Advanced SQL

04 — Lists and Table-Generating Functions

Summer 2024

Torsten Grust
Universität Tübingen, Germany

1 ¦ Lists: Aliens(?) Inside Table Cells

SQL tables adhere to the **First Normal Form** (1NF): values v inside table cells are *atomic* w.r.t. the tabular data model:

…	A	…
…	v	…

Let us now discuss the **list** data type:

- v may hold an ordered list of elements $[x_1,\ldots,x_n]$.
- SQL treats v as an atomic unit, but ...
- ... list functions and operators also enable SQL to query the x_i individually (still, that's no ↯ with 1NF).

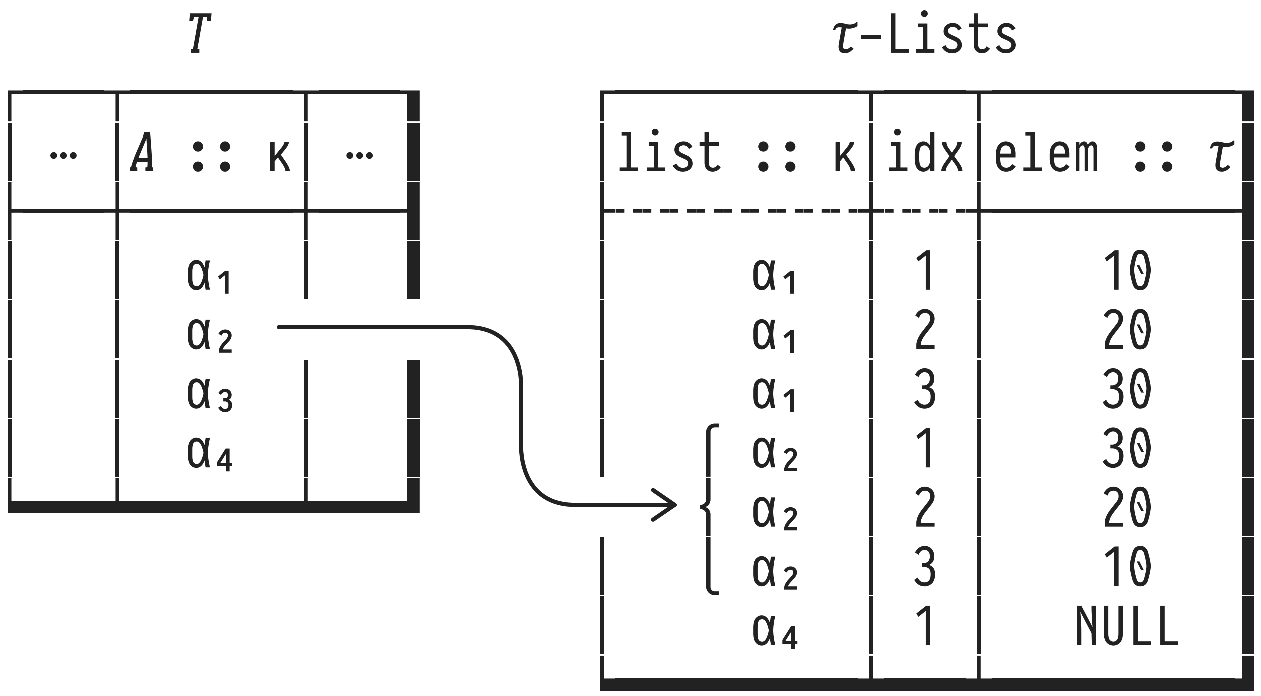
2 ┆ List Types

- For type τ, τ[] is the type of **homogeneous lists of elements of τ.**
  - τ may be built-in or user-defined (enums, row types).
  - List size is unspecified—the list is dynamic. (DuckDB also implements $\tau[n]$, the type of **arrays of fixed length** n.)

…	*A* :: int[]	…
…	[10,20,30]	…
…	[30,20,10]	…
…	[]	…
…	[NULL]	…

T

“Simulating” Lists (Tabular List Semantics)



T

…	$A :: \kappa$	…
	α_1	
	α_2	
	α_3	
	α_4	

τ-Lists

list :: κ	idx	elem :: τ
α_1	1	10
α_1	2	20
α_1	3	30
α_2	1	30
α_2	2	20
α_2	3	10
α_4	1	NULL

⚠ empty list α_3 absent

- κ denotes a suitable key data type.
- List indexes are of type int and 1-based.

