Advanced SQL

05 — Window Functions

Summer 2020

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:

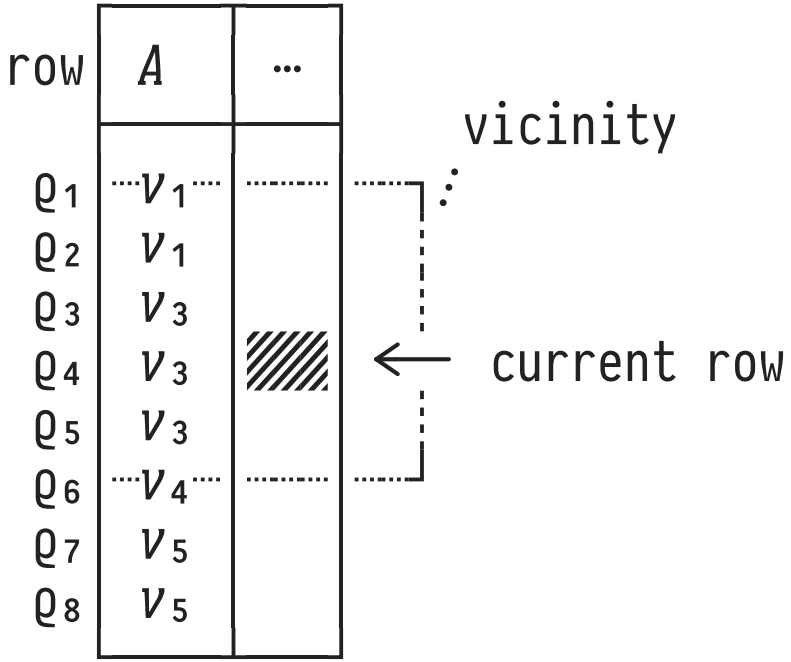
SQL Feature	Mode of Computation
function	row ⟶ row
table-generating function	row ⟶ table of rows
aggregate	group of rows ⟶ row (one per group)
window function 👈	row vicinity ⟶ row (one per row)

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



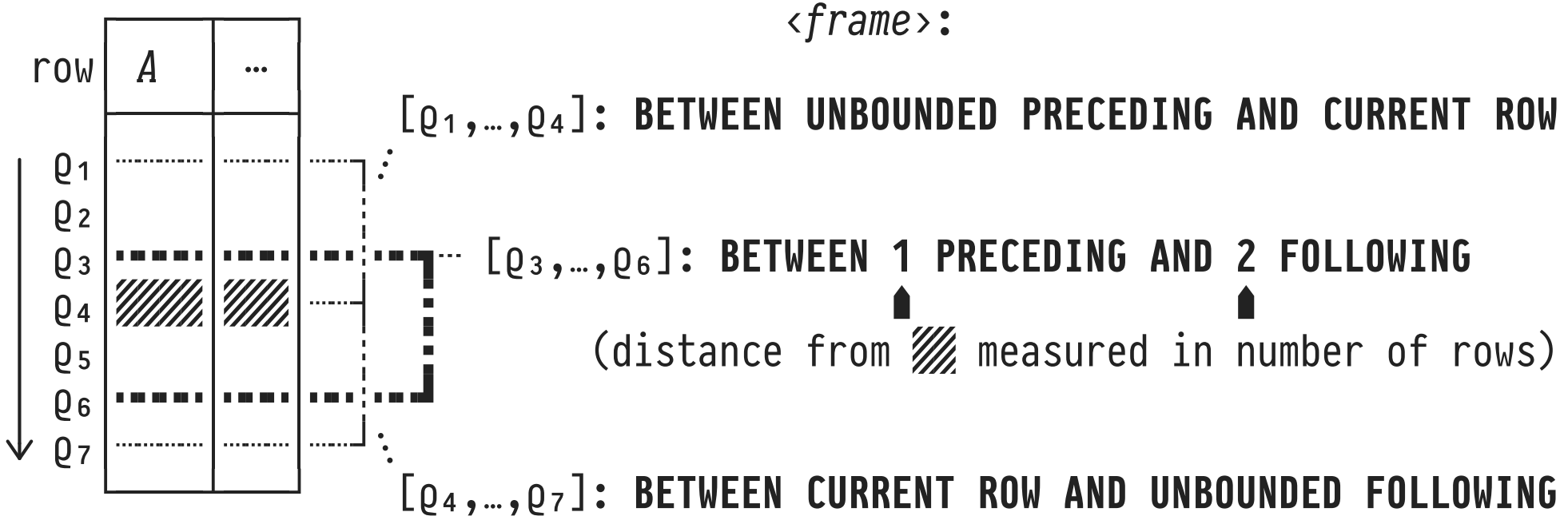
- Each row is the **current row** ▨ at one point in time.
- Row vicinity (**window**, **frame**) is based on either:
  1. row **position** (ROWS windows),
  2. row **values** v_i (RANGE windows),
  3. row **peers** (GROUPS windows).

