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:

a	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 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             -- ❷ t is bound to current row
FROM   T AS t        -- ❶ bind/introduce t
```

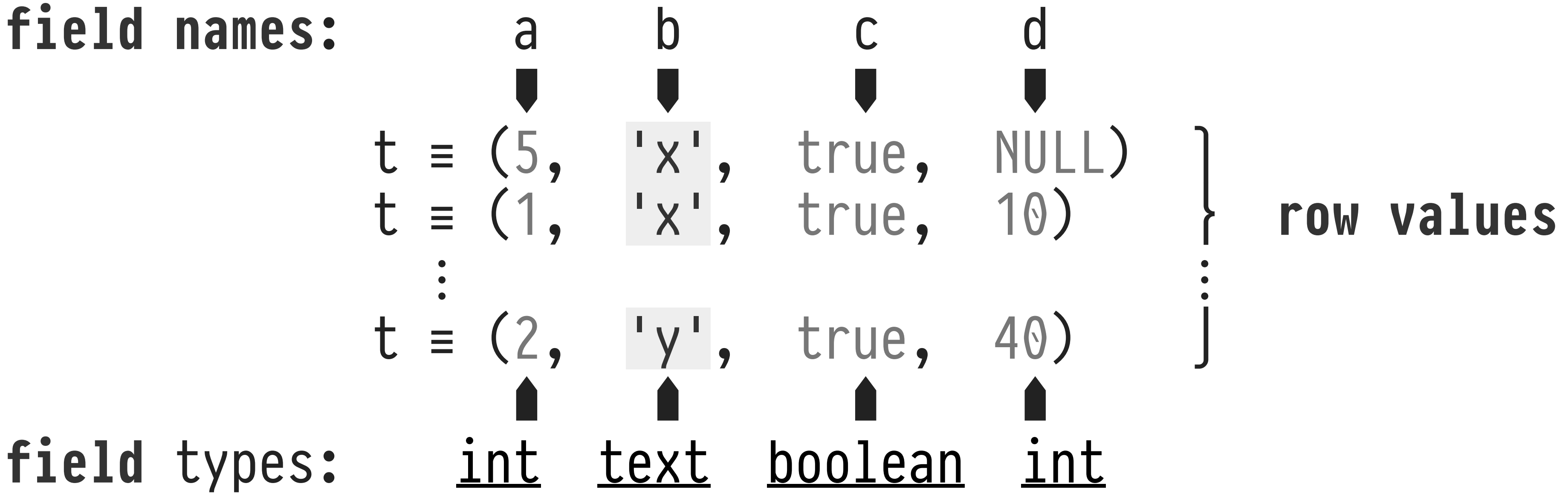
- If you omit AS t in the FROM clause, a row variable T (generally: AS ‹table name›) will be implicitly introduced.

- This course: always explicitly introduce/name row variables for disambiguation, clarity, readability.

Row Values

```
SELECT t          -- ❷ t is bound to current row
FROM   T AS t     -- ❶ 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)   }
              t ≡ (1,  'x',   true,   10)     }  row values
              ⋮                               ⋮
              t ≡ (2,  'y',   true,   40)     }

field types:     int  text  boolean  int
```

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 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.[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:

  - $t_1 = t_2 \iff t_1.a = t_2.a$ AND ⋯ AND $t_1.d = t_2.d$
  - $t_1 < t_2 \iff$
$t_1.a < t_2.a$ OR ($t_1.a = t_2.a$ AND $t_1.b < t_2.b$) OR ⋯

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

  Assume the binding $t \equiv$ (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) ⇒ ambigiuity ☹.
- This course: explicitly use AS to name columns unless a name can be derived from $\langle e_i \rangle$ (e.g., as in $\langle e_i \rangle$ ≡ t.a).
- If column/table names are case-sensitive or contain whitespace/symbols/keywords: wrap in quotes "$\langle c_i \rangle$".

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:

```sql
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 ‹e₁›, ..., ‹eₙ›
```

- 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 $\langle T_i \rangle$ and successively binds the row variables $\langle t_i \rangle$ to the tables' rows:

