…,eₙ)
```

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 $\pm n$ ⇒ return value d.

LAG/LEAD: Row Offsets (Assume: No Partitioning, ORDER BY A)
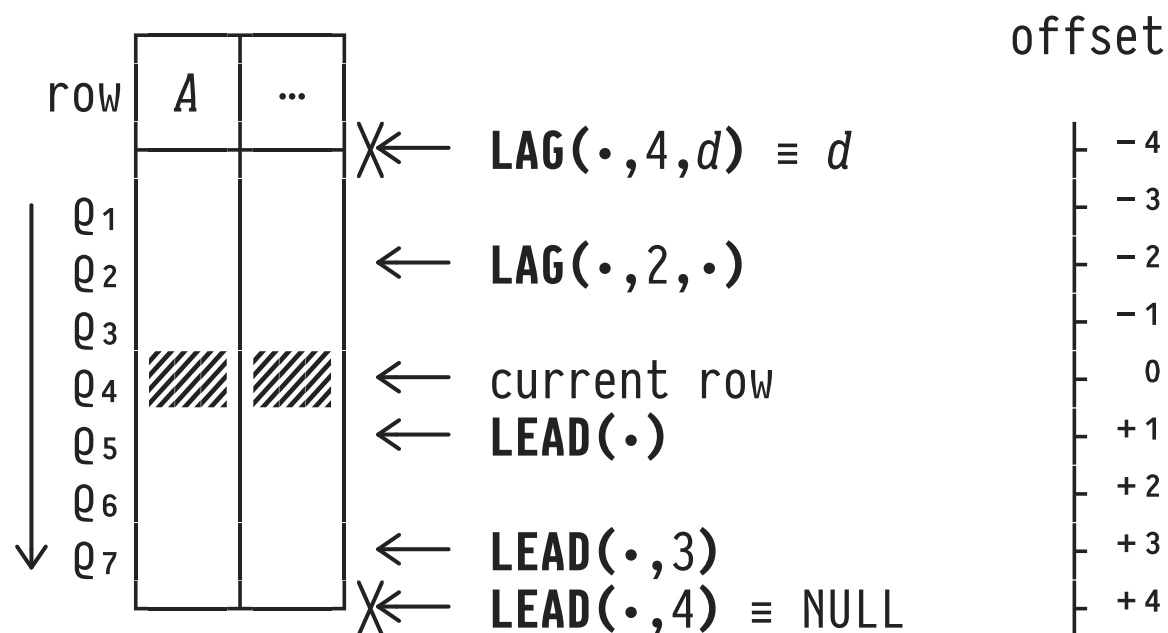
- The frame of the current row is irrelevant for LAG/LEAD.
- Default parameters if absent: $n \equiv 1$, $d \equiv$ NULL.

🔧 A March Through the Hills: Ascent or Descent?

```sql
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:

uid	ts
👤	05-25-2020 07:25:12
👤	05-25-2020 07:25:18
🕵	05-25-2020 08:01:55
🕵	05-25-2020 08:05:07
🕵	05-25-2020 08:05:30
👤	05-25-2020 08:05:39
🕵	05-25-2020 08:05:46

Table log

- **Q:** Can we **sessionize** the log so that investigators can identify *sessions* (≡ 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 ●):

$t_2 - t_1 \leq$ threshold $n \Rightarrow$ continue session (tag t_2 with **0**)
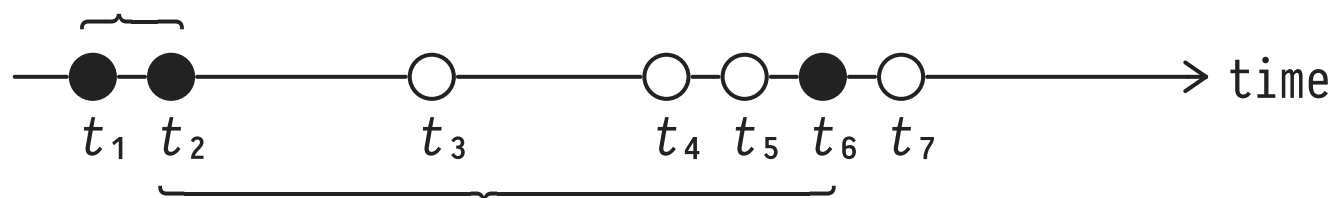
$t_6 - t_2 > n \Rightarrow$ new session (tag t_6 with **1**)