- 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

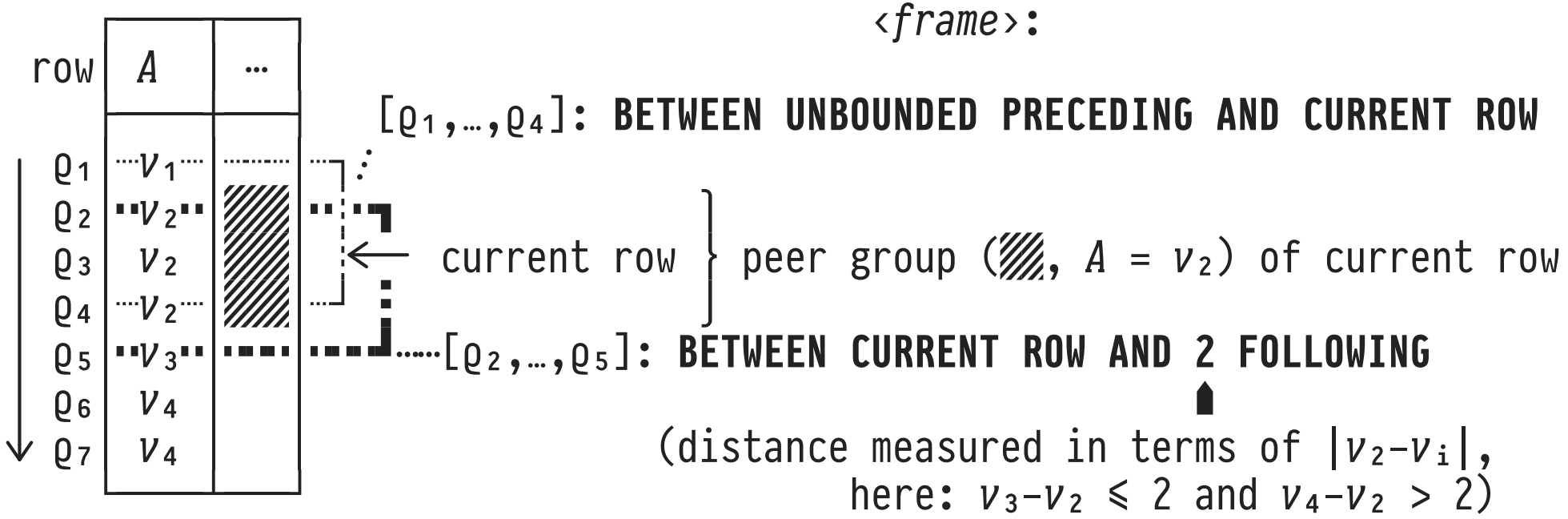
⟨*f*⟩ **OVER (ORDER BY** ⟨e_1⟩,…,⟨e_n⟩ **[ROWS** ⟨*frame*⟩ **])**