```
SELECT ...                                      -- 2
FROM   ‹T1› AS ‹t1›, ..., ‹Tn› AS ‹tn›          -- 1
```

- The $\langle T_i \rangle$ may be table names or SQL queries computing tables (in (⋯)).

- 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 ☹) columns:

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

FROM Computes Cartesian Products

```
SELECT ...
FROM   ‹T1› AS ‹t1›, ..., ‹Tn› AS ‹tn›
```

- 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 ‹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› = true are passed on.[3]
- Absence of a WHERE clause is interpreted as WHERE true.

[3] If ‹p› evaluates to NULL ≠ 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 □ irrelevant) may be **used in place of an atomic value** *v*:

□
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
- 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** $(v_1, \ldots, v_n)$ with field names $\langle c_i \rangle$:

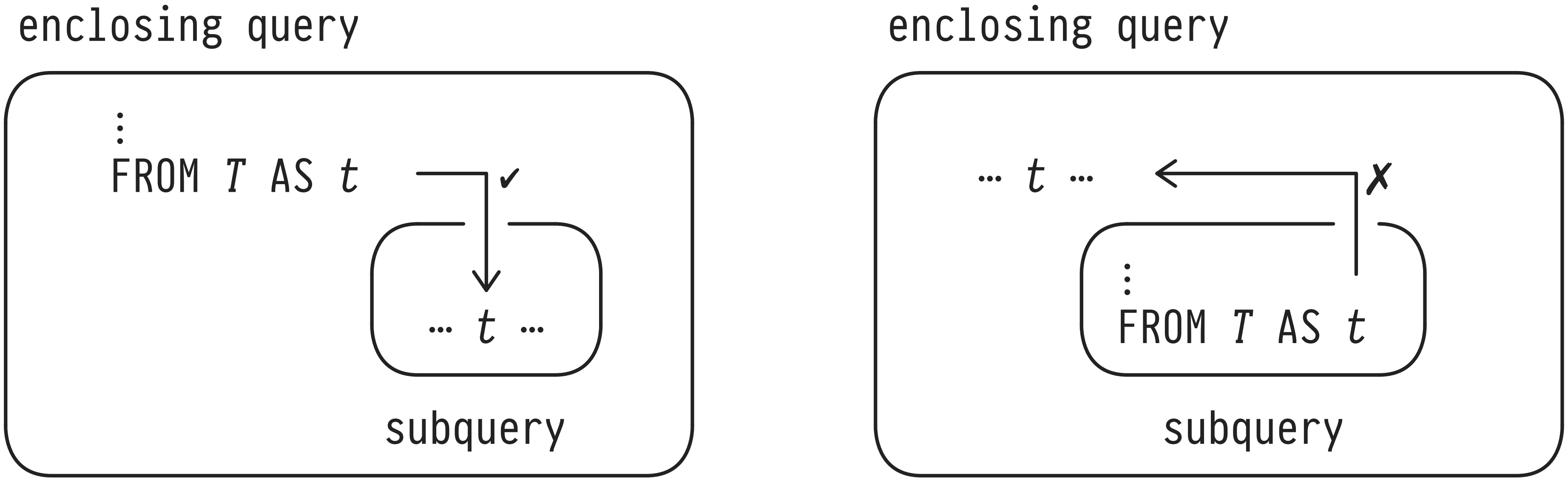
$\langle c_1 \rangle$	…	$\langle c_n \rangle$
v_1	…	v_n

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.*                                   evaluated 5 times
FROM   T AS t1                                under t1 bindings:
WHERE  t1.b <> (SELECT t2.b                ⎫     t1 ≡ (1, …)
                FROM   T AS t2             ⎬     t1 ≡ (2, …)
                WHERE  t1.a = t2.a)        ⎭     t1 ≡ (3, …)
                       ▲                         t1 ≡ (4, …)
                      free                       t1 ≡ (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 ...                        -- ❸
FROM   ...                        -- ❶
WHERE  ...                        -- ❷
ORDER BY ‹e_1›, ..., ‹e_n› -- ❹
```

- 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 $\langle e_i \rangle$ are expressions that may refer to column names in the SELECT clause.