3. **Scan** the tagged table and derive session IDs by maintaining a **runnning 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
$t_6 - t_2 > n$
⇒ new session

← log start

$t_7 - t_5 \leqslant n$
⇒ 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
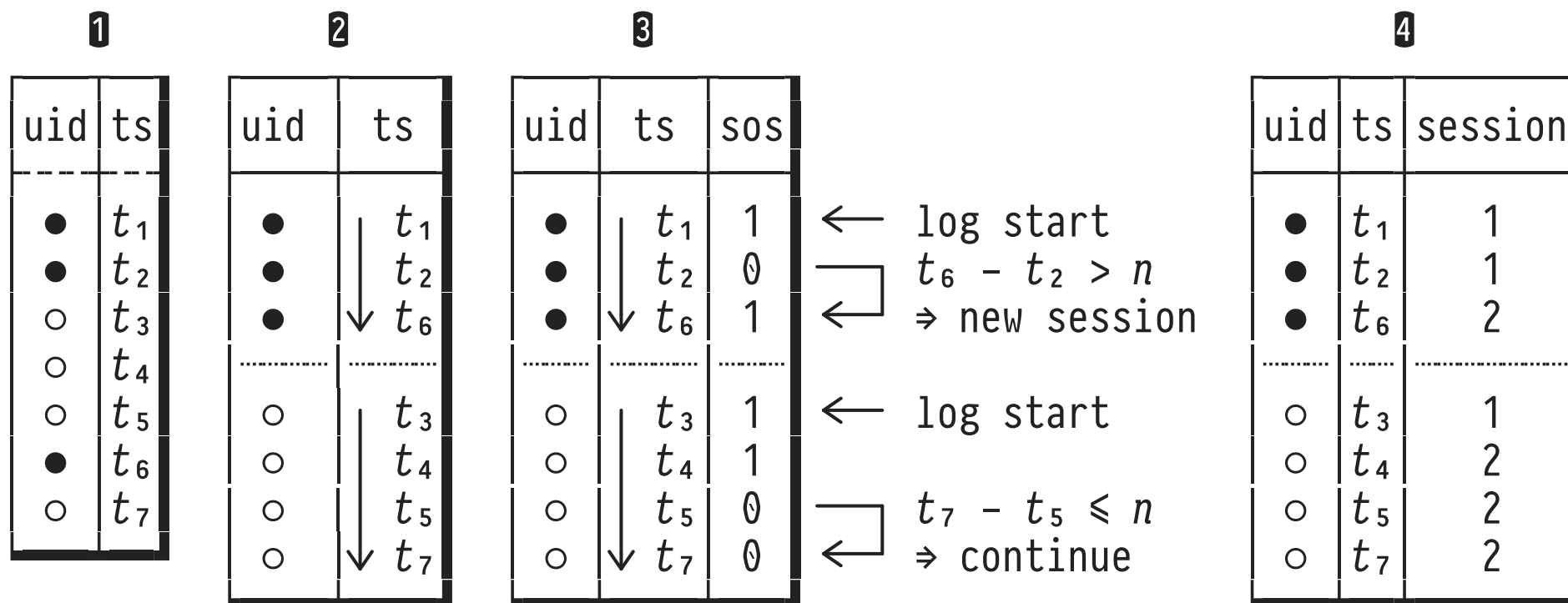
- At log start, always begin a new session (sos = 1).
- How to assign *global session IDs* (○'s sessions: 3, 4)?

