

Advanced SQL

02 — The Core of SQL

Summer 2022

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

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 of T
FROM   T AS t     -- 1 bind/introduce t
```

- If you omit **AS t** in the **FROM** clause, a row variable **T** (generally: **AS <table name>**) 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 of T
FROM   T AS t -- 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`:

field names:	a	b	c	d	
	↓	↓	↓	↓	
t ≡	(5,	'x',	true,	NULL)	} row values
t ≡	(1,	'x',	true,	10)	
⋮					
t ≡	(2,	'y',	true,	40)	
	↑	↑	↑	↑	
field types:	<u>int</u>	<u>text</u>	<u>boolean</u>	<u>int</u>	

Row Types

- `t :: T` with `T = (a int, b text, c boolean, d int)`.¹ **Row type** `T` is defined when `CREATE TABLE T (...)` is performed.
- A row type `τ` can also be explicitly defined via

```
CREATE TYPE τ 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 τ
```

¹ 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 provided verbatim (i.e., may *not* be computed).
- Notation `t.*` abbreviates `t.a, t.b, t.c, t.d` in contexts where this makes sense.²

² `t.*` is most often used in `SELECT` clauses.

Row Comparisons

- **Row comparisons** between rows t_1 , t_2 are performed field-by-field and lexicographically (provided that the field types match). Assume $t_1 :: T$, $t_2 :: T$:
 - $t_1 = t_2 \iff t_1.a = t_2.a \text{ AND } \dots \text{ AND } t_1.d = t_2.d$
 - $t_1 < t_2 \iff$
 $t_1.a < t_2.a \text{ OR } (t_1.a = t_2.a \text{ AND } t_1.b < t_2.b) \text{ OR } \dots$
- A row value is **NULL** iff *all* of its field values are **NULL**.

Assume the binding $t \equiv (\text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})$. Then $t \text{ IS NULL}$ holds.

3 | The **SELECT** Clause

A **SELECT** clause evaluates n expressions e_1, \dots, e_n :

```
SELECT  $e_1$  AS  $c_1$ , ...,  $e_n$  AS  $c_n$ 
```

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

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_i :

VALUES e_1, \dots, e_n

- If $n > 1$, the e_i must agree in arity and field types (row value e_1 is used to infer and determine types).
- **VALUES** automatically assigns column names "column< i >". 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 T_i and successively binds the row variables t_i to the tables' rows:

```
SELECT ...           -- 2
FROM    $T_1$  AS  $t_1$ , ...,  $T_n$  AS  $t_n$   -- 1
```

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

T_i AS $t_i(\underline{c_{i,1}}, \dots, \underline{c_{i,k}})$

FROM Computes Cartesian Products

