Advanced SQL

04 — Arrays and User-Defined Functions

Summer 2022

Torsten Grust
Universität Tübingen, Germany

1 ¦ Arrays: 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 **array** data type:

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

[1] To the PostgreSQL developer who decided to use $\{\cdots\}$ to denote *arrays*: **No dessert for you today!**

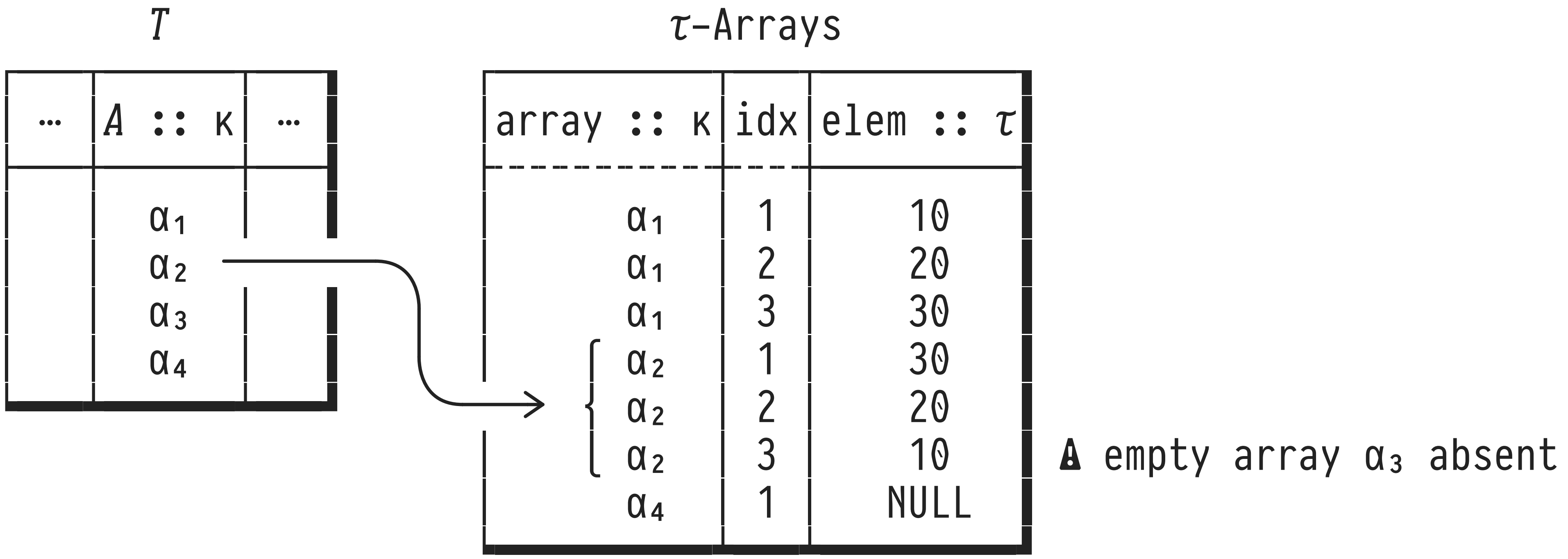
2 ┆ Array Types

- For type τ, τ[] (or τ array) is the type of **homogeneous arrays of elements of τ.**
  - τ may be built-in or user-defined (enums, row types).
  - Array size is unspecified—the array is dynamic. (PostgreSQL accepts τ[n] but the n is ignored.)

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

T

"Simulating" Arrays (Tabular Array Semantics)



T

…	A :: κ	…
	α_1	
	α_2	
	α_3	
	α_4	

τ-Arrays

array :: 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 array α_3 absent

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

One-dimensional array literals of type τ[]:

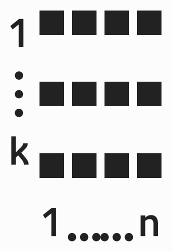
array[] :: τ[]	empty array of elements of type τ
array[x_1,…,x_n] '{x_1,…,x_n}' :: τ[]	$\Big\}$ all x_i of type τ

Multi-dimensional rectangular array literals of type τ[][]:

