# Advanced SQL

02 — The Core of SQL

Summer 2020

Torsten Grust Universität Tübingen, Germany

### 1 The Core of SQL

- Let us recollect the **core constructs of SQL**, synchronize notation, and introduce query conventions.
- If you need to refresh your SQL memory, consider
  - the notes for DB1 (Chapters 6 and 9)
  - the PostgreSQL web (Part II, The SQL Language)
- We will significantly expand on this base SQL vocabulary during the semester.

#### Sample Table

Table T serves as a common "playground" for the upcoming SQL queries:

<u>a</u>	b	C	d
1	'X'	true	10
2	'y'	true	40
3	'X'	false	30
4	'y'	false	20
5	'X'	true	NULL

Table T

```
CREATE TABLE T (a <u>int</u> PRIMARY KEY, -- implies NOT NULL
b <u>text</u>, -- here: char(1)
c <u>boolean</u>,
d <u>int</u>);
```

#### 2 Row Variables

• Iterate over all rows of table T (in *some* order: bag semantics), bind **row variable** t to current row:

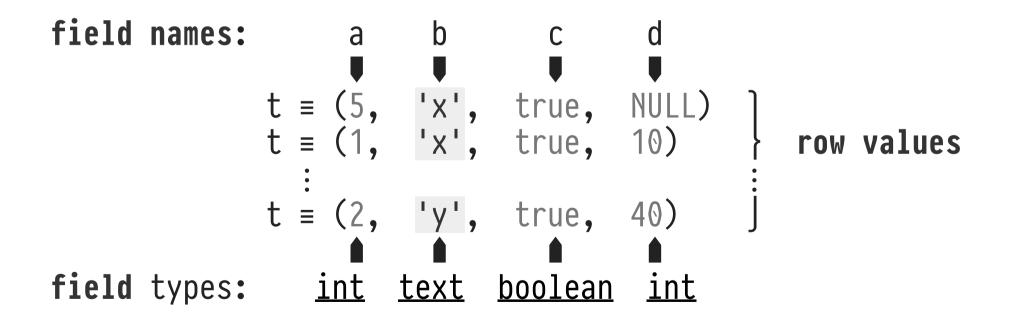
```
SELECT t -- 2 t is bound to current row -- 1 bind/introduce t
```

- If you omit AS t in the FROM clause, a row variable T (generally: AS ) will be implicitly introduced.
- This course: always explicitly introduce/name row variables for disambiguation, clarity, readability.

#### Row Values

```
SELECT t -- 2 t is bound to current row -- 1 bind/introduce t
```

 Row variable t is iteratively bound to row values whose field values and types are determined by the rows of table T:



#### Row Types

- t:: T with T = (a int, b text, c boolean, d int). Row type T is defined when CREATE TABLE T (...) is performed.
- A row type <t> can also be explicitly defined via

## CREATE TYPE <t> AS (a int, b text, c boolean, d int)

 A table T1 equivalent to T — well, almost... — may then be created via

#### CREATE TABLE T1 OF <T>

<sup>1</sup> Read :: as "has type."

#### Row Field Access and \* ("Star")

- Named field access uses dot notation. Assume t :: T and binding t ≡ (5, 'x', true, NULL) then:
  - t.b evaluates to 'x' (of type text),
  - t.d evaluates to NULL (of type int).
- Field names are not first-class in SQL and must be named verbatim (i.e., may not be computed).
- Notation t.\* abbreviates t.a, t.b, t.c, t.d in contexts where this makes sense.<sup>2</sup>

<sup>&</sup>lt;sup>2</sup> t.\* is most often used in SELECT clauses.

#### Row Comparisons

• Row comparisons between rows t<sub>1</sub>, t<sub>2</sub> are performed field-by-field and lexicographically (provided that the field types match). Assume t<sub>1</sub> :: T, t<sub>2</sub> :: T:

```
\circ t_1 = t_2 \iff t_1.a = t_2.a AND \cdots AND t_1.d = t_2.d \circ t_1 < t_2 \iff t_1.a < t_2.a OR (t_1.a = t_2.a AND t_1.b < t_2.b) OR \cdots
```

A row value is NULL iff all of its field values are NULL.