```
SELECT ...  
FROM     $T_1$  AS  $t_1$ , ...,  $T_n$  AS  $t_n$ 
```

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

6 : WHERE Discards Row Bindings

A **WHERE** clause introduces a predicate p that is evaluated under all row variable bindings generated by **FROM**:

SELECT	...	--	3
FROM	T_1 AS t_1 , ..., T_n AS t_n	--	1
WHERE	p	--	2

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

³ If p 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 *v* is required, a SQL expression in *(...)*—a **subquery**—may be used to compute *v*.
- Subqueries nest to arbitrary depth.

Scalar Subqueries: Atomic Values

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



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`
- Subquery 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 c_i may be **used in place of row value** (v_1, \dots, v_n) with field names c_i :

c_1	\dots	c_n
v_1	\dots	v_n

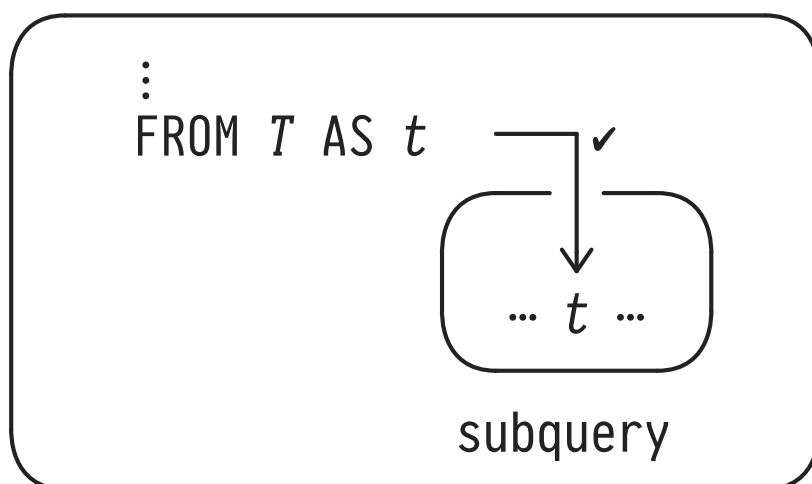
In a **scalar subquery**...

- ... an empty table is interpreted as $(\text{NULL}, \dots, \text{NULL})$,
- ... a table with > 1 rows will yield a **runtime error**.

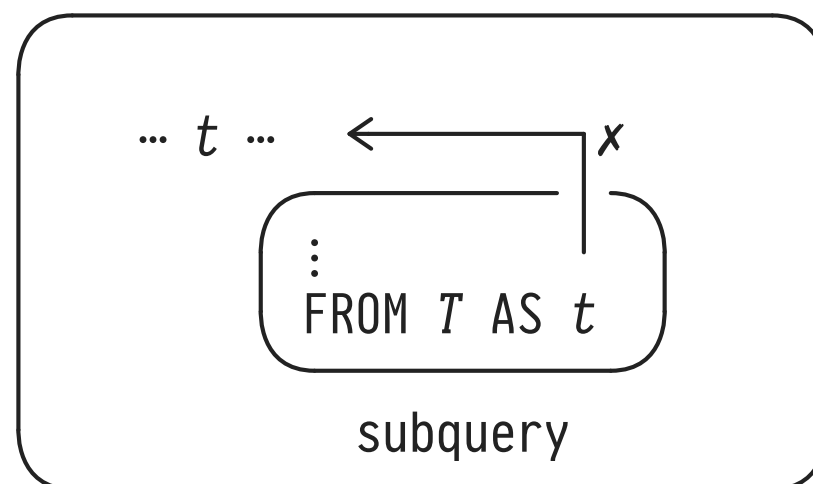
Row Variable Scoping

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

enclosing query



enclosing query



- NB: Seen from inside the subquery—*i.e.*, inside the (...)—row variable t is *free* (binding provided externally).

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 :

<pre> SELECT t1.* FROM T AS t1 WHERE t1.b <> (SELECT t2.b FROM T AS t2 WHERE t1.a = t2.a) </pre>	}	<p>evaluated 5 times under t1 bindings:</p> <p>t1 ≡ (1, ...)</p> <p>t1 ≡ (2, ...)</p> <p>t1 ≡ (3, ...)</p> <p>t1 ≡ (4, ...)</p> <p>t1 ≡ (5, ...)</p>
<p>↑</p> <p>free</p>		

- 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  $e_1, \dots, e_n$   -- 4
```

- The order of the e_i matters: sort order is determined lexicographically with e_1 being the major criterion.
- Sort criteria e_i are expressions that may refer to column names in the **SELECT** clause (evaluated after **SELECT**).

SELECT t.* FROM T AS t ...

a	b	c	d
5	'x'	true	NULL
1	'x'	true	10
4	'y'	false	20
3	'x'	false	30
2	'y'	true	40

... ORDER BY t.d ASC NULLS FIRST

a	b	c	d
4	'y'	false	20
2	'y'	true	40
3	'x'	false	30
1	'x'	true	10
5	'x'	true	NULL

... ORDER BY t.b DESC, t.c

- Note: **ASC** (ascending) is default. **NULL** is larger than any non-**NULL** value. Ties: 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