Window Frame Specifications (Variant: RANGE)

window function one column frame specification

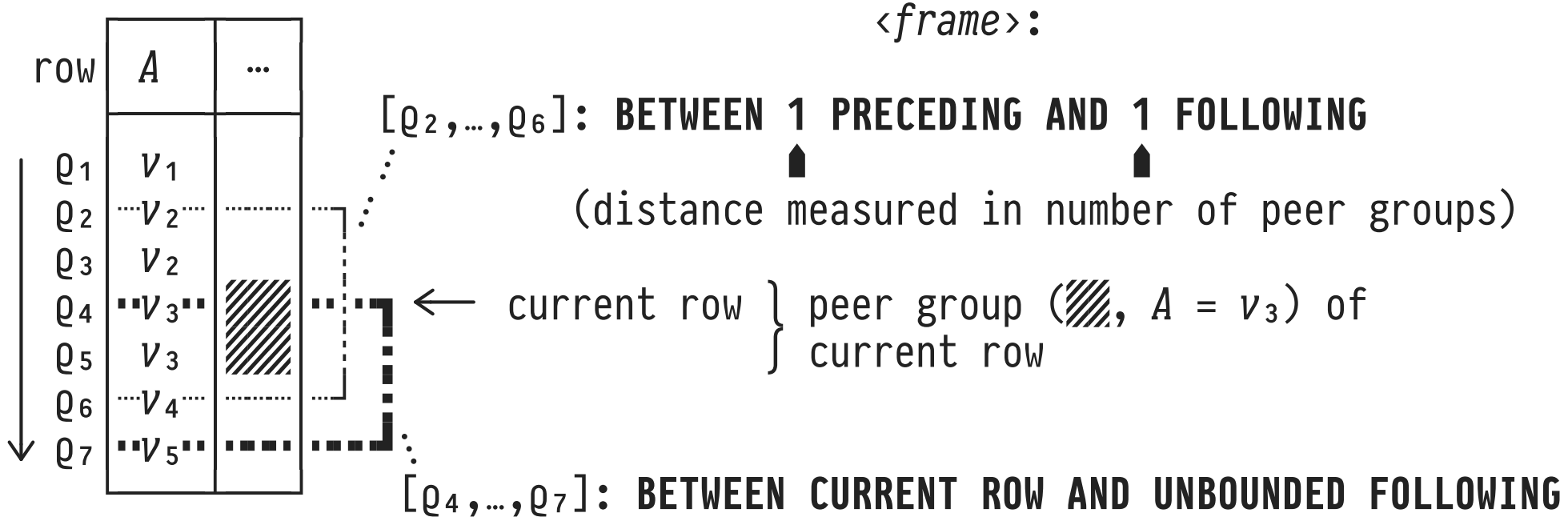
⟨*f*⟩ **OVER (ORDER BY** ⟨*A*⟩ [**RANGE** ⟨*frame*⟩])



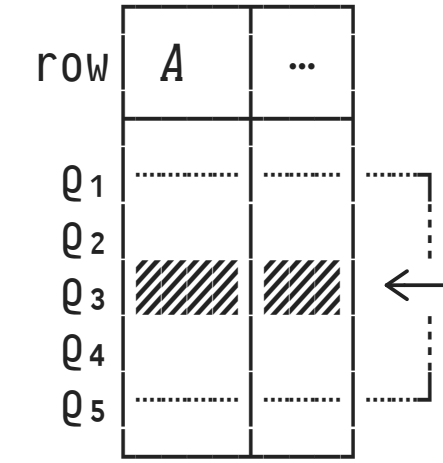
Window Frame Specifications (Variant: GROUPS)

window function ordering criteria frame specification

⟨*f*⟩ **OVER (ORDER BY** ⟨e_1⟩,…,⟨e_n⟩ **[GROUPS** ⟨*frame*⟩ **])**

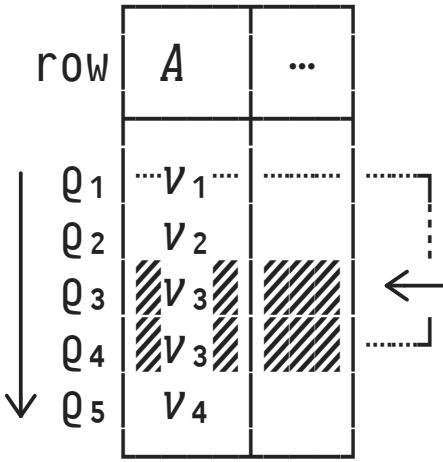


Window Frame Specifications: Abbreviations



$[\varrho_1,\ldots,\varrho_5]$: **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