Assume the binding t = (NULL, NULL, NULL, NULL). Then t IS NULL holds.

#### 3 The SELECT Clause

A **SELECT clause** evaluates n expressions  $\langle e_1 \rangle$ , ...,  $\langle e_n \rangle$ :

```
SELECT \langle e_1 \rangle AS \langle c_1 \rangle, ..., \langle e_n \rangle AS \langle c_n \rangle
```

- Creates n columns named  $\langle c_1 \rangle$ , ...,  $\langle c_n \rangle$ .
- In absence of AS  $\langle c_i \rangle$ , PostgreSQL assigns name "?column?" (for all such unnamed columns)  $\Rightarrow$  ambigiuity  $\stackrel{*}{\simeq}$ .
- This course: explicitly use AS to name columns unless a name can be derived from  $\langle e_i \rangle$  (e.g., as in  $\langle e_i \rangle \equiv t.a$ ).
- If column/table names are case-sensitive or contain whitespace/symbols/keywords: wrap in quotes "<ci>".

#### Standalone SELECT

- If query *Q* generates *n* row bindings, SELECT is evaluated *n* times to emit *n* rows (but see *aggregates* below).
- A standalone SELECT (no FROM clause) is evaluated exactly once and emits a single row:

SELECT 1 + 41 AS "The Answer", 'Gla' | 'DOS' AS Portal;

The Answer	portal
42	GlaDOS

### 4 Literal Tables (VALUES)

A VALUES clause constructs a transient table from a list of provided row values <e;>:

#### **VALUES** $\langle e_1 \rangle$ , ..., $\langle e_n \rangle$

- If n > 1, the  $\langle e_i \rangle$  must agree in arity and field types (row value  $\langle e_1 \rangle$  is used to infer and determine types).
- VALUES automatically assigns column names "column $\langle i \rangle$ ". Use column aliasing to assign names (see FROM below).
- Orthogonality: a VALUES clause (in parentheses (...)) may be used anywhere a SQL query expects a table.

# 5 Generating Row Variable Bindings (FROM)

A FROM clause expects a set of tables  $\langle T_i \rangle$  and successively binds the row variables  $\langle t_i \rangle$  to the tables' rows:

```
SELECT ... T_1 > AS < t_1 > \dots, < T_n > AS < t_n > \dots
```

- The  $\langle T_i \rangle$  may be table names or SQL queries computing tables (in  $(\cdots)$ ).
- If you need to rename the columns of  $\langle T_i \rangle$  (recall the VALUES clause), use **column aliasing** on all (or only the first  $k \cong$ ) columns:

$$\langle T_i \rangle$$
 AS  $\langle t_i \rangle (\langle c_{i1} \rangle, \ldots, \langle c_{ik} \rangle)$ 

## FROM Computes Cartesian Products

```
SELECT ... FROM \langle T_1 \rangle AS \langle t_1 \rangle, ..., \langle T_n \rangle AS \langle t_n \rangle
```

- This FROM clause generates  $|\langle T_1 \rangle| \times \cdots \times |\langle T_n \rangle|$  bindings. Semantics: compute the **Cartesian product**  $\langle T_1 \rangle \times \cdots \times \langle T_n \rangle$ , draw the bindings for the  $\langle t_i \rangle$  from this product.
- FROM operates over a *set* of tables (',' is associative and commutative).
- In particular, row variable  $\langle t_i \rangle$  is not in scope in the table subqueries  $\langle T_{i+1} \rangle$ , ...,  $\langle T_n \rangle$ .

# 6 WHERE Discards Row Bindings

A WHERE clause introduces a predicate  $\langle p \rangle$  that is evaluated under all row variable bindings generated by FROM:

```
SELECT ... -- 3 FROM \langle T_1 \rangle AS \langle t_1 \rangle, ..., \langle T_n \rangle AS \langle t_n \rangle -- 1 WHERE \langle p \rangle -- 2
```

- All row variables  $\langle t_i \rangle$  are in scope in  $\langle p \rangle$ .
- Only bindings that yield  $\langle p \rangle = \text{true}$  are passed on.<sup>3</sup>
- Absence of a WHERE clause is interpreted as WHERE true.