3 ┆ List Literals

One-dimensional list literals of type τ[]:

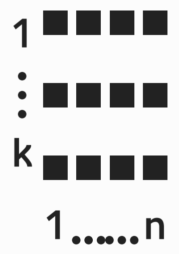
`[] ::` τ`[]`	empty list of elements of type τ
$[x_1, \ldots, x_n]$ $\text{list\_value}(x_1, \ldots, x_n)$	$\Big\}$ all x_i of type τ

Multi-dimensional list literals of type τ[][]:

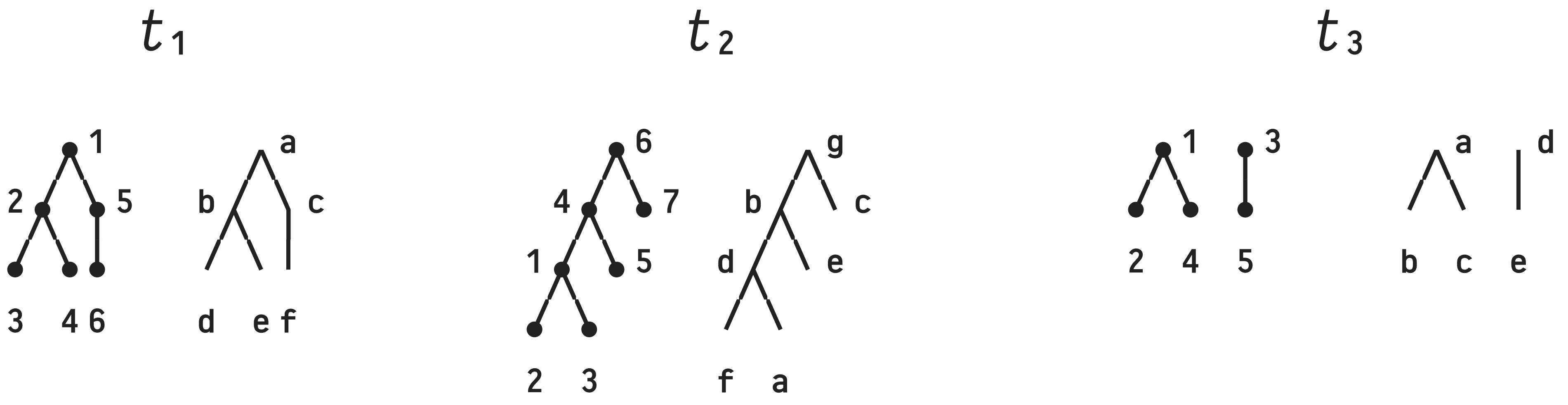
$[[x_{11}],[],[x_{31},x_{32}]]$ (ragged)

matrix: all sub-lists agree in size

$$[\underbrace{[x_{11},\ldots,x_{1n}]},\ldots,\underbrace{[x_{k1},\ldots,x_{kn}]}]$$



Example: Tree Encoding (parents[i] ≡ parent of node i)



Tree shape and node labels held in separate in-sync lists:

tree	parents	labels
t_1	[NULL,1,2,2,1,5]	['a','b','d','e','c','f']
t_2	[4,1,1,6,4,NULL,6]	['d','f','a','b','e','g','c']
t_3	[NULL,1,NULL,1,3]	['a','b','d','c','e']
	1 2 3 4 5	1 2 3 4 5 ← index i

Trees

Constructing Lists

- **Append**/**prepend** element ※ to list or
- **concatenate** lists:

```
list_append ([x₁,…,xₙ],※)              ≡   [x₁,…,xₙ,※]
list_prepend(※,[x₁,…,xₙ])              ≡   [※,x₁,…,xₙ]

list_concat([x₁,…,xₙ],    [y₁,…,yₘ])   ≡   [x₁,…,xₙ,y₁,…,yₘ]
            [x₁,…,xₙ] || [y₁,…,yₘ]     ≡   [x₁,…,xₙ,y₁,…,yₘ]
```

- Academics: "*List type τ[] forms a monoid (τ[], ||, []) with commutative operation || and neutral element [].*"