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
ϱ_1	1	●
ϱ_2	2	○
ϱ_3	3	○
ϱ_4	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

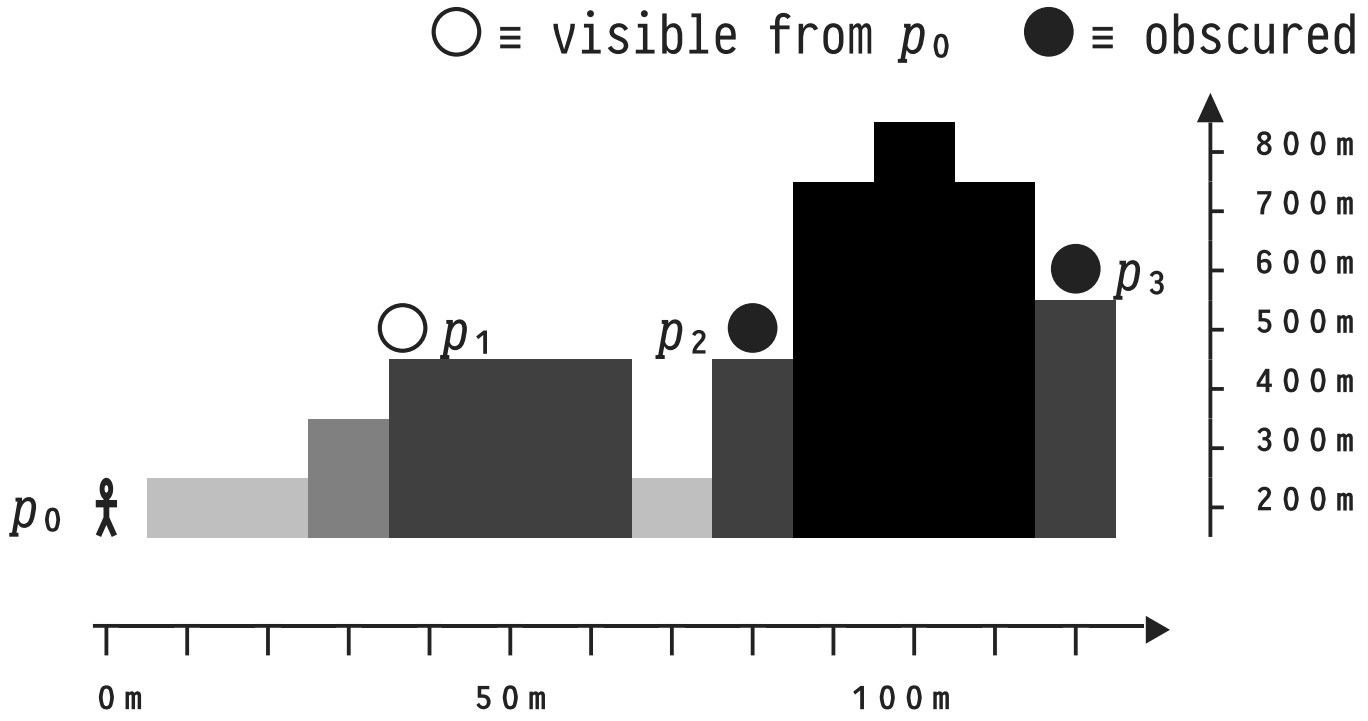
Optionally, we may **partition** the input table *before* rows are sorted and window frames are determined:

all input rows that agree on all ⟨p_i⟩ form one partition

```
⟨f⟩ OVER ([ PARTITION BY ⟨p₁⟩,…,⟨pₘ⟩ ]
          [ ORDER BY ⟨e₁⟩,…,⟨eₙ⟩ ]
          [ ⟨frame⟩ ])
```