🔧 Image Compression by Run-Length Encoding

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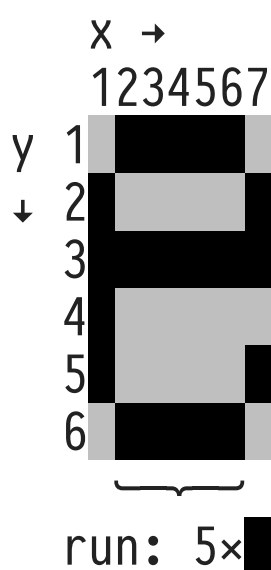
Table **original**

x	y	pixel
1	1	
2	1	
⋮	⋮	⋮
6	6	
7	6	

→

Table **encoding**

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

run: 5×■

- 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)

❶

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
⋮	⋮	⋮	⋮	⋮

❷

x	y	pixel	change?	Σ change?	
1	1		1	1	
2	1		1	2	
3	1		0	2	
4	1		0	2	run #2 of
5	1		0	2	length 5
6	1		0	2	
7	1		1	3	
⋮	⋮	⋮	⋮	⋮	

❶: LAG(pixel,1,*undefined*): pixel @ (1,1) always "changes."

❷: 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/nth row in the frame
--
           FIRST_VALUE(e)
           LAST_VALUE(e)      } OVER (…)
           NTH_VALUE(e,n)
```

- NTH_VALUE(e,n): No n^{th} row in frame ⇒ return NULL.

In-Frame Row Access
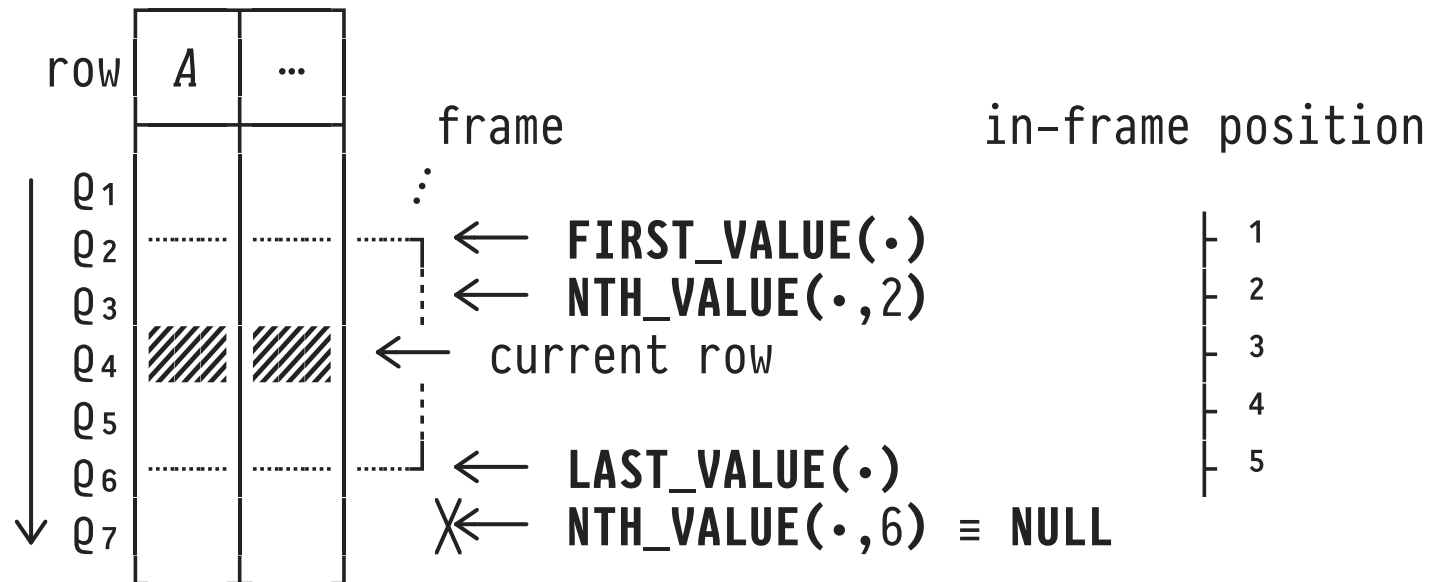
- We have FIRST_VALUE(*e*) ≡ NTH_VALUE(*e*,1).

🔧 Detecting Landscape Features
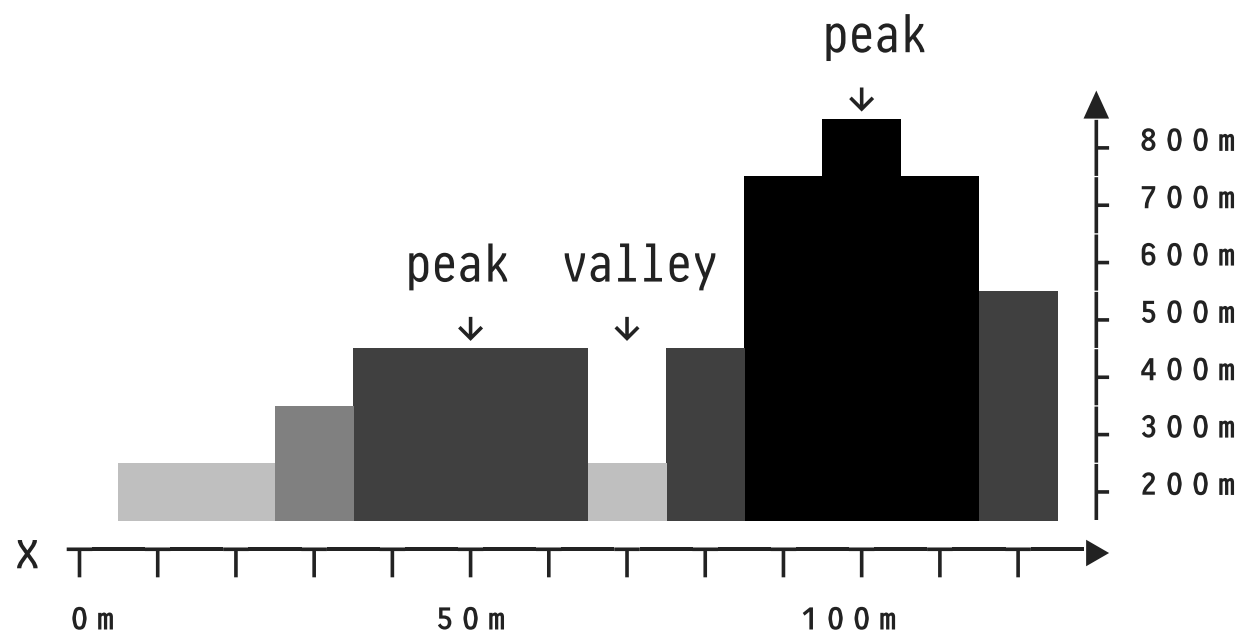
- Detect features in a hilly landscape. Attach label ∈ {peak,valley,-} to every location x.

- 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:**

valley features

string-encoded
altitude changes
('→↗↘↗' :: text)

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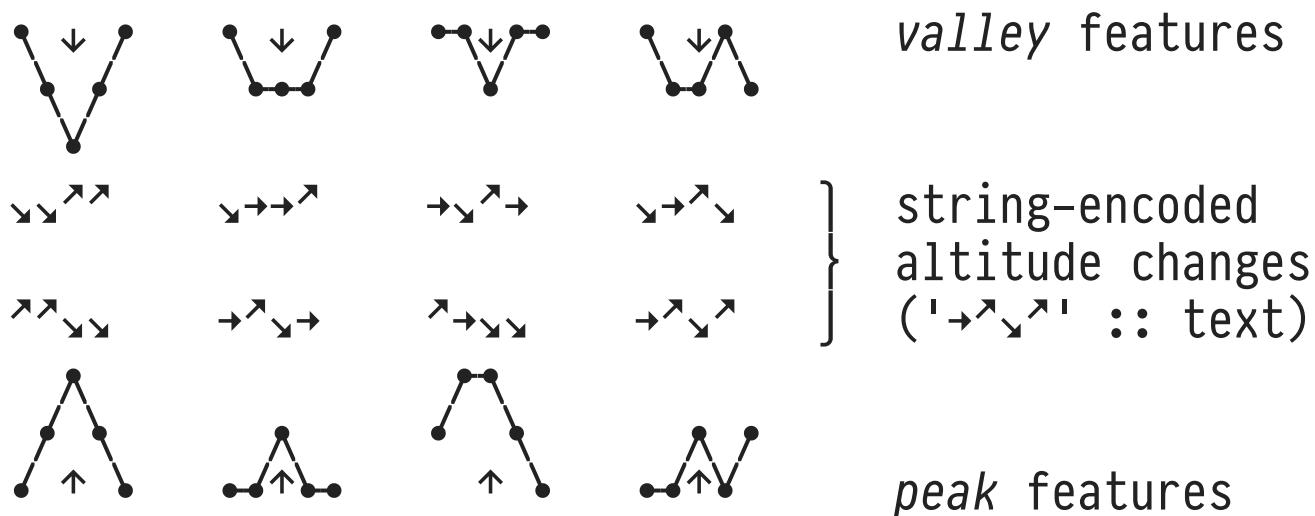
2. **Pattern match** on change strings to detect features.

🔧 **Altitude Changes in a Sliding Window**