Accessing List Elements: Indexing / Slicing

- List **indexes** i are 1-based (let $xs \equiv [x_1,x_2,\ldots,x_n]$):

$$
\begin{aligned}
xs[i] &\equiv x_i && (1 \leqslant i \leqslant n)\\
(\mathrm{NULL})[i] &\equiv \mathrm{NULL} &&\\
xs[\mathrm{NULL}] &\equiv \mathrm{NULL} &&\\
xs[i{:}j] &\equiv [x_i,\ldots,x_j] && (i > j{:}\ [])\\
xs[i{:}\ ] &\equiv [x_i,\ldots,x_n] &&\\
xs[\ {:}j] &\equiv [x_1,\ldots,x_j] &&
\end{aligned}
$$

- Access the last element / from the list back:

$$
\begin{aligned}
xs[\mathrm{len}(xs)] &\equiv x_n &&\\
xs[-i] &\equiv x_{n-(i-1)} && (1 \leqslant i \leqslant n)
\end{aligned}
$$

Searching for Elements in Lists

Indexing accesses lists by position. **Searching** accesses list by **contents**, instead.

- Let $xs \equiv [x_1,\dots,x_{i-1},※,x_{i+1},\dots,x_{j-1},※,x_{j+1},\dots,x_n]$ and comparison operator $\theta \in \{=,<,>,<>,<=,>=\}$:

$x\ \theta$ ANY(xs)	≡	$\exists\, i \in \{1,\dots,n\}: x\ \theta\ xs$[i]
$x\ \theta$ ALL(xs)	≡	$\forall\, i \in \{1,\dots,n\}: x\ \theta\ xs$[i]
list_has(xs,x)	≡	x = ANY(xs)
list_has_any(xs,[$y_1,\dots,y_m$])	≡	$\exists\, i \in \{1,\dots,m\}: y_i$ = ANY(xs)
list_position(xs,※)	≡	i (if ※ not found: 0)

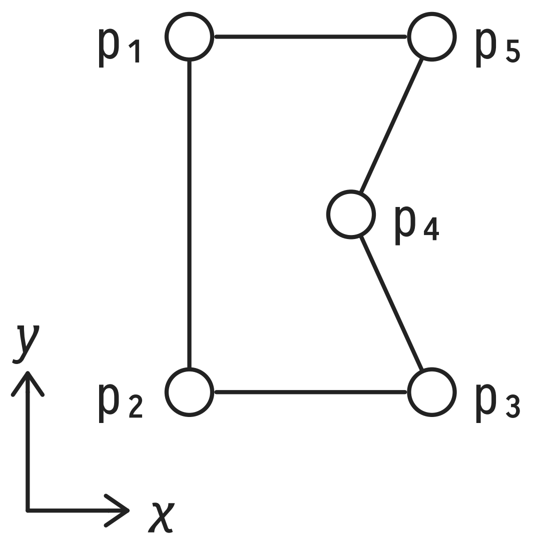
Advanced List Processing (think Haskell, APL)

```
-- map and fold(l1)
list_transform(xs, x -> e(x))    ≡ [e($x_1$),…,e($x_n$)]
list_reduce(xs, (a,x) -> a ⊗ x)) ≡ (⋯(($x_1$ ⊗ $x_2$) ⊗ $x_3$)⋯) ⊗ $x_n$
-- apply Boolean mask ($b_i$ :: boolean)
list_where(xs, [$b_1$,…,$b_n$])  ≡ [$x_i$ | i ∊ [1,…,n], $b_i$ = true]
-- filter and flatten
list_filter(xs, x -> p(x)) ≡ list_where(xs,
                               list_transform(xs, x -> p(x))
flatten(xss)               ≡ list_reduce(xss,
                               ($xs_1$,$xs_2$) -> $xs_1$ || $xs_2$)
```

Also: position-aware map (access list index *i* of element *x*):

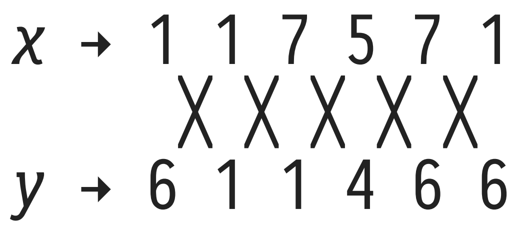
```
list_transform(xs, (x,i) -> e(x,i))  ≡ [e($x_1$,1),…,e($x_n$,n)]
```

Farewell, Tables? Use SQL as a List Programming Language?