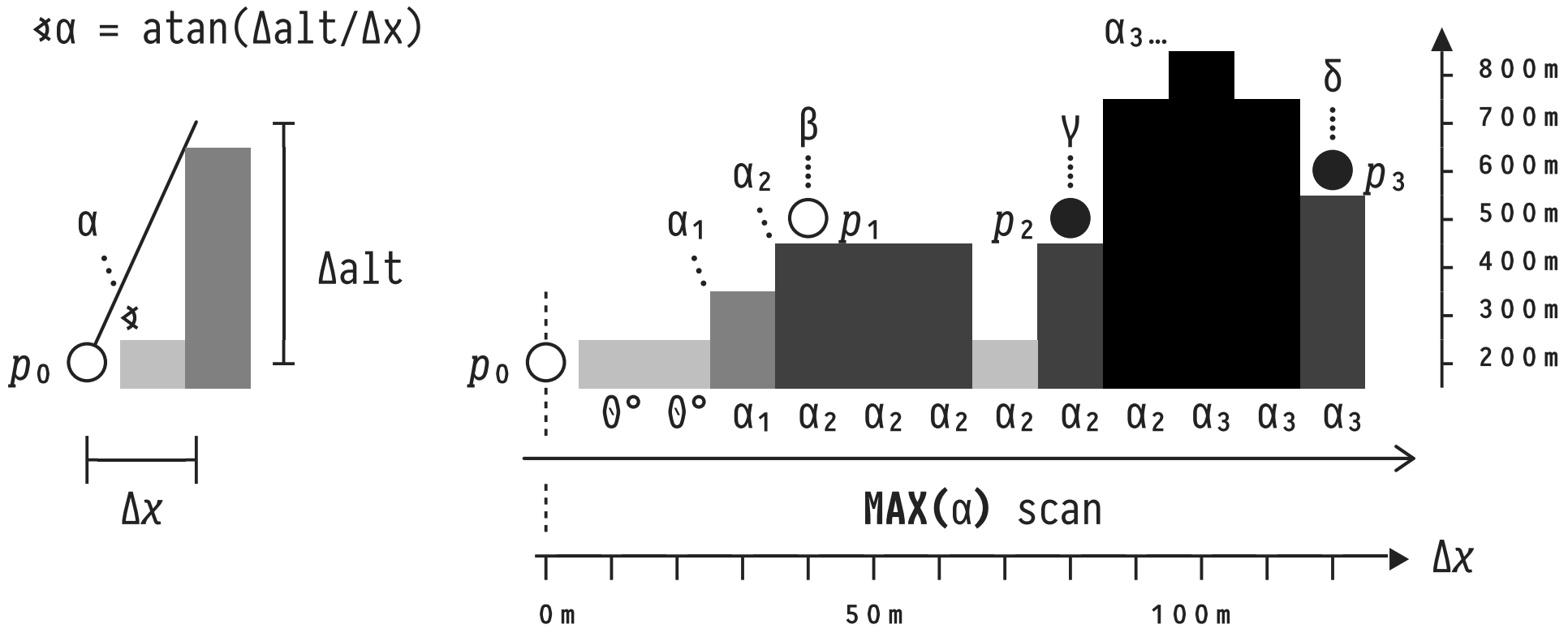
- Note:
  1. Frames **never cross partitions.**
  2. `BETWEEN … PRECEDING AND … FOLLOWING` **respects partition boundaries.**

Q: Which Spots are Visible in a Hilly Landscape?



- 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 $0° < \alpha_1 < \alpha_2 < \alpha_3$ and $\beta \geqslant \alpha_2$, $\gamma < \alpha_2$, $\delta < \alpha_3$.