<sup>&</sup>lt;sup>3</sup> If  $\langle p \rangle$  evaluates to NULL  $\neq$  true, the binding is discarded.

# 7 Compositionality: Subqueries Instead of Values

The meaning of a complex expression is determined by the meanings of constituent expressions. ??

—Principle of Compositionality

With the advent of the SQL-92 and SQL:1999 standards, SQL has gained in compositionality and orthogonality:

- Wherever a (tabular or scalar) value  $\nu$  is required, a SQL expression in (...)—a **subquery**—may be used to compute  $\nu$ .
- Subqueries nest to arbitrary depth.

#### Scalar Subqueries: Atomic Values

A SQL query that computes a **single-row**, **single-column table** (column name — irrelevant) may be **used in place of an atomic value**  $\nu$ :



#### In a scalar subquery...

- ... an empty table is interpreted as NULL,
- ... a table with > 1 rows or > 1 columns will yield a runtime error ...

#### Scalar Subqueries: Atomic Values

```
generate single column

SELECT 2 + (SELECT t.d AS _
FROM T AS t
WHERE t.a = 2) AS "The Answer"

equality predicate on key column,
will yield ≤ 1 rows
```

- Runtime errors for WHERE t.a > 2 or SELECT t.a, t.d
- Yields NULL: WHERE t.a = 0
- AS \_ assigns a "don't care" column name this is a case where column naming is obsolete and adds nothing.

#### Scalar Subqueries: Row Values

A SQL query that computes a **single-row table** with column names  $\langle c_i \rangle$  may be **used in place of row value**  $(v_1, ..., v_n)$  with field names  $\langle c_i \rangle$ :

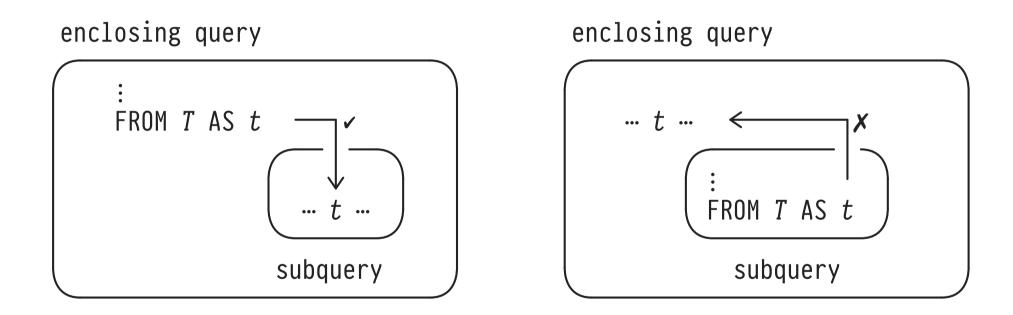


#### In a scalar subquery...

- ... an empty table is interpreted as (NULL, ..., NULL),
- ... a table with > 1 rows will yield a runtime error.

### Row Variable Scoping

Subqueries may refer to any row variable t bound in their enclosing queries (up to the top-level query):



• Note: From inside the subquery—*i.e.*, inside the (…)—row variable *t* is *free*.

# Subqueries, Free Row Variables, Correlation

• If t is free in subquery q, we may understand the subquery as a function q(t): you supply a value for t, I will compute the (tabular) value of q:

```
SELECT t1.*

FROM T AS t1

WHERE t1.b <> (SELECT t2.b
FROM T AS t2
WHERE t1.a = t2.a)

In the sunder t1 bindings:

t1 \equiv (1, ...)
t1 \equiv (2, ...)
t1 \equiv (3, ...)
t1 \equiv (4, ...)
t1 \equiv (4, ...)
t1 \equiv (5, ...)
```

 Subqueries featuring free variables are also known as correlated.

```
8 Row Ordering (ORDER BY)
```

SQL tables are unordered bags of rows, but rows may be locally ordered for result display or positional access:

```
SELECT ... -- 3

FROM ... -- 1

WHERE ... -- 2

ORDER BY \langle e_1 \rangle, ..., \langle e_n \rangle -- 4
```

- The order of the  $\langle e_i \rangle$  matters: sort order is determined lexicographically with  $\langle e_1 \rangle$  being the major criterion.
- The sort criteria <e;> are expressions that may refer to column names in the SELECT clause.