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

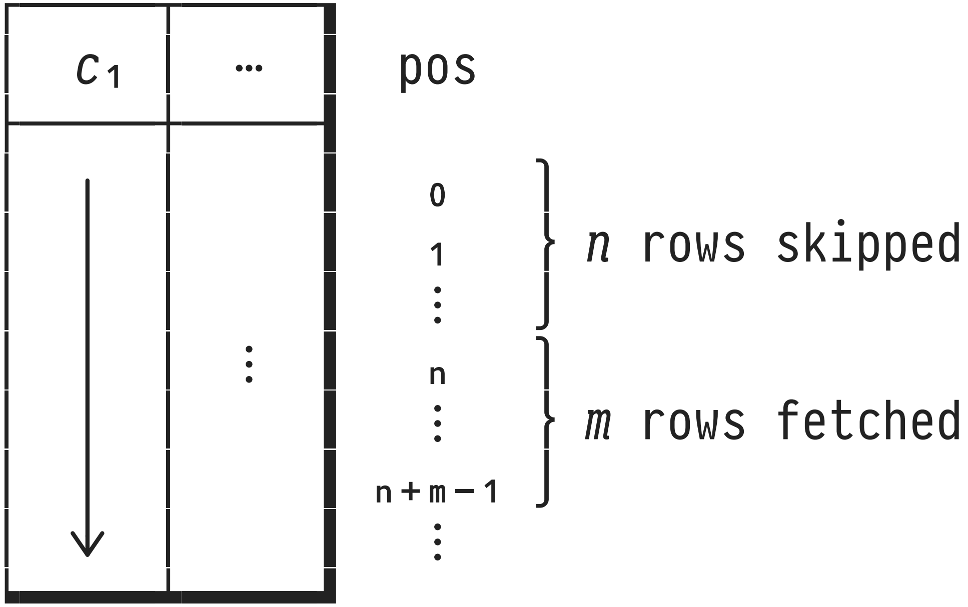
ORDER BY establishes a well-defined row order that is **local** to the current (sub)query:

```
 may yield rows in any order
   ↓
SELECT t1.*
FROM   (SELECT t2.*                 ⎫  guaranteed row order
        FROM   T AS t2              ⎬  inside the subquery only
        ORDER BY t2.a) AS t1;       ⎭
```

- ⚠️ 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."*



```
⋮
ORDER BY c₁
OFFSET ‹n›
LIMIT ‹m›
```

- 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:**

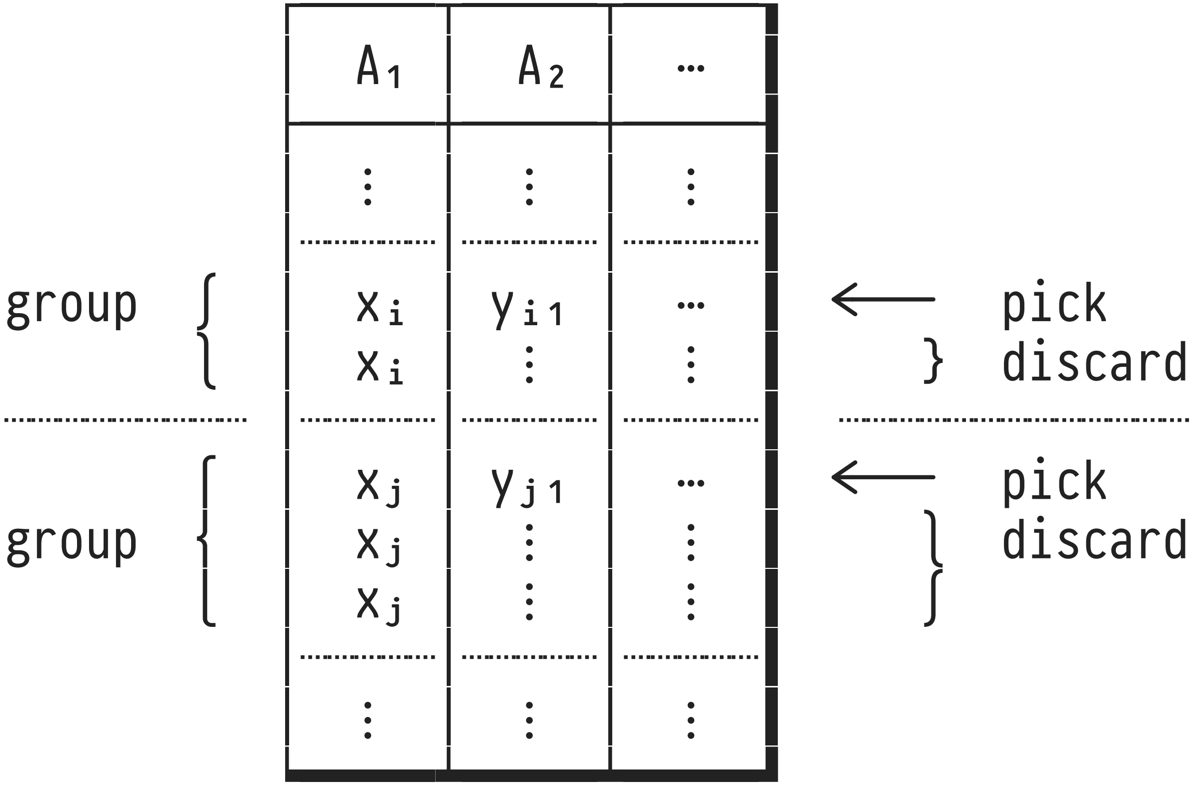
```
                     prefix of ORDER BY clause
                    ⏞
SELECT DISTINCT ON ❹ (‹e_1›,…,‹e_n›) ‹c_1›,…,‹c_k›    -- ②
FROM     …                                             -- ①
ORDER BY ‹e_1›,…,‹e_n›,‹e_{n+1}›,…,‹e_m›              -- ③
```