↓

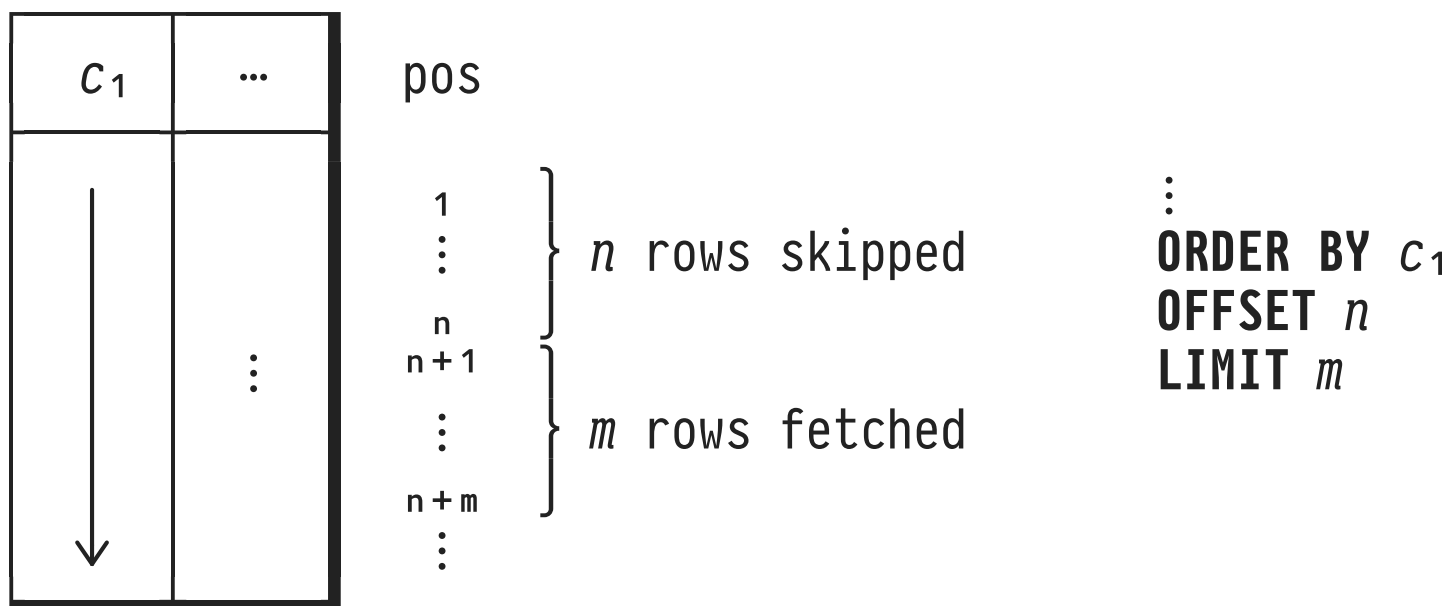
```
SELECT t1.*  
FROM   (SELECT t2.*  
        FROM   T AS t2  
        ORDER BY t2.a) AS t1;
```

} 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 n^{th} 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	4 (e_1, \dots, e_n)	-- 2
FROM	\dots	-- 1
ORDER BY	$e_1, \dots, e_n, e_{n+1}, \dots, e_m$	-- 3

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

DISTINCT ON: Group, Then Pick First in Each Group

SELECT DISTINCT ON (A_1) ... -- For each A_1 , pick the row ...
FROM ...
ORDER BY A_1 , A_2 **DESC** -- ... with the largest A_2

	A_1	A_2	...
	\vdots	\vdots	\vdots
group {	x_i	y_{i1}	...
	x_i	\vdots	\vdots
group {	x_j	y_{j1}	...
	x_j	\vdots	\vdots
	x_j	\vdots	\vdots
	\vdots	\vdots	\vdots

← } pick
discard

← } pick
discard

DISTINCT: Table-Wide Duplicate Removal

Keep only a single row from each group of **duplicates**:

```
SELECT DISTINCT ③  c1, ..., ck      -- ②
FROM             ...                -- ①
```

- True duplicate removal: rows are considered identical if they agree on **all** *k* columns *c_i*.⁴
- 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.

⁴ This is equivalent to **SELECT DISTINCT ON (c₁,...,c_k) c₁,...,c_k FROM ...**.