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

frame   alt

FIRST_VALUE(·)   LAST_VALUE(·)
NTH_VALUE(·,2)   NTH_VALUE(·,4)

current row
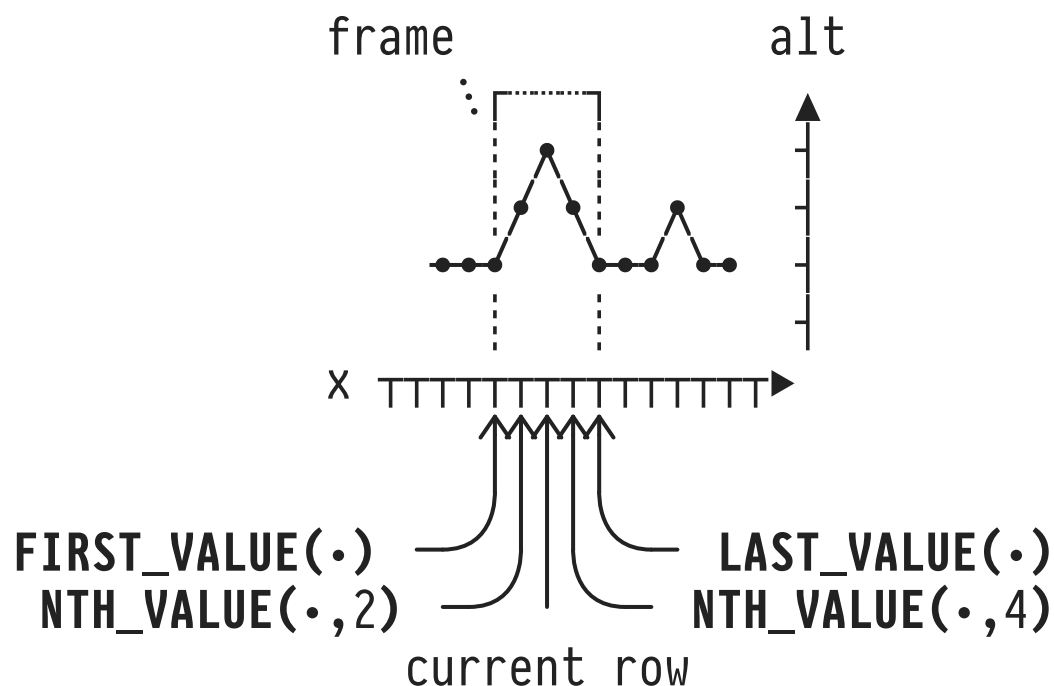
- FIRST_VALUE(alt) < NTH_VALUE(alt,2) ⟹ ascent ('↗').

🔧 **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),
 sign(NTH_VALUE(m.alt,2) OVER w-m.alt                      ),
 sign(m.alt                      -NTH_VALUE(m.alt,4) OVER w),
 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).
- UDF encodes altitude changes: slope(-1,-1,0,1) ≡ '↘↘→↗'.

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 *                              Output
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)  -- ⎭
)
```