#### SELECT t.\* FROM T AS t ...

	a	b	С	d	
	5	'X'	true	NULL	
	1	'X'	true	10	
	4	'y'	false	20	
	3	1X1	false	30	
	2	'y'	true	40	
··· ORDE	R	BY t.	d ASC	NULLS	FIRST

a	b	С	d	
4	'y'	false	20	
2	۱ý۱	true	40	
3	ΙΧΊ	false	30	
1	' X '	true	10	
5	1 X 1	true	NULL	
ORD	ER BY	t.b D	ESC,	t.c

 Note: ASC (ascending) is default. NULL is larger than any non-NULL value. Ties<sup>\*</sup>: order is implementation-dependent.

#### Row Order is Local Only

ORDER BY establishes a well-defined row order that is **local** to the current (sub)query:

```
may yield rows in any order

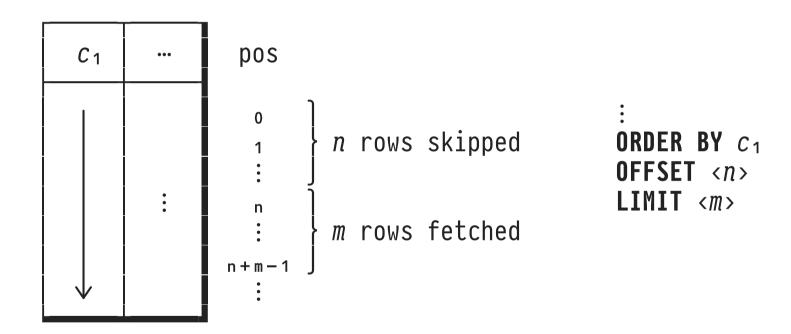
SELECT t1.*
FROM (SELECT t2.*
FROM T AS t2
ORDER BY t2.a) AS t1;

guaranteed row order
inside the subquery only
```

- Never rely on row orders that appear consistent across runs — may vary between DBMSs, presence of indexes, etc.
- Q: What, then, is such local row order good for?

#### Positional Access to Rows

Once row order has been established it makes sense to "skip to the nth row" or "fetch the next m rows."



- OFFSET 0: read from the start. LIMIT ALL: fetch all rows.
- Alternative syntax: FETCH [FIRST | NEXT] <m> ROWS ONLY.

# 9 Identify Particular Rows Among Peers (DISTINCT ON)

Extract the first row among a group of equivalent rows:

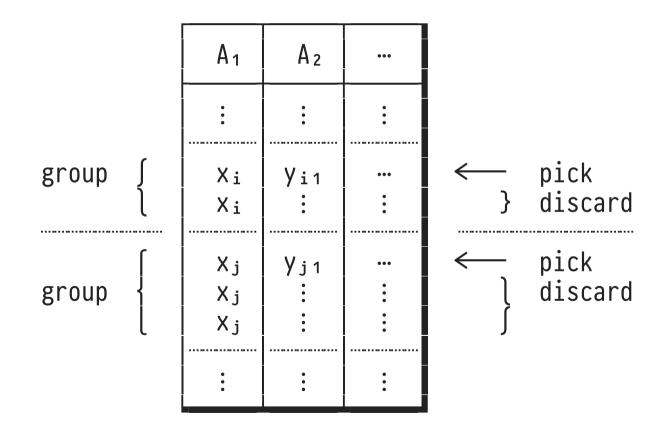
```
prefix of ORDER BY clause 

SELECT DISTINCT ON \Omega (\langle e_1 \rangle, \ldots, \langle e_n \rangle) \langle c_1 \rangle, \ldots, \langle c_k \rangle -- \Omega FROM ... ORDER BY \langle e_1 \rangle, \ldots, \langle e_n \rangle, \langle e_{n+1} \rangle, \ldots, \langle e_m \rangle -- \Omega
```

- 1. Sort rows in  $\langle e_1 \rangle, \ldots, \langle e_n \rangle, \langle e_{n+1} \rangle, \ldots, \langle e_m \rangle$  order.
- 2. Rows with identical  $\langle e_1 \rangle, \ldots, \langle e_n \rangle$  values form one **group**.
- 3. From each of these groups, pick the first row in  $\langle e_{n+1} \rangle, \ldots, \langle e_m \rangle$  order.
- A Without ORDER BY, step 3 picks any row in each group.