10 | Summarizing Values: Aggregates

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

- Simplest form: *collection* \equiv entire table:

```
SELECT agg1(e1) AS c1, ..., aggn(en) AS cn  
FROM    ...
```

- Reduction of input rows: result table will have **one row**.
- Cannot mix aggregates with non-aggregate expression *e* in a **SELECT** clause:⁵ which value of *e* should we pick?

⁵ But see **GROUP BY** later on.

Aggregate Functions: Semantics

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

- Aggregate *agg* defined by triple $(\phi^{agg}, z^{agg}, \oplus^{agg})$:
 - ϕ^{agg} (*empty*): aggregate of the empty value collection
 - z^{agg} (*zero*): aggregate value initialiser
 - \oplus^{agg} (*merge*): add value to existing aggregate

```
a ←  $\phi^{agg}$            -- a will be aggregate value
for t in T            -- iterate over all rows of T
| x ← e(t)            -- value to be aggregated
| if x ≠ NULL         -- aggregates ignore NULL values (*)
| | if a =  $\phi^{agg}$     -- once we see first non-NULL value:
| | | a ←  $z^{agg}$       -- initialize aggregate
| | | a ←  $\oplus^{agg}(a, x)$  -- maintain running aggregate
```

Aggregate Functions: Semantics

Aggregate agg	ϕ^{agg}	z^{agg}	$\oplus^{agg}(a, x)$
COUNT	0	0	$a + 1$
SUM	NULL ⁶	0	$a + x$
AVG ⁷	NULL	$\langle 0, 0 \rangle$	$\langle a.1 + x, a.2 + 1 \rangle$
MAX	NULL	$-\infty$	$\max_2(a, x)$
MIN	NULL	$+\infty$	$\min_2(a, x)$
bool_and	NULL	true	$a \wedge x$
bool_or	NULL	false	$a \vee x$
\vdots	\vdots	\vdots	\vdots

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

⁶ 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.

⁷ Returns $a.1 / a.2$ as final aggregate value.

Aggregate Functions on Table T

```

SELECT COUNT(*)           AS "#rows",
       COUNT(t.d)         AS "#d",
       SUM(t.d)            AS "Σd",
       MAX(t.b)            AS "max(b)",
       bool_and(t.c)       AS "∀c",
       bool_or(t.d = 30)   AS "∃d=30"
FROM   T AS t
WHERE  p

```

#rows	#d	Σd	max(b)	∀c	∃d=30
5	4	100	'y'	false	true

$p \equiv \text{true}$

#rows	#d	Σd	max(b)	∀c	∃d=30
0	0	NULL	NULL	NULL	NULL

$p \equiv \text{false}$

Ordered Aggregates

- For most aggregates *agg*, merge \oplus^{agg} is commutative (and associative): row order does not matter.
- **Order-sensitive aggregates** admit a trailing `ORDER BY e_1, \dots, e_n` argument that defines row order:⁸

```
--          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'

⁸ $\oplus^{\text{string-agg}}$ essentially is `||` (string concatenation) which is not commutative.

Filtered and Unique Aggregates

```
SELECT agg(e) FILTER (WHERE p)
FROM ...
```

- **FILTER** clause alters aggregate semantics (see *):

$$\begin{array}{l} \vdots \\ x \leftarrow e(t) \\ \text{if } x \neq \text{NULL} \wedge p(x): \\ \vdots \end{array}$$

```
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_1, \dots, e_m	-- 5
FROM ...	-- 1
WHERE ...	-- 2
GROUP BY g_1, \dots, g_n	-- 3
HAVING p	-- 4

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

GROUP BY Partitions Rows

`SELECT ...`
`FROM ...`
`GROUP BY A_1`
`HAVING ...`

← evaluated once per **group** (not per row)

	A_1	A_2	...
	\vdots	\vdots	\vdots
the x_i group {	x_i	y_{i1}	\vdots
	x_i	y_{i2}	\vdots

the x_j group {	x_j	y_{j1}	\vdots
	x_j	y_{j2}	\vdots
	\vdots	\vdots	\vdots

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 $\text{bag}(\tau)$ ⁹

👍	👎		✖		👍
↓	↓		↓		↓
<pre>SELECT t.b, t.d FROM T AS t GROUP BY t.b</pre>		<pre>SELECT the(t.b) AS b, SUM(t.d) AS "Σd" FROM T AS t GROUP BY t.b</pre>			

- $t.d$ references current group of d values: violates 1NF!
 \Rightarrow 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 \Rightarrow SQL: using just $t.b$ is OK.
- (✖ May think of **hypothetical** aggregate $\text{the}(e)$ that picks one among equal e values.)