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 p₁,…,pₘ ]
                      [ ORDER BY e₁,…,eₙ ] )
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
ϱ1	1	1	1	1
ϱ2	2	2	2	2
ϱ3	**3**	3	3	3
ϱ4	**3**	4	3	3
ϱ5	**3**	5	3	3
ϱ6	4	6	4	6
ϱ7	**6**	7	5	7
ϱ8	**6**	8	5	7
ϱ9	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: DENSE_RANK() ≤ RANK() ≤ ROW_NUMBER()

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

species	length	height	legs
⋮	⋮	⋮	$\in$ {2,4,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

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**					
ref	ref	**ROW_NUMBER()**				
5	1	–	1	=	0	⎤
2	2	–	2	=	0	⎬ range 0 ≡ r_0
14	3	–	3	=	0	⎦
3	5	–	4	=	1	⎤
1	6	–	5	=	1	⎬ range 1 ≡ r_1
42	7	–	6	=	1	⎦
6	10	–	7	=	3	⎬ range 3 ≡ r_2
10	13	–	8	=	5	⎤ range 5 ≡ r_3
7	14	–	9	=	5	⎦
13	42	–	10	=	32	⎬ range 32 ≡ r_4

subtract

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

row	A	PERCENT_RANK	CUME_DIST	NTILE(3)
ϱ1	1	0	$1/9$	1
ϱ2	2	$1/8$	$2/9$	1
ϱ3	**3**	$2/8$	$5/9$	1
ϱ4	**3**	$2/8$	$5/9$	2
ϱ5	**3**	$2/8$	$5/9$	2
ϱ6	4	$5/8$	$6/9$	2
ϱ7	**6**	$6/8$	$8/9$	3
ϱ8	**6**	$6/8$	$8/9$	3
ϱ9	7	$8/8$	$9/9$	3

⋯ Rows that agree on
⋯ the sort criterion
⋯ (here: A) rank equally
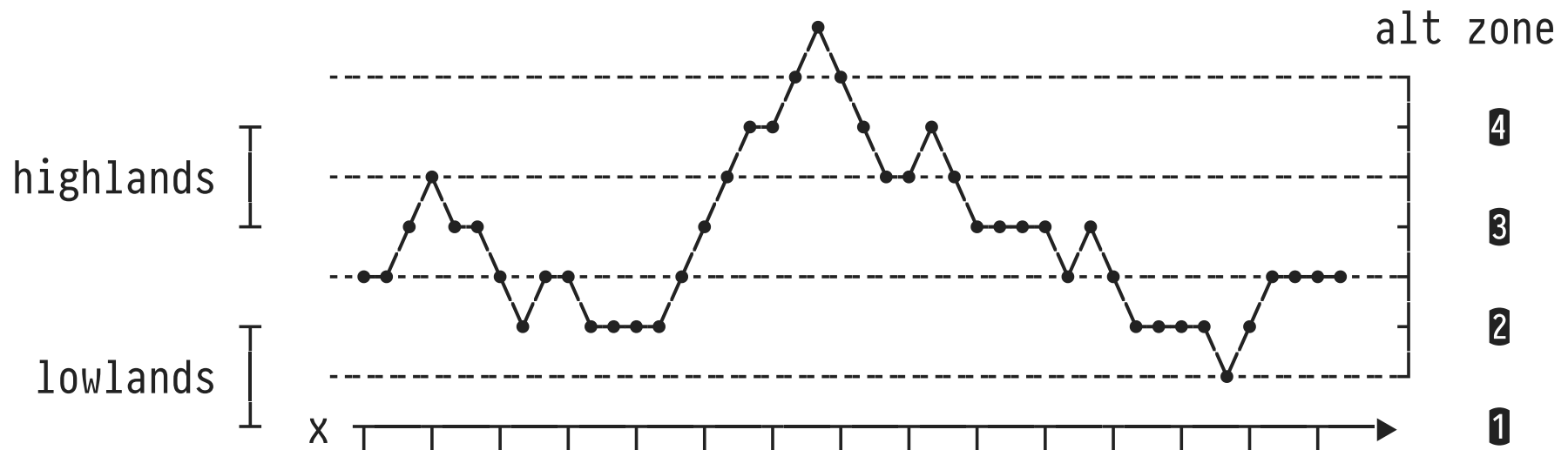
← current row is in the nth
of 3 chunks of rows

$n\%$ 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
  1. **equal-sized** vegetation **zones** and
  2. lowlands (altitude in the lowest **20%**) and highlands (between **60%-80%** of maximum altitude).