## DISTINCT ON: Group, Then Pick First in Each Group

```
SELECT DISTINCT ON (A<sub>1</sub>) ... -- For each A<sub>1</sub>, pick the row ... FROM ... ORDER BY A<sub>1</sub>, A<sub>2</sub> DESC -- ... with the largest A<sub>2</sub>
```



## DISTINCT: Table-Wide Duplicate Removal

Keep only a single row from each group of duplicates:

```
SELECT DISTINCT 2 < c_1 > \ldots < c_k > -- 2
FROM \ldots
```

- True duplicate removal: rows are considered identical if they agree on **all** k columns  $\langle c_i \rangle$ .
- Row order is irrelevant. DISTINCT returns a set of rows.
- May use SELECT ALL ... to explicitly document that a query is expected to return duplicate rows.

<sup>&</sup>lt;sup>4</sup> This is equivalent to SELECT DISTINCT ON  $(\langle c_1 \rangle, ..., \langle c_k \rangle)$   $\langle c_1 \rangle, ..., \langle c_k \rangle$  FROM ....

# 10 | Summarizing Values: Aggregates

**Aggregate functions** (short: aggregates) reduce a collection of values to a single value (think summation, maximum).

• Simplest form: *collection* ≡ entire table:

```
SELECT \langle agg_1 \rangle (\langle e_1 \rangle) AS \langle c_1 \rangle, ..., \langle agg_n \rangle (\langle e_n \rangle) AS \langle c_n \rangle FROM ...
```

- Reduction of input rows: result table will have one row.
- Cannot mix aggregates with non-aggregate expression <e>
   in SELECT clause: which value of <e>
   should we pick?

<sup>&</sup>lt;sup>5</sup> But see GROUP BY later on.

### Aggregate Functions: Semantics

```
SELECT agg(e) AS c — e will typically refer to t — range over entire table T
```

Aggregate agg defined by triple (Φ<sup>agg</sup>, z<sup>agg</sup>, ⊕<sup>agg</sup>):
Φ<sup>agg</sup> (empty): aggregate of the empty value collection
z<sup>agg</sup> (zero): aggregate value initialiser
⊕<sup>agg</sup> (merge): add value to existing aggregate

```
a \leftarrow \phi^{agg} -- a will be aggregate value

for t in T -- iterate over all rows of T

x \leftarrow e(t) -- value to be aggregated

if x \neq \text{NULL} -- aggregates ignore NULL values (*)

a \leftarrow z^{agg} -- once we see first non-NULL value:

a \leftarrow \theta^{agg}(a, x) -- maintain running aggregate
```

#### Aggregate Functions: Semantics

Aggregate agg	φagg	Z <sup>agg</sup>	$\bigoplus$ agg(a, $\chi$ )
COUNT	0	0	a + 1
SUM	$NULL^6$	0	a + x
AVG <sup>7</sup>	NULL	<0, 0>	(a.1 + x, a.2 + 1)
MAX	NULL	-∞	$\max_{2}(a, x)$
MIN	NULL	+∞	$\min_{2}(a, x)$
bool_and	NULL	true	$a \wedge x$
bool_or	NULL	false	$a \vee x$
•	•	•	

 The special form COUNT(\*) counts rows regardless of their fields' contents (NULL, in particular).

<sup>&</sup>lt;sup>6</sup> If you think "this is wrong," we're two already. Possible upside: sum differentiates between summation over an empty collection vs. a collection of all 0s.

<sup>&</sup>lt;sup>7</sup> Returns a.1 / a.2 as final aggregate value.