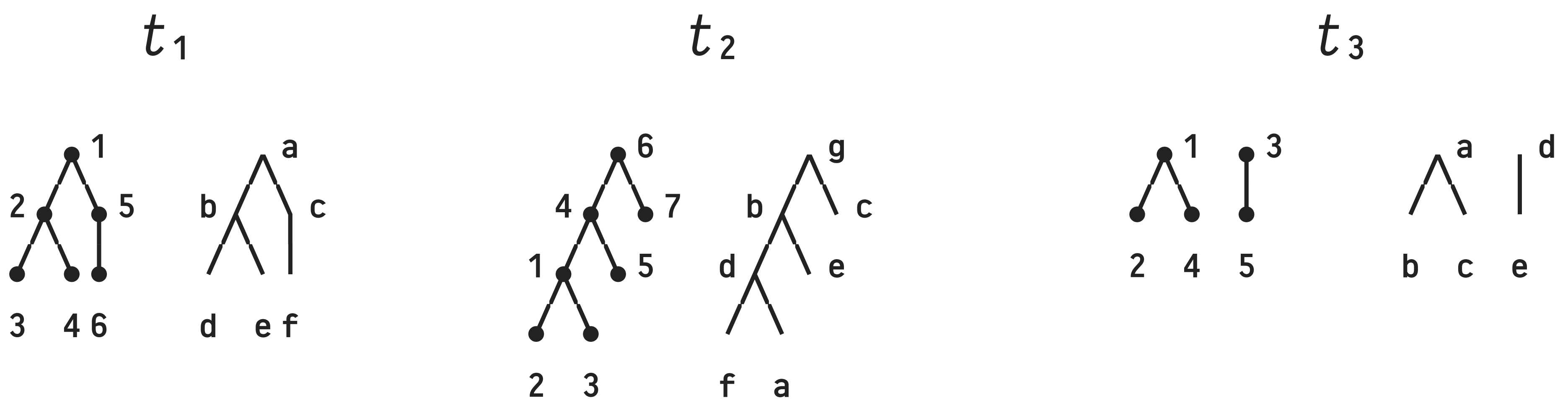
⚠ all sub-arrays need to agree in size

$$\texttt{array[}\overbrace{\texttt{array[}x_{11},\ldots,x_{1n}\texttt{]}},\ldots,\overbrace{\texttt{array[}x_{k1},\ldots,x_{kn}\texttt{]}}\texttt{]}$$

'{{x_{11},…,x_{1n}},…,{x_{k1},…,x_{kn}}}' :: τ[][]



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



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

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 Arrays

- **Append**/**prepend** element ⁜ to array or
- **concatenate** arrays:

```
array_append (array[x₁,…,xₙ],⁜)  ≡  array[x₁,…,xₙ,⁜]
array_prepend(array[x₁,…,xₙ],⁜)  ≡  array[⁜,x₁,…,xₙ]

array_cat(array[x₁,…,xₙ],
          array[y₁,…,yₘ])        ≡  array[x₁,…,xₙ,y₁,…,yₘ]
```

- Overloaded operator || embraces all of the above:

```
xs || ⁜  ≡ array_append(xs, ⁜)
 ⁜ || xs ≡ array_prepend(xs, ⁜)
xs || ys ≡ array_cat(xs,ys)
```

Accessing Arrays: Indexing / Slicing

- Array **indexes** i are 1-based (let $xs \equiv \text{array}[x_1,\dots,x_n]$):

```
   xs[i]      ≡ xᵢ        xs one-dimensional, i ∉ {1,…,n}: NULL
(NULL)[i]     ≡ NULL
   xs[NULL]   ≡ NULL
   xs[i:j]    ≡ array[xᵢ,…,xⱼ]                      i > j: array[]
   xs[i: ]    ≡ array[xᵢ,…,xₙ]
   xs[ :j]    ≡ array[x₁,…,xⱼ]
```

- Access **last element** x_n:

```
xs[array_length(xs,1)]    # of elements in dimension 1: n
xs[cardinality(xs)]
            ↑
 ∑ (# of elements) in all dimensions
```

Searching for Elements in Arrays

Indexing accesses array by position. **Searching** accesses arrays by **contents**, instead.