🔧 Altitudinal Mountain Zones

```sql
-- 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
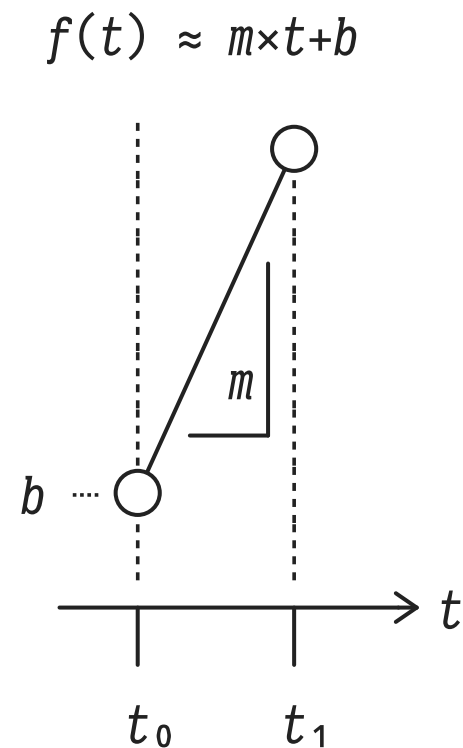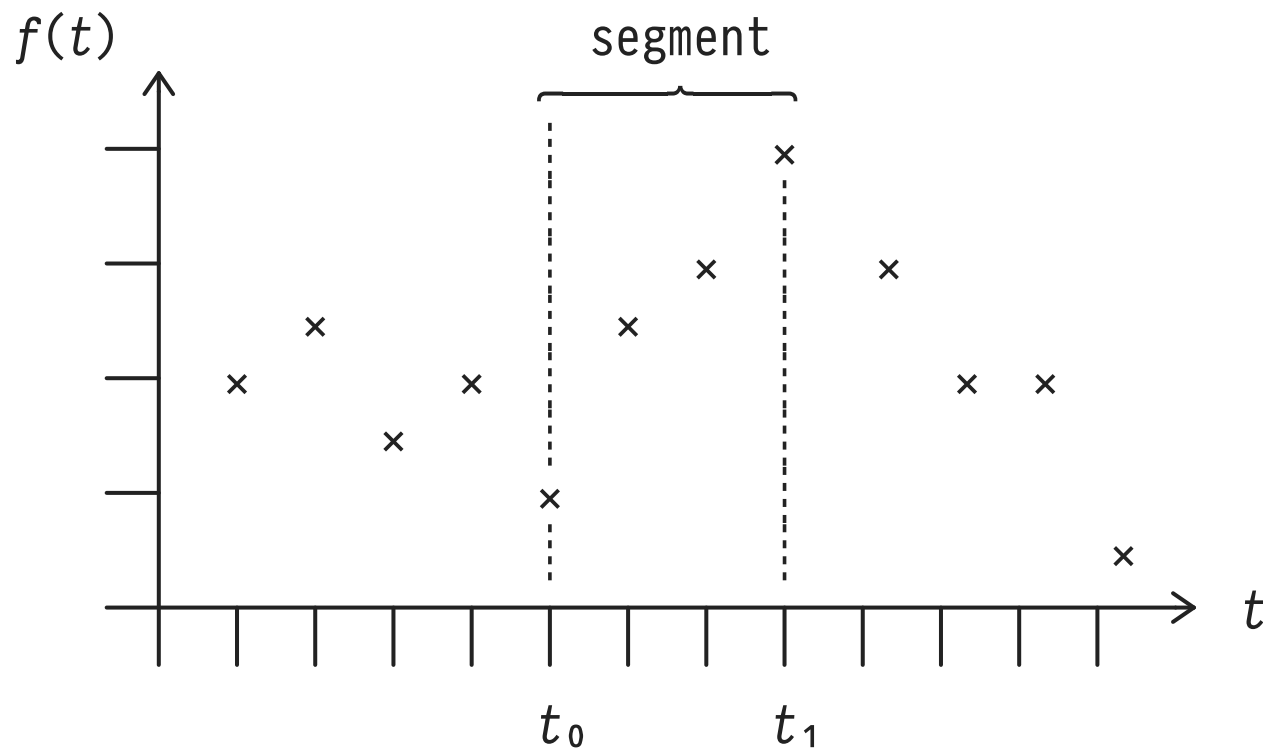
1. NTILE(n) **segments** time series at desired granularity.
2. Compute m, b in each **segment ≡ window frame.**

9 ⋮ Summary: Window Function Semantics[2]

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