#### Aggregate Functions on Table T

#rows	#d	Σd	max(b)	Уc	0E=bE
5	4	100	'y'	false	true
		< <p>&lt;<p>&lt;<p>&lt;<p>&lt;<p>&lt;<p>&lt;<p>&lt;<p>&lt;</p></p></p></p></p></p></p></p>	∍ ≡ true	9	

$$\langle p \rangle \equiv false$$

#### Ordered Aggregates

- For most aggregates agg, ⊕<sup>agg</sup> is commutative (and associative): row order does not matter.
- Order-sensitive aggregates admit a trailing

  ORDER BY <e\_1>,...,<e\_n> argument that defines row order:<sup>8</sup>

```
-- cast to text separator string

SELECT string_agg(t.a :: text, ',' ORDER BY t.d) AS "all a"

FROM T AS t
```

all a '1,4,3,2,5'

 $<sup>^{8}</sup>$   $\oplus$  string-agg essentially is || (string concatenation) which is not commutative.

## Filtered and Unique Aggregates

```
SELECT <agg>(<e>) FILTER (WHERE )
FROM ...
```

• FILTER clause alters aggregate semantics (see \*):

```
:

x ← e(t)

if x ≠ NULL ∧ p(x):

:
```

```
SELECT <agg>(DISTINCT <e>)
FROM ...
```

Aggregates distinct (non-NULL) values of expression <e>.
 (May use ALL to flag that duplicates are expected.)

# 11 Forming Groups of Rows

Once FROM has generated row bindings, SQL clauses operate row-by-row. After GROUP BY: operate group-by-group:

```
      SELECT (e<sub>1</sub>), ..., (e<sub>m</sub>)
      --
      5

      FROM ...
      --
      1

      WHERE ...
      --
      2

      GROUP BY (g<sub>1</sub>), ..., (g<sub>n</sub>)
      --
      3

      HAVING (p)
      --
      4
```

- All rows that agree on all expressions  $\langle g_i \rangle$  (the set of grouping criteria) form one group.
- $\Rightarrow$  Steps **4** and **5** process groups (*not* individual rows). This affects expressions  $\langle p \rangle$  and the  $\langle e_i \rangle$ .

#### **GROUP BY Partitions Rows**

FROM ... evaluated once per group (not per row)
HAVING ...

	A <sub>1</sub>	A <sub>2</sub>	•••
	•	•	:
the $x_i$ group $\left\{ \right.$	X <sub>i</sub> X <sub>i</sub>	Уі1 Уі2	:
the x <sub>j</sub> group {	Хj Хj	Уј1 Уј2	:
	:	•	:

Grouping partitions the row bindings:

- there are no empty groups
- each row belongs to exactly one group

### **GROUP BY** Changes Field Types From τ To bag(τ)<sup>9</sup>

```
SELECT t.b, t.d SELECT the(t.b) AS b, SUM(t.d) AS "\Sd"
FROM T AS t
GROUP BY t.b

SELECT the(t.b) AS b, SUM(t.d) AS "\Sd"
FROM T AS t
GROUP BY t.b
```

- t.d references current group of d values: violates 1NF!
   ⇒ After GROUP BY: must use aggregates on field values.
- t.b references current group of b values all of which are
   equal in a group ⇒ SQL: using just t.b is OK.
- (\* May think of **hypothetical** aggregate the(<e>) that picks one among equal <e> values.)

<sup>&</sup>lt;sup>9</sup> A view of GROUP BY that is due to Philip Wadler.

#### Aggregates are Evaluated Once Per Group

group	size	Σd	∀even(a)	all a
'x'	3	40	false	'1;3;5'
'y'	2	60	true	'2 <b>;</b> 4'

HAVING acts like WHERE but after grouping:
 = false discards groups (not rows).

#### **Grouping Criteria**

- The grouping criteria <g<sub>i</sub>> form a set—order is irrelevant.
- Grouping on a **key** effectively puts each row in its own singleton group. (Typically a query smell .)
- Expressions <e> that are functionally dependent on the
   <gi> are constant within a group (and thus can be used in SELECT).
  - $\circ$  If SQL does not know about the FD, explicitly add  $\langle e \rangle$  to the set of  $\langle g_i \rangle$ —this will not affect the grouping.

