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 $[x_1,\ldots,x_n]$.
- 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 ϟ 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 :: κ	…
	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

⚠ α_3 absent (empty list)

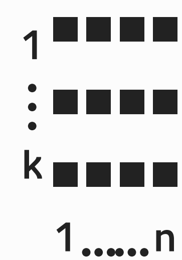
- κ denotes a suitable key data type.
- List indexes are of type int and 1-based.

3 ┆ List Literals

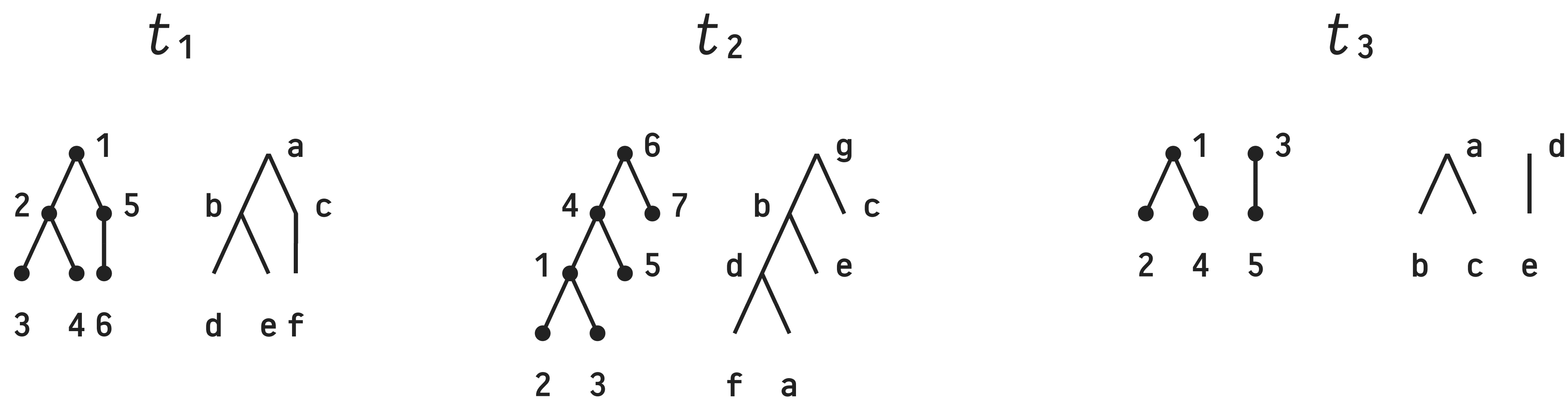
One-dimensional list literals of type τ[]:

[] :: τ[]	empty list of elements of type τ
[x_1,…,x_n]	all x_i of one type τ
list_value(x_1,…,x_n)	(or castable to τ)

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

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

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:

$$
\begin{array}{llll}
\textbf{list\_append}\;([x_1,\ldots,x_n],※) & & \equiv & [x_1,\ldots,x_n,※] \\
\textbf{list\_prepend}(※,[x_1,\ldots,x_n]) & & \equiv & [※,x_1,\ldots,x_n] \\
 & & & \\
\textbf{list\_concat}([x_1,\ldots,x_n], & [y_1,\ldots,y_m]) & \equiv & [x_1,\ldots,x_n,y_1,\ldots,y_m] \\
\qquad\qquad\; [x_1,\ldots,x_n] \;||\; & [y_1,\ldots,y_m] & \equiv & [x_1,\ldots,x_n,y_1,\ldots,y_m]
\end{array}
$$

- 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 $xs \equiv [x_1, x_2, \ldots, x_n]$):

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

- Access the last element / from the list back:

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

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

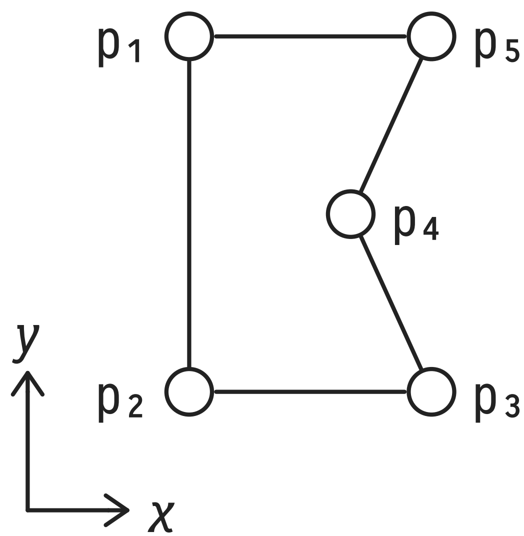
Advanced List Processing (think Haskell, APL)

```
-- map and fold(l1)
list_transform(xs, x -> e(x))    ≡ [e(x₁),…,e(xₙ)]
list_reduce(xs, (a,x) -> a ⊗ x)) ≡ (⋯((x₁ ⊗ x₂) ⊗ x₃)⋯) ⊗ xₙ
-- apply Boolean mask (bᵢ :: boolean)
list_where(xs, [b₁,…,bₙ])  ≡ [xᵢ | i ∈ [1,…,n], bᵢ = true]
-- filter and flatten
list_filter(xs, x -> p(x)) ≡ list_where(xs,
                               list_transform(xs, x -> p(x))
flatten(xss)               ≡ list_reduce(xss,
                               (xs₁,xs₂) -> xs₁ || xs₂)
```

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

```
list_transform(xs, (x,i) -> e(x,i))  ≡ [e(x₁,1),…,e(xₙ,n)]
```

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



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

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

$$
\begin{array}{l}
x \rightarrow 1\ 1\ 7\ 5\ 7\ 1 \\
\times\ \times\ \times\ \times\ \times \\
y \rightarrow 6\ 1\ 1\ 4\ 6\ 6
\end{array}
$$

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

4 ¦ Bridging Lists and Tables: unnest & aggregate list

```
SELECT t.elem
FROM   unnest([x₁,…,xₙ]) AS t(elem)
```

$$\underbrace{[x_1,\dots,x_n]}_{\equiv\ xs}$$

≡ Table t

elem
x_1
⋮
x_n

```
SELECT list(t.elem) AS xs
FROM   (VALUES (x₁),
                 ⋮
               (xₙ)) 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([x1,…,xn])
       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 (x1,1),
                 ⋮
               (xn,n)) AS t(elem,idx)
```

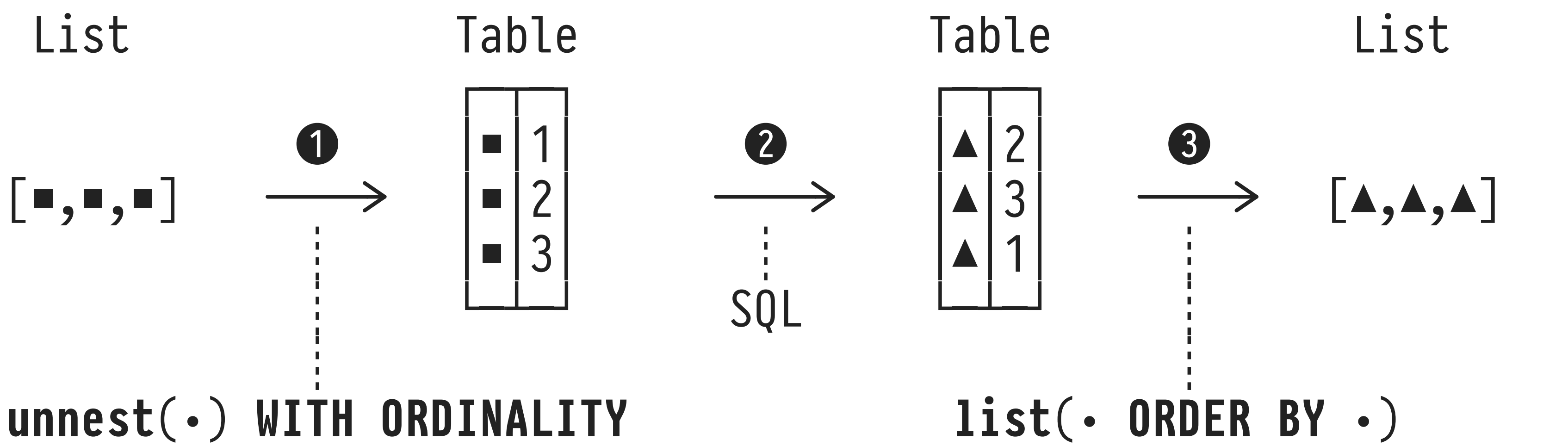
≡

xs
$[x_1,…,x_n]$

- $f(\cdots)$ 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 *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.
  - τ`[]`: 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 $m \equiv \mathtt{map}([k_1,\ldots,k_n],\ [v_1,\ldots,v_n])$, $k_i \neq k_j$ for $i \neq j$.

$m[k_i] \equiv v_i$ -- if $k_i \in [k_1,\ldots,k_n]$
$m[k_i] \equiv$ NULL -- otherwise
map_extract(m,k_i) $\equiv [v_i]$ -- if $k_i \in [k_1,\ldots,k_n]$
map_extract(m,k_i) $\equiv$ [] -- otherwise

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

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

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 $\tau_1 \rightarrow \text{TABLEOF } \tau_2$[1] are known as **table-generating or set-returning functions.** These may be invoked wherever a query expects a table (FROM clause): compositionality.

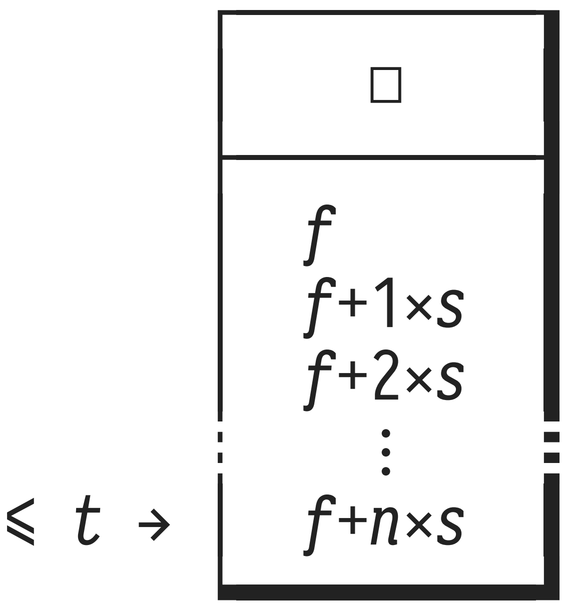
- Table-generating functions are prevalent in SQL.

[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

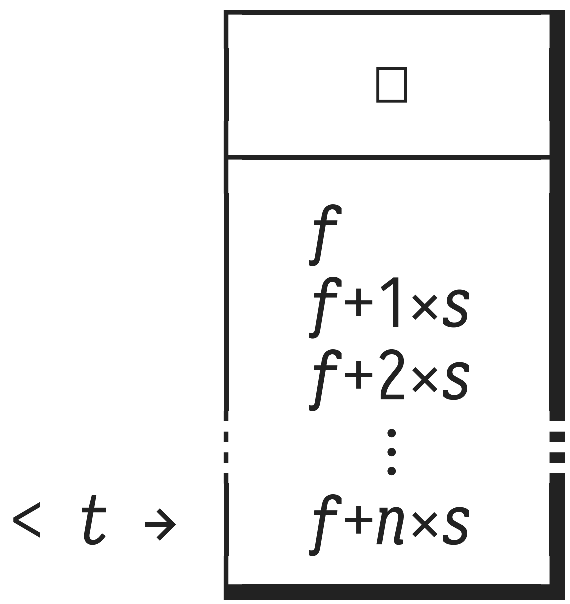
Built-in table-generating functions that generate **tables of consecutive numbers/values:**

generate_series(f,t,s)



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

range(f,t,s)



step $s \equiv 1$, 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:

$Q_1 \times Q_2 \times Q_3$ ❶ original order as suggested by FROM clause

$Q_1 \times Q_3 \times Q_2$ ❷ swapped Q_2, Q_3 (Q_1, Q_3 now adjacent)

$(Q_3 \bowtie Q_1) \times Q_2$ 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.
- 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 /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 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
```

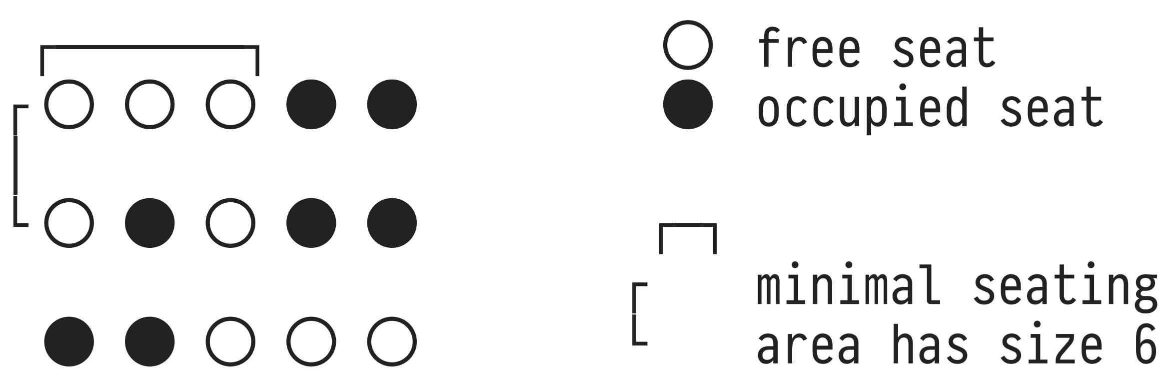
legs	species	height
2	Tyrannosaurus	7
2	Ceratosaurus	4
2	Spinosaurus	2.4
4	Supersaurus	10
4	Brachiosaurus	7.6
4	Diplodocus	3.6

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 $K = 5$:



Finding Seats: Parse the ICPC Input Format

- Typical ICPC character-based input format:

...XX␍		.	free seat
.X.XX␍		X	occupied seat
XX...		␍	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 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 ⩾ *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.