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

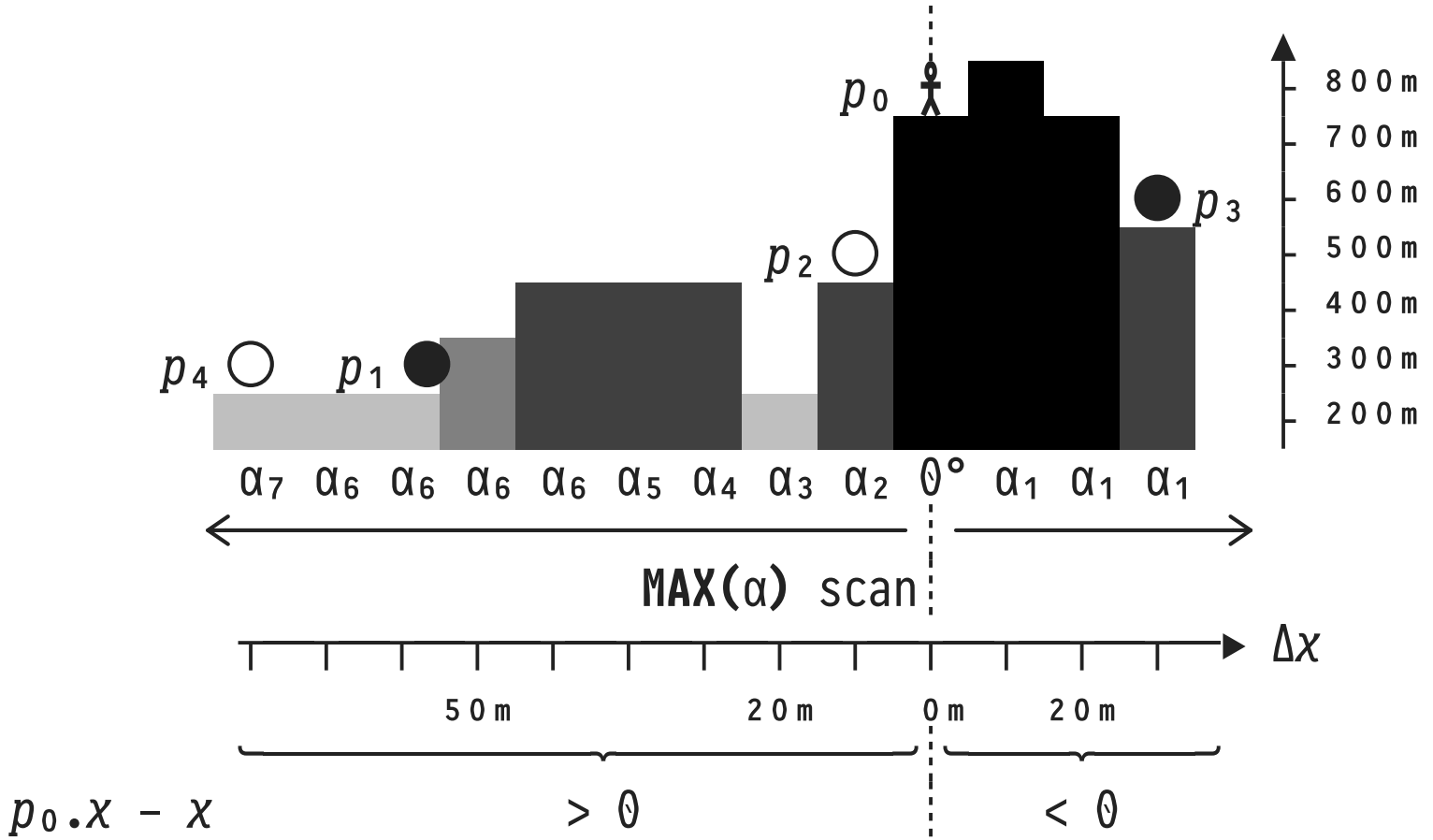
- **Output:** Can p_0 see the point on the hilltop at x?

x	visible?
0	true
10	true
⋮	⋮
120	false

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

```
WITH
-- ❶ Angles 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),
-- ❷ 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
⋮
-- 2 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
),
⋮
```

- ∀a ∈ angles: a.x ≠ p0.x ⟹ We end up with **two** partitions.

Scans are a general and expressive computational pattern:

```
 ‹agg›(‹e›) OVER (ORDER BY ‹e1›, …, ‹en›
 └──┬──┘                {ROWS,RANGE,GROUPS} BETWEEN
(ϕ,z,⊕)                        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 $[x_1, x_2, \ldots] = [z,\ z \oplus x_1,\ (z \oplus x_1) \oplus x_2,\ \ldots]$

- 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   unnest(string_to_array(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]
```