$$\begin{array}{lcl} & & \;\;x\;\;y \\ & & \;\;\downarrow\;\;\downarrow \\ p_1 & = & (1,6) \\ p_2 & = & (1,1) \\ p_3 & = & (7,1) \\ p_4 & = & (5,4) \\ p_5 & = & (7,6) \end{array}$$

- Area of the 2D polygon $p_1 \cdots p_5$ ("shoe lace" formula):



$$\frac{1}{2} \times \left(\begin{array}{l} \quad p_1.x \times p_2.y - p_2.x \times p_1.y \\ + \; p_2.x \times p_3.y - p_3.x \times p_2.y \\ + \; \cdots \end{array} \right)$$

4 ¦ Bridging Lists and Tables: unnest & aggregate list

```
SELECT t.elem
FROM   unnest([x_1,…,x_n]) AS t(elem)
              └────┬────┘
                 ≡ xs
```

≡

Table t

elem
x_1
⋮
x_n

```
SELECT list(t.elem) AS xs
FROM   (VALUES (x_1),
                 ⋮
               (x_n)) AS t(elem)
```

≡

xs
$[x_1,…,x_n]$

- unnest(·): a *table-returning function*. More on that soon.
- ⚠️ Preservation of order of the x_i is *not* guaranteed…

Representing Order (Indices) As First-Class Values

```
SELECT t.*
FROM   unnest([$x_1$,…,$x_n$])
       WITH ORDINALITY AS t(elem,idx)
                                ↑
```

≡

elem	idx
x_1	1
⋮	⋮
x_n	n

```
       recall ordered aggregates
       ⏟
SELECT list(t.elem ORDER BY t.idx) AS $xs$
FROM   (VALUES ($x_1$,1),
                  ⋮
               ($x_n$,$n$)) AS t(elem,idx)
```

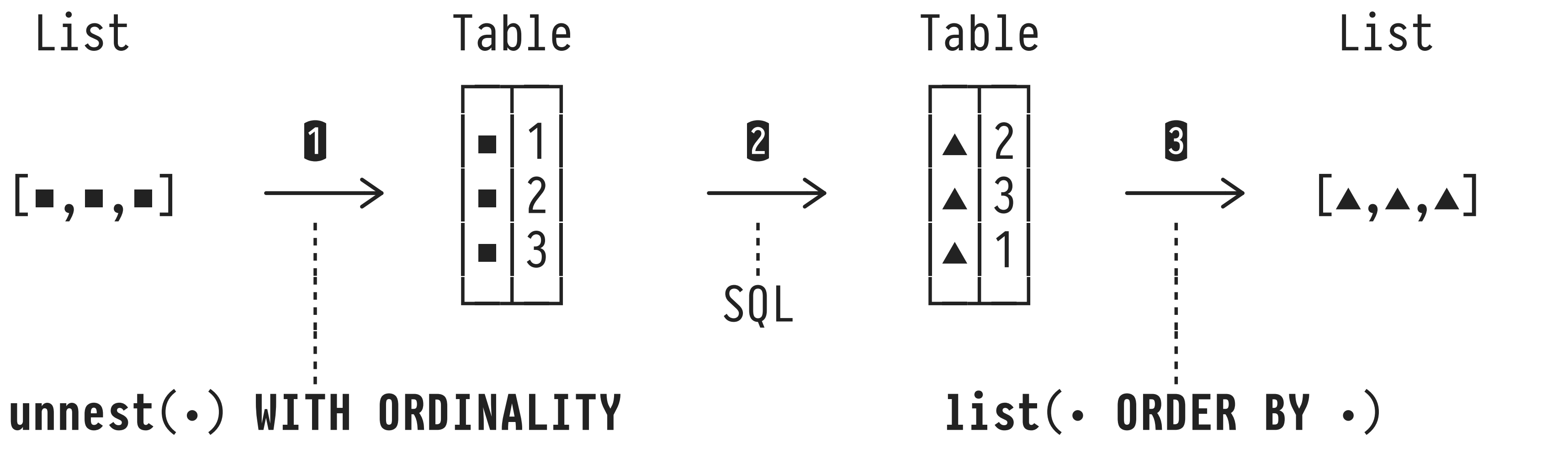
≡

xs
[x_1,…,x_n]

- f(…) WITH ORDINALITY adds a trailing column (see ↑) of ascending indices 1,2,… to the output of function f.

A Relational List Programming Pattern

Availability of unnest(•) and ordered aggregate list(•) suggests a pattern for **relational list programming:**



- At ❷ use the full force of SQL, read/transform/generate elements and their positions at will.
- ❶+❸ constitute **overhead:** an RDBMS is *not* a list PL.

Nested Structs + Lists vs. JSON

- SQL type constructors struct(…) and τ[] nest arbitrarily: may build complex tree-shaped structures much like JSON.

- DuckDB supports bidirectional casting between SQL and JSON values:
  - SQL structs ↔ JSON objects, SQL lists ↔ JSON arrays.
  - SQL → JSON ✓ ⚠️ Prerequisites for SQL ← JSON casting:
    - JSON object keys needs to be known *a priori*.
    - JSON arrays need to be homogeneous.

- SQL's unnest(•)/list(•) processing idiom applies.

5 ┆ DuckDB: Key/Value Maps (Type Constructor `map`)

- Recall the container types:
  - `struct`: map fixed fields to values of varied types.
  - `τ[]`: map dynamic `int` index set to values of type τ.

- Additional container type in DuckDB:
  - `map(τ₁,τ₂)`: map dynamic set of keys (of type τ_1) to values (of type τ_2).

- **Example:** two equivalent `map(int,boolean)` literals:

```
map {1:true, 3:true, 4:false}
              ≡
map([1,3,4], [true,true,false])
```

DuckDB: Accessing/Constructing Maps

Let $m \equiv \mathrm{map}([k_1,\ldots,k_n], [v_1,\ldots,v_n])$, $k_i \neq k_j$ for $i \neq j$.

```
$m[k_i]$            ≡ $[v_i]$  -- if $k_i \in [k_1,...,k_n]$
$m[k_i]$            ≡ []     -- otherwise (NB. $m[\cdot]$ cannot fail)

cardinality($m$) ≡ $n$
map_keys($m$)    ≡ $[k_1,\ldots,k_n]$
map_values($m$)  ≡ $[v_1,\ldots,v_n]$

map_entries($m$) ≡ $\{k_1:v_1,\ldots,k_n:v_n\}$
map_from_entries([{□:$k_1$,□:$v_1$},…,{□:$k_n$,□:$v_n$}]) ≡ $m$ -- any □

map_concat($m_1$,$m_2$)   -- merges maps $m_1$ and $m_2$
                      -- (keys in $m_2$ overwrite those in $m_1$)
```

6 ¦ Table-Generating Functions

What is the **type** of unnest(•)?

- unnest(•) establishes a bridge between lists and SQL's tabular data model:

$$\text{unnest} :: \tau[] \rightarrow \text{TABLEOF } \tau$$

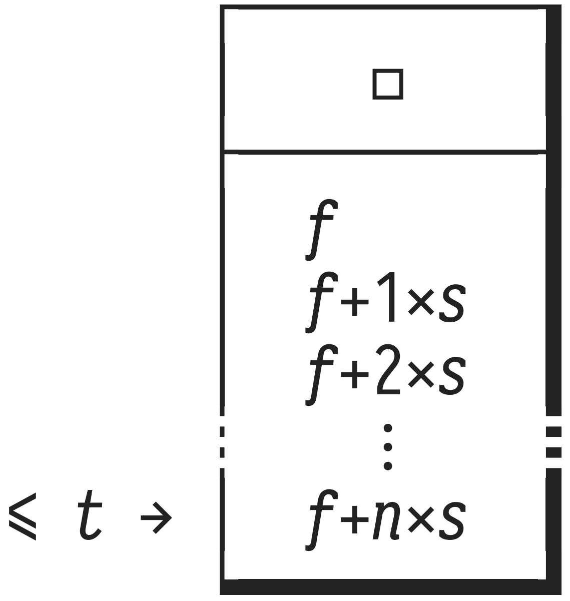
- In SQL, functions of type $\tau_1 \rightarrow \text{TABLEOF } \tau_2$ are known as **table-generating or set-returning functions.** May be invoked wherever a query expects a table (FROM clause): compositionality.

- Several of these functions are built into DuckDB.

Series Generators

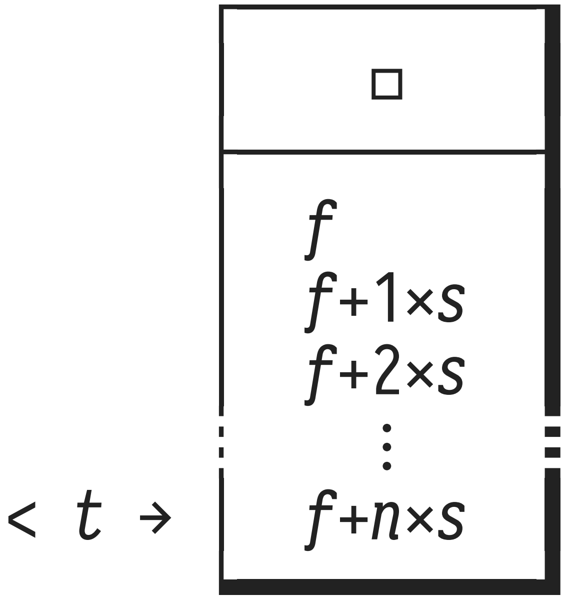
Built-in table-generating functions that generate **tables of consecutive numbers/values:**

generate_series(f,t,s)



s ≡ 1, if absent
f,t: numbers/timestamps

range(f,t,s)



s ≡ 1, if absent
stop value t excluded

7 ¦ ',' in the FROM Clause and Row Variable References

```
SELECT …
FROM   Q₁ AS t₁, Q₂ AS t₂, Q₃ AS t₃ -- t_{i<j} not free in Q_j
```

- Q: Why is $t_{i<j}$ *not* usable in Q_j?
- A: "*… the ',' in FROM is commutative and associative…*". Query optimization might rearrange the Q_j:

$$\begin{tikzcd}
Q_1 \times Q_2 \times Q_3 \arrow[d, bend left] \\
Q_1 \times Q_3 \times Q_2 \arrow[d, bend left] \\
(Q_3 \bowtie Q_1) \times Q_2
\end{tikzcd}$$

❶ original order as suggested by FROM clause

❷ swapped Q_2,Q_3 (Q_1,Q_3 now adjacent)

❸ join Q_3,Q_1 first (we expect small $|Q_3 \bowtie Q_1|$)

But Dependent Iteration in FROM is Useful...

Recall (find largest label in each tree t_1):

```
SELECT t₁.tree, MAX(t₂.label) AS "largest label"
--       Q₁                 Q₂
--     ⏞⏞⏞⏞⏞        ⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞⏞
FROM   Trees AS t₁, unnest(t₁.labels) AS t₂(label)
GROUP BY t₁.tree;          ↑
                           ↯
```

- **Dependent iteration** (here: Q_2 depends on t_1 defined in Q_1) has its uses and admits intuitve query formulation.

- **NB.** DuckDB's “friendly SQL” analyzes dependencies between FROM clause entries, introduces LATERAL automatically.

LATERAL:[1] Dependent Iteration for Everyone

Prefix Q_j with LATERAL in the FROM clause to announce dependent iteration:

```
SELECT …
FROM   Q₁ AS t₁, …, LATERAL Qⱼ AS tⱼ, …
                             ↑
                  may refer to t₁,…,tⱼ₋₁
```

- Works for *any* table-valued SQL expression Q_j, subqueries in (…) in particular.
  - Good style: be explicit and use LATERAL even on DuckDB.

[1] Lateral /ˈlæt(ə)rəl/ a. [Latin lateralis]: *sideways*

LATERAL: SQL's for each-Loop

LATERAL admits the formulation of **nested-loops** computation:

```
SELECT e
FROM   Q₁ AS t₁, LATERAL Q₂ AS t₂, LATERAL Q₃ AS t₃
```

is evaluated just like this nested loop:

```
for $t_1$ in $Q_1$
  for $t_2$ in $Q_2(t_1)$
    for $t_3$ in $Q_3(t_1,t_2)$
      return $e(t_1,t_2,t_3)$
```

- Convenient, intuitive, and perfectly OK.
  But much like hand-cuffs for the query optimizer. ⚠

LATERAL Example: Find the Top *n* Rows Among a Peer Group

Which are **the three tallest** two- and four-legged dinosaurs?

```
SELECT locomotion.legs, tallest.species, tallest.height
FROM   (VALUES (2), (4)) AS locomotion(legs),
       LATERAL (SELECT d.*
                FROM   dinosaurs AS d
                WHERE  d.legs = locomotion.legs ←
                ORDER BY d.height DESC
                LIMIT 3) AS tallest
```