⁹ A view of **GROUP BY** that is due to Philip Wadler.

Aggregates are Evaluated Once Per Group

```

SELECT t.b                                AS "group",
       COUNT(*)                          AS size,
       SUM(t.d)                          AS "Σd",
       bool_and(t.a % 2 = 0)             AS "∀even(a)",
       string_agg(t.a :: text, ';' )    AS "all a"
FROM   T AS t
GROUP BY t.b;

```

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

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

Grouping Criteria

- The grouping criteria g_i 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 g_i are constant within a group (and thus can be used in `SELECT`).
 - If SQL does not know about the FD, explicitly add e to the set of g_i —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*):

q_1	UNION ALL	q_2	-- \cup^+ (bag union)
q_1	INTERSECT ALL	q_2	-- \cap^+ (bag intersection)
q_1	EXCEPT ALL	q_2	-- \setminus^+ (bag difference)

- Row types (width, field types) of the q_i must match.
- With **ALL**, row multiplicities are respected: if row r occurs n_i times in q_i , r will occur $\max(n_1 - n_2, 0)$ times in q_1 **EXCEPT ALL** q_2 (**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>	<u>species</u>
'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 \Rightarrow perform separate GROUP BYs on each relevant dimension:
- SQL syntactic sugar:


```
SELECT  $e_1, \dots, e_m$   
FROM  $T$   
GROUP BY GROUPING SETS ( $G_1, \dots, G_n$ )
```

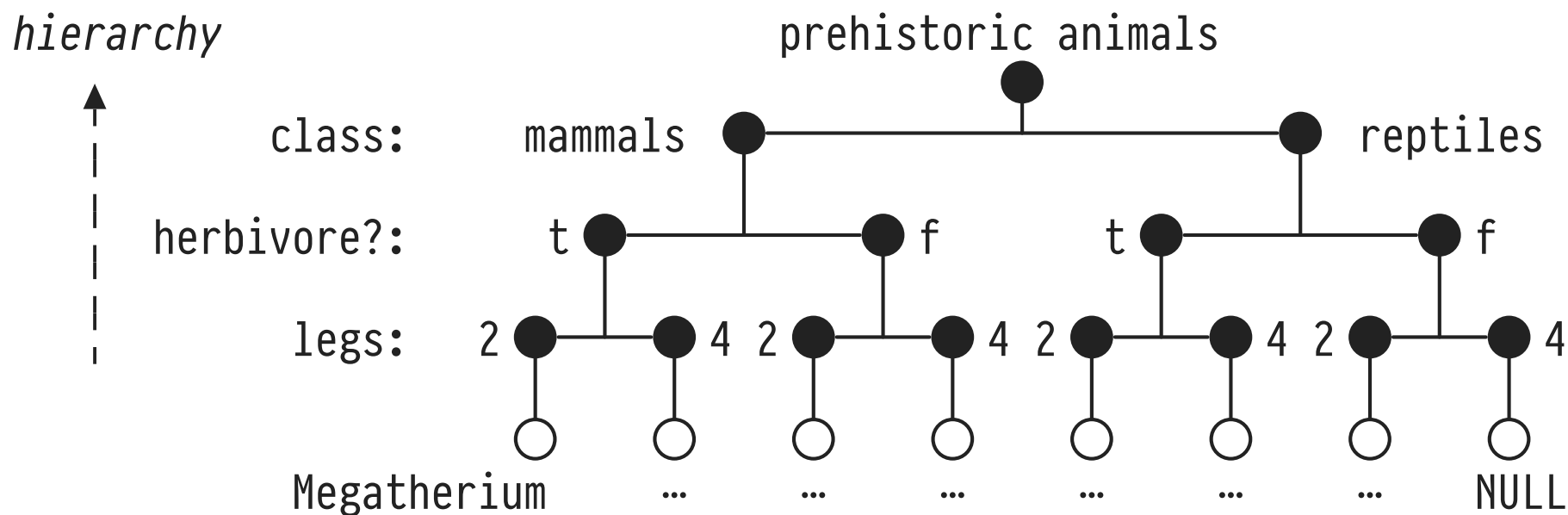
-- G_i : grouping criteria
-- sets in (...)

- Yields n GROUP BY queries q_i , glued together by UNION ALL. If non-aggregate $e_j \notin G_i$, $e_j \equiv \text{NULL}$ in q_i .

Hierarchical Dimensions: **ROLLUP**

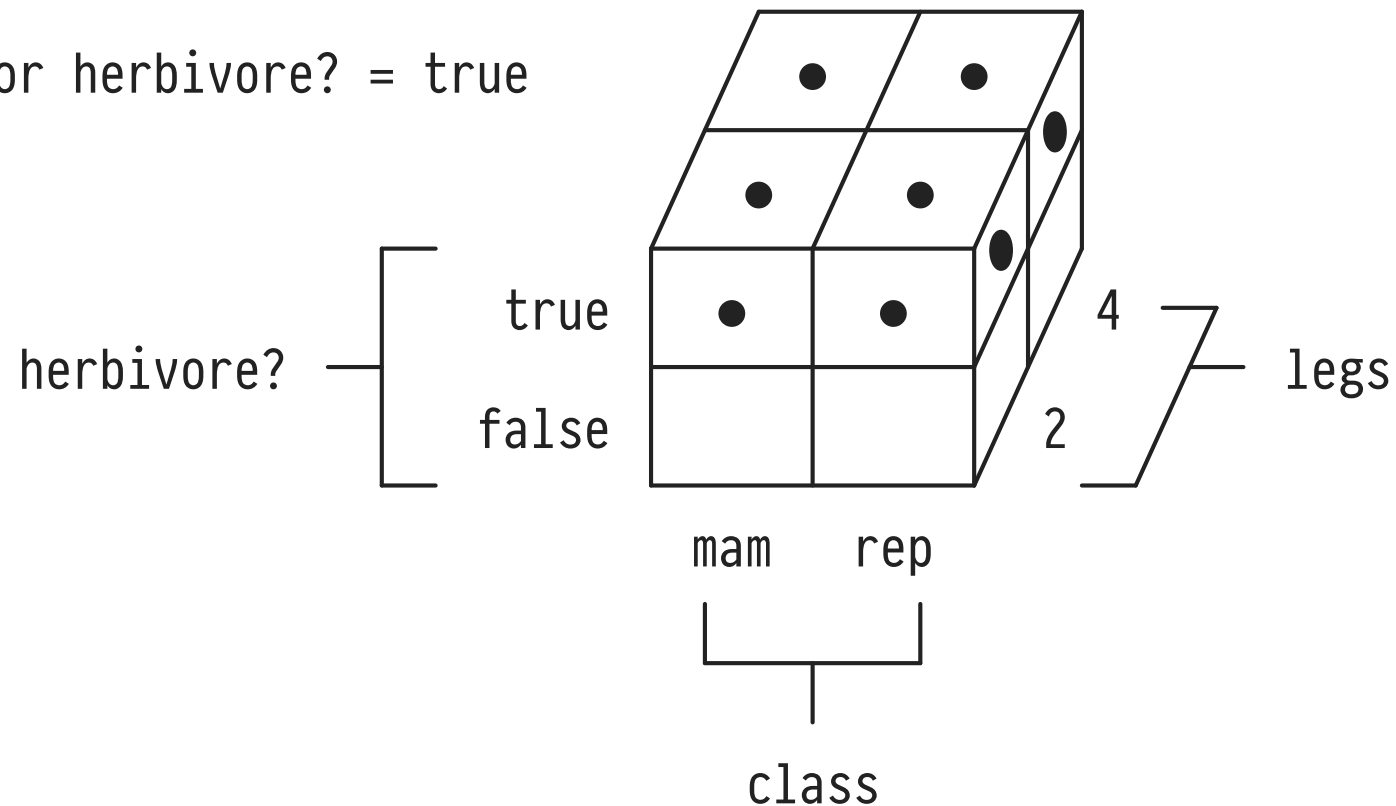
- **Group along a path** from any node G_n up to the root:

ROLLUP $(G_1, \dots, G_n) \equiv$ GROUPING SETS $((G_1, \dots, G_{n-1}, G_n)$
 $, (G_1, \dots, G_{n-1}), \dots$
 $, (G_1),$
 $, ())$  | *hierarchy*
↓



Analyze All Dimension Combinations: CUBE

● \equiv slice for herbivore? = true



CUBE $(G_1, \dots, G_n) \equiv$ GROUPING SETS $\left(\begin{array}{l} (G_1, \dots, G_n) \\ \vdots \\ () \end{array} \right) \left. \vphantom{\begin{array}{l} (G_1, \dots, G_n) \\ \vdots \\ () \end{array}} \right\} \text{all } 2^n \text{ subsets considered}$

14 : SQL Evaluation vs. Reading Order

```

SELECT DISTINCT ON (es 7) es 3, aggs 6
FROM   qs      1
WHERE  p        2
GROUP BY es     4
HAVING p        5

```

```

    UNION / EXCEPT / INTERSECT 8 } repeated 0 or more times,
    :                               } all evaluated before 9

```

```

ORDER BY es     9
OFFSET  n      10
LIMIT   n      10

```

- Reading order is: (7, 3, 6, 1 ⚠, 2, 4, 5, 8)⁺, 9, 10.

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:

WITH $T_1(c_{11}, \dots, c_{1, k_1}) \text{ AS } ($ $q_1),$ \vdots $T_n(c_{n1}, \dots, c_{n, k_n}) \text{ AS } ($ $q_n)$	}	Query q_i may refer to tables T_1, \dots, T_{i-1}
q		q may refer to all tables T_i

- “Literate SQL”: Reading and writing order coincide.
- Think of **let** $T_1 = q_1, \dots, T_n = q_n$ **in** q in your favorite FP language. The T_i are undefined outside **WITH**.

SQL With **WITH** — Sample Uses

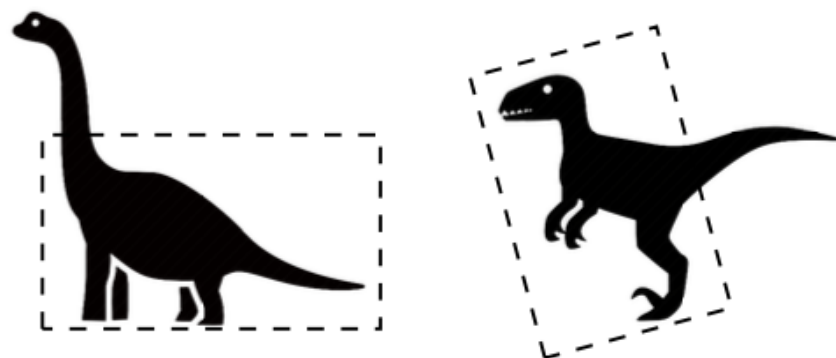
1. **Define queries in stages**, intermediate results in tables T_i . May use $q \equiv \text{TABLE } T_i^{10}$ to debug stage i .
2. **Bundle a query with test data:**

```
WITH
  prehistoric(class,"herbivore?",legs,species) AS (
    VALUES ('mammalia',true,2,'Megatherium'),
           ⋮
           ('reptilia',false,4,NULL)
  )
SELECT MAX(p.legs)
FROM   prehistoric AS p
```

¹⁰ Syntactic sugar for `SELECT t.* FROM T_i 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	?

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
Tyrannosaurus	7.0	15.2	2
Velociraptor	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

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	0.447
4	0.201

Transient Table **bodies**

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`'s shape the closest
 - Use the locomotion (column `legs`) in `b` to complete `d`, output completed `d`

¹¹ In this course, *query plan* refers to a “plan of attack” for a query problem, not `EXPLAIN` output.