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

$$x\ \theta\ \text{ANY}(xs) \equiv \exists\ i \in \{1,\cdots,n\}:\ x\ \theta\ xs[i]$$
$$x\ \theta\ \text{ALL}(xs) \equiv \forall\ i \in \{1,\cdots,n\}:\ x\ \theta\ xs[i]$$

```
array_position(xs,※)  ≡ i             if ※ not found: NULL
array_positions(xs,※) ≡ array[i,j]    if ※ not found: array[]
```

$$\text{array\_replace}(xs,※,\#) \equiv \text{array}[x_1,\ldots,\underset{i}{\#},\ldots,\underset{j}{\#},\ldots,x_n]$$

4 ¦ A Bridge Between Arrays and Tables: unnest & array_agg

```
SELECT t.elem
FROM   unnest(array[x_1,…,x_n]) AS t(elem)
```

$$\underbrace{\texttt{array[}x_1,\ldots,x_n\texttt{]}}_{\equiv\ xs}$$

≡

Table t

elem
x_1
⋮
x_n

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

≡

xs
$\{x_1,\ldots,x_n\}$

- unnest(•): a *set-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(array[x_1,…,x_n])
       WITH ORDINALITY AS t(elem,idx)
                                  ↑
```

≡

elem	idx
x_1	1
⋮	⋮
x_n	n