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.

- ⚠️ Without ORDER BY, step 3 picks *any* row in each group.

DISTINCT ON: Group, Then Pick First in Each Group

```
SELECT DISTINCT ON (A1) ...   -- For each A1, pick the row …
FROM   ...
ORDER BY A1, A2 DESC          -- … with the largest A2
```



DISTINCT: Table-Wide Duplicate Removal

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

```
SELECT DISTINCT ❸ ‹c₁›,...,‹cₖ›   -- ❷
FROM   ...                        -- ❶
```

- 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 (‹c_1›,...,‹c_k›) ‹c_1›,...,‹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* ≡ entire table:

SELECT ‹agg_1›(‹e_1›) **AS** ‹c_1›, …, ‹agg_n›(‹e_n›) **AS** ‹c_n›
FROM …

- Reduction of input rows: result table will have **one row.**

- Cannot mix aggregates with non-aggregate expression ‹e› in 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 (ϕ^{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

```
$a \leftarrow \phi^{agg}$                 -- $a$ will be aggregate value
for $t$ in $T$                            -- iterate over all rows of $T$
    $x \leftarrow e(t)$                   --  value to be aggregated
    if $x \neq$ NULL                      --   aggregates ignore NULL values (⁑)
        if $a = \phi^{agg}$               --   once we see first non-NULL value:
            $a \leftarrow z^{agg}$        --     initialize aggregate
        $a \leftarrow \oplus^{agg}(a, x)$ --     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	‹0, 0›	$\langle a.1 + x, a.2 + 1\rangle$
MAX	NULL	$-\infty$	$\mathrm{max}_2(a, x)$
MIN	NULL	$+\infty$	$\mathrm{min}_2(a, x)$
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).

[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 "∑d",
       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	∑d	max(b)	∀c	∃d=30
5	4	100	'y'	false	true

‹*p*› ≡ true

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

‹*p*› ≡ false

Ordered Aggregates

- For most aggregates *agg*, $\oplus^{agg}$ is commutative (and associative): row order does not matter.

- **Order-sensitive aggregates** admit a trailing ORDER BY $\langle e_1 \rangle, \ldots, \langle e_n \rangle$ 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] $\oplus^{string\text{-}agg}$ 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 ⁑):

```
⋮
$x \leftarrow e(t)$
if $x \neq$ NULL $\land$ $p(x)$:
⋮
```

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

- All rows that agree on all expressions $\langle g_i \rangle$ (the *set* of **grouping criteria**) form one **group**.

- ