

Advanced SQL

④

Lists and Table-Generating Functions

Winter 2025/26

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:

- Value v may be an ordered list of elements $[x_1, \dots, x_n]$.
- 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 \hookrightarrow with 1NF).

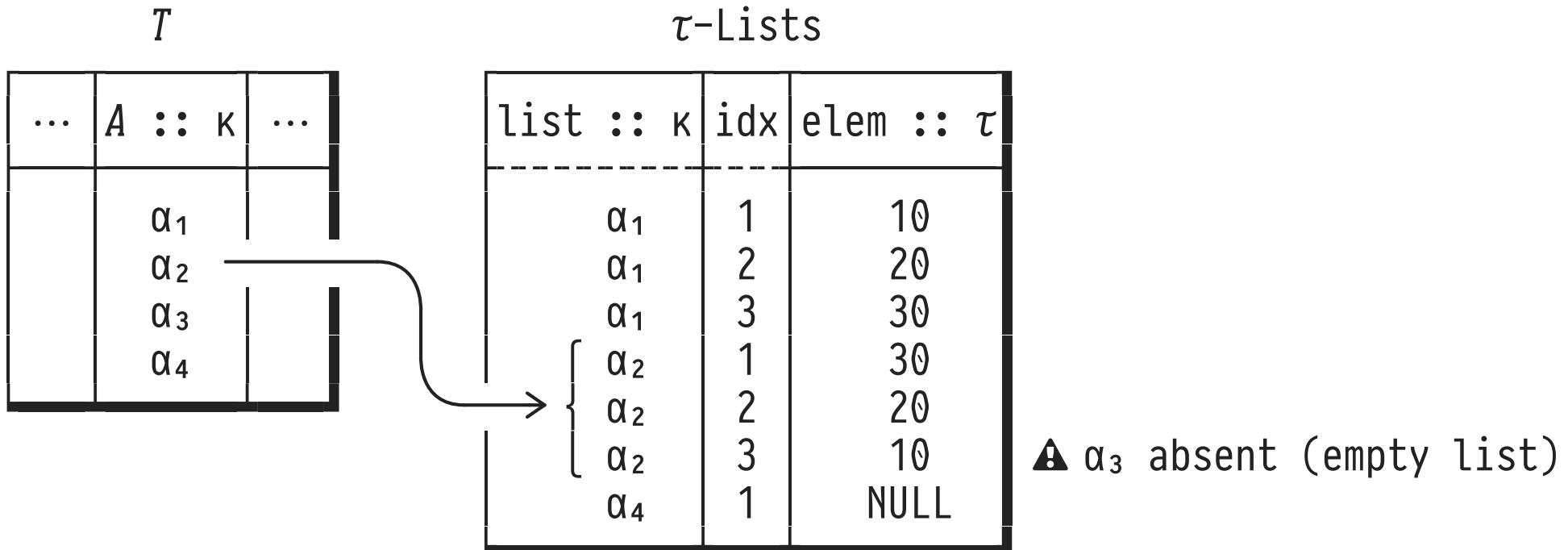
2 | List Types

- For type τ , $\tau[]$ 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** .)

...	$A :: \text{int}[]$...
...	[10, 20, 30]	...
...	[30, 20, 10]	...
...	[]	...
...	[NULL]	...

T

“Simulating” Lists (Tabular List Semantics)



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

3 | List Literals

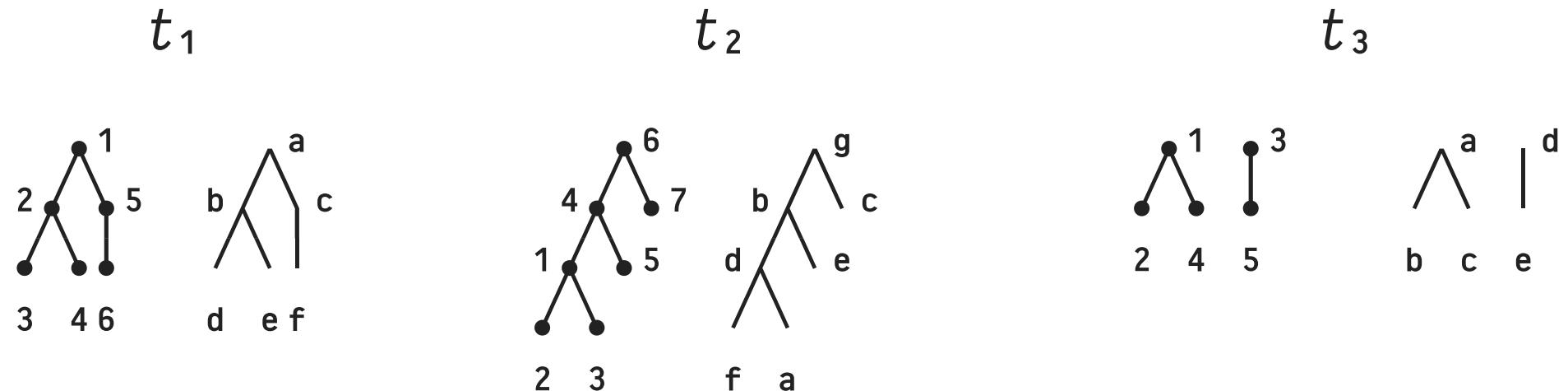
One-dimensional list literals of type $\tau[]$:

$[] :: \tau[]$	empty list of elements of type τ
$[x_1, \dots, x_n]$ list_value(x_1, \dots, x_n)	} all x_i of one type τ (or castable to τ)

Multi-dimensional list literals of type $\tau[][]$:

$[[x_{11}], [], [x_{31}, x_{32}]]$	(ragged)
matrix: all sub-lists agree in size	
$\overbrace{[[x_{11}, \dots, x_{1n}]]}^1, \dots, \overbrace{[x_{k1}, \dots, x_{kn}]}^k$	$1 \blacksquare \blacksquare \blacksquare \blacksquare$ $\vdots \blacksquare \blacksquare \blacksquare \blacksquare$ $k \blacksquare \blacksquare \blacksquare \blacksquare$ $1 \dots n$

Example: Tree Encoding ($\text{parents}[i] \equiv \text{parent of node } i$)



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

<u>tree</u>	<u>parents</u>	<u>labels</u>
t_1	<code>[NULL,1,2,2,1,5]</code>	<code>['a','b','d','e','c','f']</code>
t_2	<code>[4,1,1,6,4,NULL,6]</code>	<code>['d','f','a','b','e','g','c']</code>
t_3	<code>[NULL,1,NULL,1,3]</code> 1 2 3 4 5	<code>['a','b','d','c','e']</code> 1 2 3 4 5 \leftarrow index i

Trees

Constructing Lists

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

list_append $([x_1, \dots, x_n], \ast)$ $\equiv [x_1, \dots, x_n, \ast]$

listprepend $(\ast, [x_1, \dots, x_n])$ $\equiv [\ast, x_1, \dots, x_n]$

list_concat $([x_1, \dots, x_n], [y_1, \dots, y_m])$ $\equiv [x_1, \dots, x_n, y_1, \dots, y_m]$

$[x_1, \dots, x_n] \parallel [y_1, \dots, y_m]$ $\equiv [x_1, \dots, x_n, y_1, \dots, y_m]$

- Academics: “*List type $\tau[]$ forms a monoid $(\tau[], \parallel, [])$ with commutative operation \parallel and neutral element $[]$.*”

Accessing List Elements: Indexing / Slicing

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

$xs[i]$	$\equiv x_i$	$(1 \leq i \leq n)$
$(\text{NULL})[i]$	$\equiv \text{NULL}$	
$xs[\text{NULL}]$	$\equiv \text{NULL}$	
$xs[i:j]$	$\equiv [x_i, \dots, x_j]$	$(i > j: [])$
$xs[i:]$	$\equiv [x_i, \dots, x_n]$	
$xs[:j]$	$\equiv [x_1, \dots, x_j]$	

- Access the last element / from the list back:

$xs[\text{len}(xs)]$	$\equiv x_n$	
$xs[-i]$	$\equiv x_{n-(i-1)}$	$(1 \leq i \leq n)$

Searching for Elements in Lists

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

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

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

Advanced List Processing (think Haskell, APL)

```

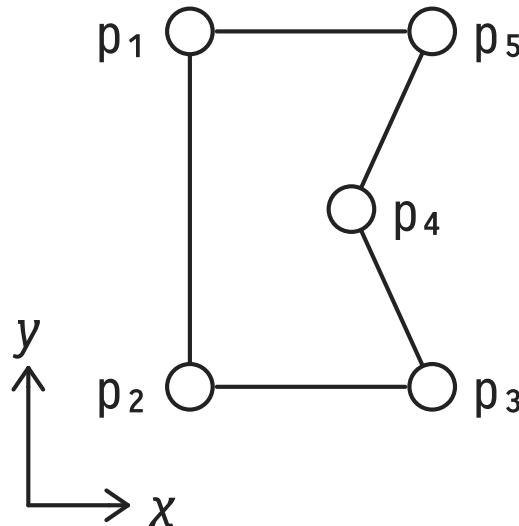
-- map and fold(l1)
list_transform(xs,  $x \rightarrow e(x)$ )  $\equiv [e(x_1), \dots, e(x_n)]$ 
list_reduce(xs,  $(a, x) \rightarrow a \otimes x$ )  $\equiv (\dots((x_1 \otimes x_2) \otimes x_3) \dots) \otimes x_n$ 
-- apply Boolean mask ( $b_i :: \text{boolean}$ )
list_where(xs,  $[b_1, \dots, b_n]$ )  $\equiv [x_i \mid i \in [1, \dots, n], b_i = \text{true}]$ 
-- filter and flatten
list_filter(xs,  $x \rightarrow p(x)$ )  $\equiv \text{list\_where}(xs,$ 
                            $\text{list\_transform}(xs, x \rightarrow p(x))$ 
flatten(xss)  $\equiv \text{list\_reduce}(xss,$ 
                            $(xs_1, xs_2) \rightarrow xs_1 \mid\mid xs_2)$ 

```

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

```
list_transform(xs,  $(x, i) \rightarrow e(x, i)$ )  $\equiv [e(x_1, 1), \dots, e(x_n, n)]$ 
```

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



x	y
↓	↓
$p_1 = (1, 6)$	
$p_2 = (1, 1)$	
$p_3 = (7, 1)$	
$p_4 = (5, 4)$	
$p_5 = (7, 6)$	

- Area of the 2D polygon $p_1 \dots p_5$ (“shoe lace” formula):

$$\begin{array}{r}
 x \rightarrow 1 \ 1 \ 7 \ 5 \ 7 \ 1 \\
 y \rightarrow 6 \ 1 \ 1 \ 4 \ 6 \ 6
 \end{array}
 \quad \mid \quad
 \frac{1}{2} \times \left(\begin{array}{l}
 p_1.x \times p_2.y - p_2.x \times p_1.y \\
 + p_2.x \times p_3.y - p_3.x \times p_2.y \\
 + \dots
 \end{array} \right)$$

4 | Bridging Lists and Tables: `unnest` & aggregate list

```
SELECT t.elem
FROM   unnest([ $x_1, \dots, x_n$ ]) AS t(elem)
       $\underbrace{\phantom{[x_1, \dots, x_n]}}$ 
       $\equiv xs$ 
```

Table t

elem
x_1
\vdots
x_n

```
SELECT list(t.elem) AS xs
FROM   (VALUES ( $x_1$ ),
       $\vdots$ 
      ( $x_n$ )) AS t(elem)
```

xs
$[x_1, \dots, x_n]$

- `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([ $x_1, \dots, x_n$ ])  

WITH ORDINALITY AS t(elem, idx)
```

↑

recall ordered aggregates

≡

elem	idx
x_1	1
\vdots	\vdots
x_n	n

```
SELECT list(t.elem ORDER BY t.idx) AS xs  

FROM (VALUES ( $x_1, 1$ ),  

         $\vdots$   

        ( $x_n, n$ )) AS t(elem, idx)
```

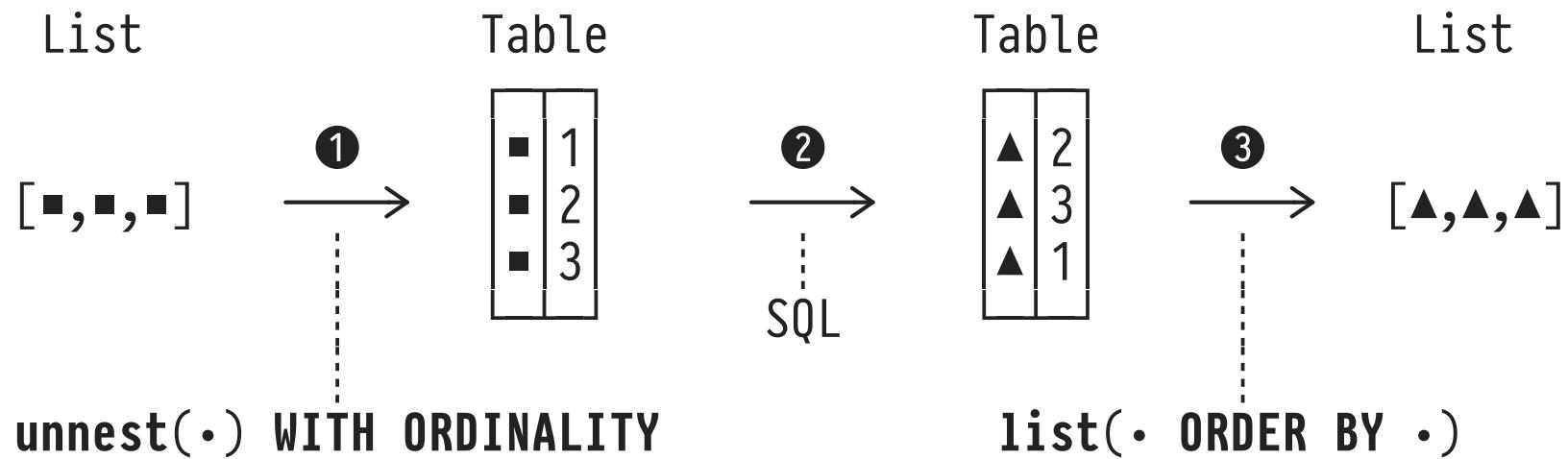
≡

xs
$[x_1, \dots, x_n]$

- $f(\dots)$ **WITH ORDINALITY** adds a trailing column (see ↑) of ascending indices $1, 2, \dots$ 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 **②** use the full force of SQL, read/transform/generate elements and their positions at will.
- **①+③** 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 \leftrightarrow JSON objects, SQL lists \leftrightarrow JSON arrays.
 - SQL \rightarrow JSON ✓  Prerequisites for SQL \leftarrow JSON casting:
 - JSON object keys needs to be *statically* known.
 - 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.
 - `$\tau[]$` : map dynamic `int` index set to values of type τ .
- Additional container type in DuckDB:
 - `map(τ_1, τ_2)`: 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 $m \equiv \text{map}([k_1, \dots, k_n], [v_1, \dots, v_n])$, $k_i \neq k_j$ for $i \neq j$.

$m[k_i]$	$\equiv v_i$	-- if $k_i \in [k_1, \dots, k_n]$
$m[k_i]$	$\equiv \text{NULL}$	-- otherwise
$\text{map_extract}(m, k_i)$	$\equiv [v_i]$	-- if $k_i \in [k_1, \dots, k_n]$
$\text{map_extract}(m, k_i)$	$\equiv []$	-- otherwise

$\text{cardinality}(m)$	$\equiv n$
$\text{map_keys}(m)$	$\equiv [k_1, \dots, k_n]$
$\text{map_values}(m)$	$\equiv [v_1, \dots, v_n]$

$\text{map_entries}(m)$	$\equiv \{k_1:v_1, \dots, k_n:v_n\}$
$\text{map_from_entries}([\{\square:k_1, \square:v_1\}, \dots, \{\square:k_n, \square:v_n\}])$	$\equiv m$ -- any \square

$\text{map_concat}(m_1, m_2)$	-- 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:

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

- In SQL, functions of type $\tau_1 \rightarrow \text{TABLEOF } \tau_2$ ¹ are known as **table-generating or set-returning functions**. These may be invoked wherever a query expects a table (`FROM` clause): compositionality.
- Table-generating functions are prevalent in SQL.

¹ Note: `TABLEOF τ` denotes the type of tables whose single column holds values of type τ . (`TABLEOF` is not available in DuckDB.)

Series Generators

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

generate_series(f, t, s)

$\leq t \rightarrow$	<table border="1"> <tr><td>□</td></tr> <tr><td>f</td></tr> <tr><td>$f+1 \times s$</td></tr> <tr><td>$f+2 \times s$</td></tr> <tr><td>\vdots</td></tr> <tr><td>$f+n \times s$</td></tr> </table>	□	f	$f+1 \times s$	$f+2 \times s$	\vdots	$f+n \times s$
□							
f							
$f+1 \times s$							
$f+2 \times s$							
\vdots							
$f+n \times s$							

step $s \equiv 1$, if absent
 f, t : numbers/timestamps

range(f, t, s)

$< t \rightarrow$	<table border="1"> <tr><td>□</td></tr> <tr><td>f</td></tr> <tr><td>$f+1 \times s$</td></tr> <tr><td>$f+2 \times s$</td></tr> <tr><td>\vdots</td></tr> <tr><td>$f+n \times s$</td></tr> </table>	□	f	$f+1 \times s$	$f+2 \times s$	\vdots	$f+n \times s$
□							
f							
$f+1 \times s$							
$f+2 \times s$							
\vdots							
$f+n \times s$							

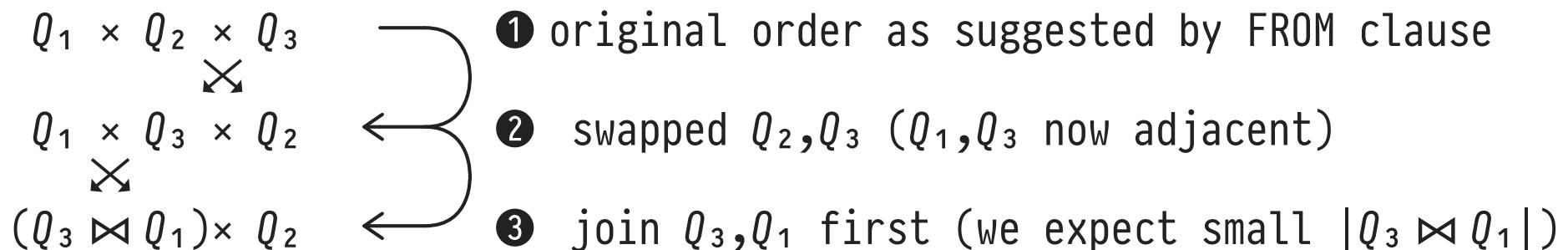
step $s \equiv 1$, if absent
stop value t excluded

7 | ',' in the **FROM** Clause and Row Variable References

SELECT ...

FROM Q_1 AS t_1 , Q_2 AS t_2 , Q_3 AS t_3 -- $t_{i < j}$ 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 :



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 intuitive query formulation.
-  DuckDB's “friendly SQL” analyzes dependencies between `FROM` clause entries, introduces `LATERAL` automatically.

LATERAL:² 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.

² Lateral /'læt(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 t1 in Q1
  for t2 in Q2(t1)
    for t3 in Q3(t1,t2)
      return e(t1,t2,t3)
```

- 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
```

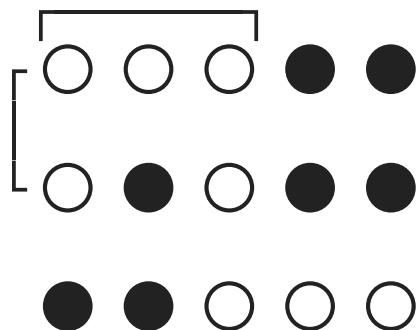
legs	species	height
2	Tyrannosaurus	7
2	Ceratosaurus	4
2	Spinosaurus	2.4
4	Supersaurus	10
4	Brachiosaurus	7.6
4	Diplodocus	3.6

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 $K = 5$:



○ free seat
● occupied seat

□ minimal seating
area has size 6

🔧 Finding Seats: Parse the ICPC Input Format

- Typical ICPC character-based input format:

...XX _R	.	free seat
.X.XX _R	X	occupied seat
XX...	_R	new line

- Parse into table making seat position/status explicit:

<u>row</u>	<u>col</u>	<u>taken?</u>
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 Possible 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 $\geq K$.
 - 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 $\left(\sum_{r=1}^{\text{rows}} r \right) \times \left(\sum_{c=1}^{\text{cols}} c \right)$ rectangles.
- Generally: If possible, test/filter early.