```
                recall ordered aggregates
       ┌───────────────────────────────────┐
SELECT array_agg(t.elem ORDER BY t.idx) AS xs
FROM   (VALUES (x_1,1),
                  ⋮
               (x_n,n)) AS t(elem,idx)
```

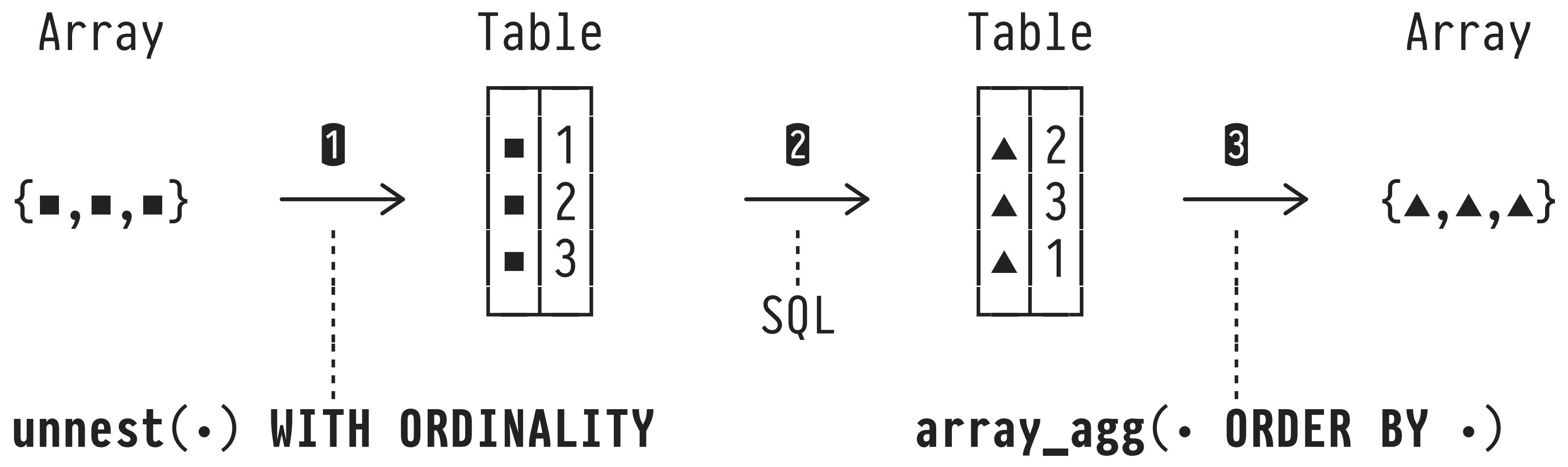
≡

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 Array Programming Pattern

Availability of unnest(•) and ordered array_agg(•) suggests a pattern for **relational array 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* an array PL.

5 ┆ Table-Generating Functions

What is the **type** of unnest(•)?

- unnest(•) establishes a bridge between arrays and SQL's tabular data model:[2]

$$\text{unnest} :: \tau[\,] \rightarrow \text{SETOF}\ \tau$$

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

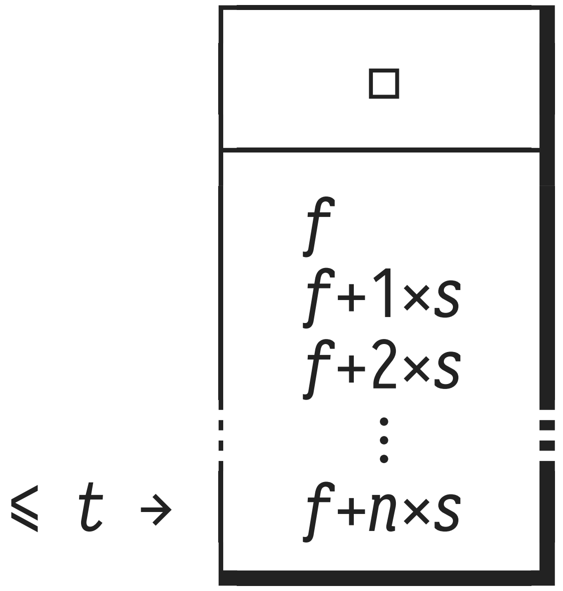
- Several built-in, but may also be **defined by the user.**

[2] Unfortunate naming again: SETOF should probably read BAGOF or TABLE OF.

Series and Subscript Generators

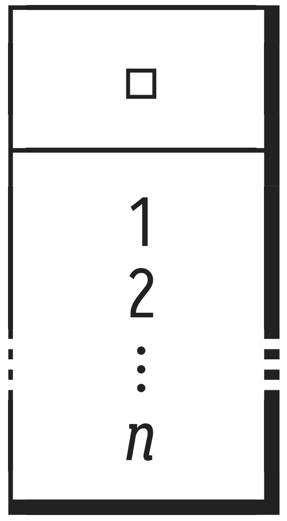
Built-in table-generating functions that generate **tables of consecutive numbers:**

generate_series(f,t,s)



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

generate_subscripts(xs,d)



n ≡ array_length(xs,d)
can also enumerate n,…,1

Text Generators (Regular Expression Matching)

Use *regular expression*[3] *re* to extract **matched substrings** from *t* or **split** text *t* at defined positions:

1. regexp_matches(*t*,*re*,'g'), yields SETOF text[]:
   Generates one array *xs* per match of *re* in *t*. Element *xs*[*i*] holds the **match** of the *i*th *capture group* (in (…)).

2. regexp_split_to_table(*t*,*re*), yields SETOF text:
   Uses the matches of *re* in *t* as *separators* to **split** *t*. Yields table of *n*+1 rows if *re* matches *n* times.

[3] See regexr.com for tutorials and an interactive playground, for example.

Breaking Bad: Parse a Chemical Formula (e.g., $C_6H_5O_7^{3-}$)

```
SELECT t.match[1] AS element,    -- ⎫ extract match details
       t.match[2] AS "# atoms",  -- ⎬ from the (…)
       t.match[3] AS charge      -- ⎭ (capture groups)
FROM   regexp_matches(
          'C₆H₅O₇³⁻',
          '([A-Za-z]+)([₀-₉]*)([⁰-⁹]+[⁺⁻])?',
          'g')
       AS t(match);
```

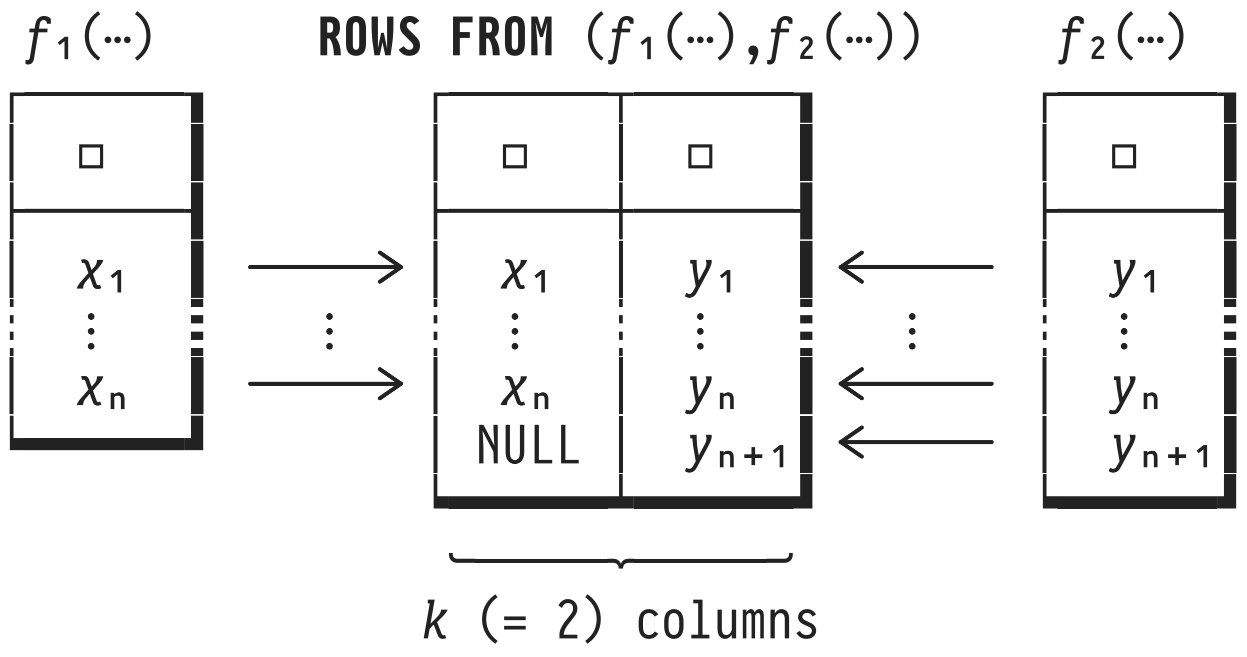
element	# atoms	charge
C	$_6$	NULL
H	$_5$	NULL
O	$_7$	$^{3-}$

} NULL ≡ no match

Zipping Arrays and Table-Generating Functions

Zip: pair elements based on position ("ORDINALITY join"):

- Zipping table functions f_i: **ROWS FROM**(f_1(…),…,f_k(…))
- Zipping arrays xs_i: **unnest**(xs_1,xs_2,…,xs_k)



6 ¦ User-Defined SQL Functions (UDFs)

The body of a **user-defined SQL function (UDFs)** evaluates $n \geqslant 1$ arbitrary SQL statements in sequence:

```
CREATE FUNCTION f(x₁ τ₁,…,xₖ τₖ) RETURNS τ AS
$$
  q₁;     -- ⎫
  q₂;     -- ⎬  evaluate the queries qᵢ in order,
  ⋮       -- ⋮  qₙ defines the result
  qₙ      -- ⎭
$$
LANGUAGE SQL [IMMUTABLE];
                 ↑
 all qᵢ are read-only ⇒ f is free of side effects
```

- UDF f is stored persistently. Remove via DROP FUNCTION.

UDF Types

- UDF f is k-ary with type $\tau_1 \times \cdots \times \tau_k \to \tau$.
    - **Argument types** τ_i must be **atomic** or **row types.**
    - **Overloading** allowed as long as $(f,\tau_1,\ldots,\tau_k)$ is unique.
    - Limited form of **polymorphism:** any τ_i and τ may be anyelement/anyarray/anyenum/anyrange.
        - ⚠️ If any… occurs more than once in the function signature, *all* occurrences denote the *same* type:

$$\begin{aligned}
f_1 &:: \overbrace{\text{anyelement} \times \text{anyelement}}^{=} \to \text{boolean} \\
f_2 &:: \underbrace{\text{anyarray} \times \text{integer} \to \text{anyelement}}_{\text{elem } =}
\end{aligned}$$

UDFs Can Return Tables

A UDF $f :: \tau_1 \times \cdots \times \tau_k \to \tau$ may be of **two flavors:**

	atomic τ	**$\tau \equiv$ SETOF τ'**
If q_n[4] returns no rows,	returns NULL	returns empty table
If q_n returns rows,	returns the first row	returns all rows
May be invoked	wherever $v::\tau$ is used	in the FROM clause

Regular vs. Table-generating UDFs

- A UDF may invoke INSERT/DELETE/UPDATE statements in q_i and thus incur side-effects. (Hmm, UD***F***...ϟ)
  - No IMMUTABLE option—use VOLATILE instead. Consider adding … RETURNING $e_1,\ldots,e_m$ if q_i is the last query.
  - Use CREATE PROCEDURE if f is all about side-effects.

[4] Recall: f's body evaluates queries $q_1,\ldots,q_n$ (in this order).

Example UDF: Map Unicode Subscripts

Map subscript symbol '$_0$',…,'$_9$' to its value in {0,…,9}:

```
CREATE FUNCTION subscript(s text) RETURNS int AS
$$
  SELECT subs.value::int - 1
  FROM   unnest(array['₀','₁','₂',…,'₉'])
         WITH ORDINALITY AS subs(sym,value)
  WHERE  subs.sym = s
$$
LANGUAGE SQL IMMUTABLE;
```

- This is a UDF with atomic return type: yields NULL if s does not denote a valid subscript.

Example UDF: Issue Unique ID, Write Protocol

Generate ID of the form '‹prefix›###' and log time of issue:

```
CREATE FUNCTION new_ID(prefix text) RETURNS text AS
$$
  INSERT INTO issue(id,"when") VALUES
    (DEFAULT, 'now'::timestamp)
  RETURNING prefix || id::text      -- id: just generated
$$
LANGUAGE SQL VOLATILE;       -- function is side-effecting
```

id	when
⋮	⋮
42	2020-05-12 17:26:14.188803
⋮	⋮

Table issue

Example Table-Generating UDF: Flatten a 2D-Array

Unnest 2D array *xss* in *column-major order*:[5]

```
CREATE OR REPLACE FUNCTION unnest2(xss anyarray)
  RETURNS SETOF anyelement AS
$$
  SELECT xss[row][col]
  FROM   generate_subscripts(xss,1) AS row,
         generate_subscripts(xss,2) AS col
  ORDER BY col, row  --  return elements in column-major order
$$
LANGUAGE SQL IMMUTABLE;
```

- ⚠️ Intended type is unnest2 :: τ[][] → SETOF τ.

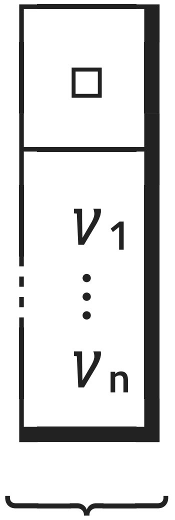
[5] Built-in function unnest(•) can flatten *n*-dimensional arrays in row-major order.

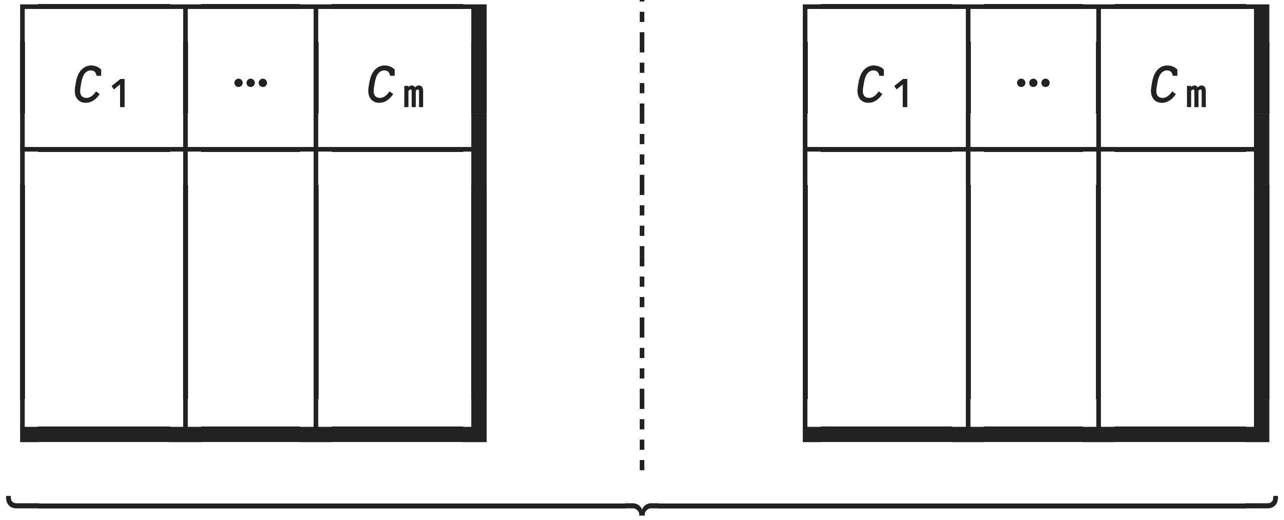
Table-Generating UDFs: Returning Typed Rows

Assume a table-generating UDF f :: … → τ.

If τ ≡

SETOF τ' τ' atomic	SETOF τ' $\tau' \equiv (c_1::\tau_1,\dots,c_m::\tau_m)$	TABLE $(c_1\ \tau_1,\dots,c_m\ \tau_m)$





$v_i::\tau'$

equivalent, but do not need named row type τ'

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