# 12 | Bag and Set Operations

Tables contain **bags of rows.** SQL provides the common family of binary **bag operations** (*no* row order):

```
\langle q_1 \rangle UNION ALL \langle q_2 \rangle --- U<sup>+</sup> (bag union) \langle q_1 \rangle INTERSECT ALL \langle q_2 \rangle --- N<sup>+</sup> (bag intersection) \langle q_1 \rangle EXCEPT ALL \langle q_2 \rangle --- \\ (bag difference)
```

- Row types (field names/types) of queries  $\langle q_i \rangle$  must match.
- With ALL, row multiplicities are respected: if row r occurs  $n_i$  times in  $\langle q_i \rangle$ , r will occur  $\max(n_1-n_2,0)$  times in  $\langle q_1 \rangle$  EXCEPT ALL  $\langle q_2 \rangle$  (INTERSECT ALL:  $\min(n_1,n_2)$ ).
  - Without ALL: obtain set semantics (no duplicates).

#### 13 | Multi-Dimensional Data

- Relational representation of *measures* (*facts*) depending on multiple parameters (*dimensions*).
- Example: table prehistoric with **dimensions** class, herbivore?, legs, **fact** species:

<u>class</u>	<u>herbivore?</u>	<u>legs</u>	species
'mammalia'	true	2	'Megatherium'
'mammalia'	true	4	'Paraceratherium'
'mammalia'	false	2	NULL
'mammalia'	false	4	'Sabretooth'
'reptilia'	true	2	'Iguanodon'
'reptilia'	true	4	'Brachiosaurus'
'reptilia'	false	2	'Velociraptor'
'reptilia'	false	4	NULL

Table prehistoric

#### Multiple GROUP BYs: GROUPING SETS

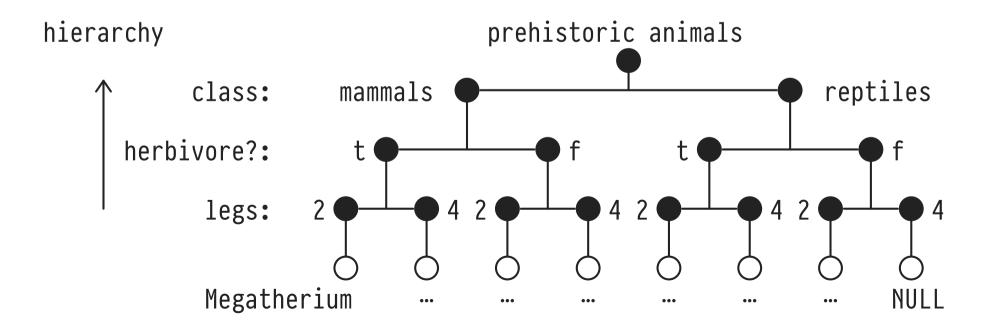
- Analyze (here: group, then aggregate) table ⟨T⟩ along multiple dimensions ⇒ perform separate GROUP BYs on each relevant dimension:
- SQL syntactic sugar:

```
SELECT <e<sub>1</sub>>, ..., <e<sub>m</sub>>
FROM <T> -- G<sub>i</sub>: grouping criteria
GROUP BY GROUPING SETS (G<sub>1</sub>,...,G<sub>n</sub>) -- sets in (…)
```

Yields n individual GROUP BY queries q<sub>i</sub>, glued together
 by UNION ALL. If ⟨e<sub>j</sub>⟩ ∉ G<sub>i</sub>, ⟨e<sub>j</sub>⟩ ≡ NULL in q<sub>i</sub>.

• Group along a path from any node Gn up to the root:

ROLLUP 
$$(G_1,...,G_n) \equiv GROUPING SETS ((G_1,...,G_{n-1},G_n), (G_1,...,G_{n-1}), ..., (G_1), (G_1),$$



### Analyze All Dimension Combinations: CUBE

• slice for herbivore? = true herbivore? legs mam rep class

CUBE 
$$(G_1,...,G_n) \equiv GROUPING SETS ((G_1,...,G_n), \\ \vdots \\ ())$$
 all  $2^n$  subsets considered

# 14 | SQL Evaluation vs. Reading Order

```
SELECT DISTINCT ON (<es> 0) <es> 0, <aggs> 0
FROM (qs>
WHERE  2
GROUP BY (es> 4
HAVING 
  UNION / EXCEPT / INTERSECT 3 | repeated 0 or more times, all evaluated before 9
ORDER BY <es> 0
OFFSET <n>
LIMIT <n>
```

• Reading order is: (7,3,6,1,0,4,5,8)+,9,0.