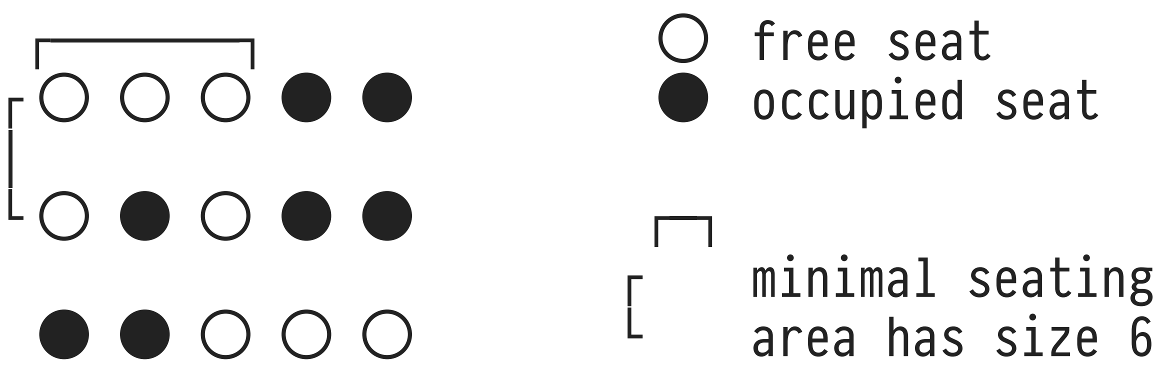
legs	species	height
2	Tyrannosaurus	7
2	Ceratosaurus	4
2	Spinosaurus	2.4
4	Supersaurus	10
4	Brachiosaurus	7.6
4	Diplodocus	3.6

8 ⁞ ACM ICPC: Finding Seats

ACM ICPC Task **Finding Seats** (South American Regionals, 2007)

"K friends go to the movies but they are late for tickets. To sit close to each other, they look for K free seats such that the rectangle containing these seats has minimal area."

- Assume $K = 5$:



Finding Seats: Parse the ICPC Input Format

- Typical ICPC character-based input format:

```
...XX␍
.X.XX␍
XX...
```

.	free seat
X	occupied seat
␍	new line

- **Parse into table** making seat position/status explicit:

row	col	taken?
1	1	false
1	2	false
1	3	false
1	4	true
⋮	⋮	⋮
3	5	false

Table seats

Finding Seats: Parse the ICPC Input Format (Table seats)

```
-- Assume cinema ≡ E'...XX\n.X.XX\nXX...'

SELECT  row.pos, col.pos, col.x = 'X' AS "taken?"
FROM    -- rows
        unnest(string_split(cinema, E'\n'))
        WITH ORDINALITY AS row(xs, pos),
        -- columns
LATERAL unnest(string_split(row.xs, ''))
        WITH ORDINALITY AS col(x, pos)
```

- `string_split(cinema, E'\n')` yields a list of three row strings: `'...XX'`, `'.X.XX'`, `'XX...'`.
- `string_split(row.xs, '')` splits string `row.xs` into a list of individual characters (= seats).

Finding Seats: A Problem Solution (Generate and Test)

- **Query Plan:**

  1. Parse the input into a seating plan table seats.
  2. **Generate all** possible north-west (*nw*) and south-east (*se*) corners of rectangular seating areas:
     - For each such ⌜*nw*,*se*⌟ rectangle, scan its seats and **test** whether the number of free seats is ⩾ *K*.
     - If so, record *nw* together with the rectangle's width/height.
  3. Among these rectangles with sufficient seating space, select one with minimal area.

Finding Seats: Generating All Possible Rectangles

Generate all ⌜*nw*,*se*⌟ corners for rectangles in the seating plan (table seats):

```
SELECT nw, se
FROM   seats AS nw,          -- self-join
       seats AS se
WHERE  nw.col <= se.col      -- ⌜nw is to the top left of se⌟
AND    nw.row <= se.row
```

- This generates $\left(\sum_{r=1}^{rows} r\right) \times \left(\sum_{c=1}^{cols} c\right)$ rectangles.
- Generally: If possible, test/filter early.