```
SELECT …
FROM   Q₁ AS t₁, Q₂ AS t₂, Q₃ AS t₃ -- tᵢ<ⱼ not free in Qⱼ
```

- 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;           ↑
                            ϟ
```

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

- ⇒ Exception: the arguments of table-generating functions may refer to row variables defined earlier (like t_1).

LATERAL:[6] 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 with table functions.

[6] 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
```

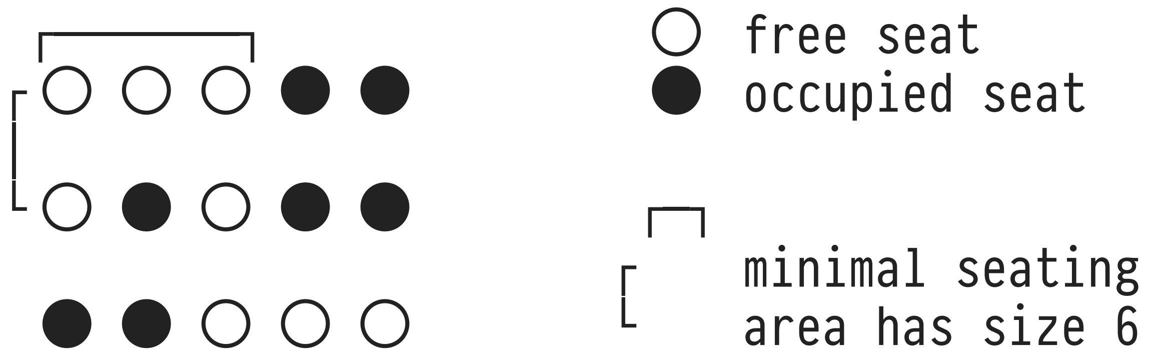
legs	species	height
2	Tyrannosaurus	7
2	Ceratosaurus	4
2	Spinosaurus	2.4
4	Supersaurus	10
4	Brachiosaurus	7.6
4	Diplodocus	3.6

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)

```
\set cinema '...XX\\n.X.XX\\nXX...'

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

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

Finding Seats: A Problem Solution (Generate and Test)

- **Query Plan:**

  1. Determine the extent ($rows \times cols$) of the cinema seating plan.
  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 those with minimal area.

Finding Seats: Generating All Possible Rectangles

Generate all ⌜*nw*,*se*⌟ corners for rectangles up to maximum size *rows* × *cols*:

```
SELECT ROW(row_nw, col_nw) AS nw,
       ROW(row_se, col_se) AS se
FROM   generate_series(1, rows)                 AS row_nw,
       generate_series(1, cols)                 AS col_nw,
       LATERAL generate_series(row_nw, rows) AS row_se,
       LATERAL generate_series(col_nw, cols) AS col_se
```

Generates $\left(\sum_{r=1}^{rows} r\right) \times \left(\sum_{c=1}^{cols} c\right)$ rectangles ⇒ test/filter early!