#### Query Nesting and (Non-)Readability

```
SELECT
FROM
       (SELECT ...
        FROM (SELECT ...
                 FROM ...
                 : ) AS <descriptive>
        : ) AS ...
```

 The more complex the query and the more useful the <descriptive> name becomes, the deeper it is buried.



• Query is a **syntactic monolith**. Tough to develop a query in stages/phases and assess the correctness of its parts.

# 15 The let...in of SQL: WITH (Common Table Expressions)

Use **common table expressions (CTEs)** to bind table names before they are used, potentially multiple times:

- "Literate SQL": Reading and writing order coincide.
- Think of let  $\langle T_1 \rangle = \langle q_1 \rangle$ , ... in  $\langle q \rangle$  in your favorite FP language. The  $\langle T_i \rangle$  are undefined outside WITH.

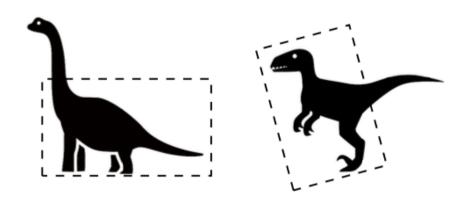
#### SQL With WITH

- 1. **Define queries in stages,** intermediate results in tables  $\langle T_i \rangle$ . May use  $\langle q \rangle \equiv TABLE \langle T_i \rangle^{10}$  to debug stage i.
- 2. Bundle a query with test data:

<sup>10</sup> Syntactic sugar for SELECT t.\* FROM <Ti> AS t.

# 16 Use Case: WITH (Dinosaur Body Shapes)

Paleontology: dinosaur body shape (height/length ratio) and form of locomotion (using 2 or 4 legs) correlate:



 Use this correlation to infer bipedality (quadropedality) in incomplete dinosaur data sets:

<u>species</u>	height	length	legs
Gallimimus	2.4	5.5	?

# Y Dinosaur Body Shapes

<u>species</u>	height	length	legs
Ceratosaurus	4.0	6.1	2
Deinonychus	1.5	2.7	2
Microvenator	0.8	1.2	2
Plateosaurus	2.1	7.9	2
Spinosaurus	2.4	12.2	2 2 2 2 2
Tyrannosaurus	7.0	15.2	2
Vélociraptor	0.6	1.8	2
Apatosaurus	2.2	22.9	4
Brachiosaurus	7.6	30.5	4
Diplodocus	3.6	27.1	4
Supersaurus	10.0	30.5	4
Albertosaurus	4.6	9.1	NULL
Argentinosaurus	10.7	36.6	NULL
Compsognathus	0.6	0.9	NULL
Gallimimus	2.4	5.5	NULL
Mamenchisaurus	5.3	21.0	NULL
Oviraptor	0.9	1.5	NULL
Ultrasaurus	8.1	30.5	NULL

Table dinosaurs

## Y Dinosaur Body Shapes

```
WITH
bodies(legs, shape) AS (
    SELECT d.legs, AVG(d.height / d.length) AS shape
    FROM dinosaurs AS d
    WHERE d.legs IS NOT NULL
    GROUP BY d.legs
)
:
```

<u>legs</u>	shape
2	0.201
4	0.447

Transient Table bodies

# Y Dinosaur Body Shapes

#### • Query Plan: 11 🗨

- 0. Assume average body shapes in bodies are available
- 1. Iterate over all dinosaurs d:
  - If locomotion for d is known, output d as is
  - If locomotion for d is unknown:
    - Compute body shape for d
    - Find the shape entry b in bodies that matches d the closest
    - Use the locomotion in b to complete d, output completed d

<sup>&</sup>lt;sup>11</sup> In this course, *query plan* refers to a "plan of attack" for a query problem, not EXPLAIN output.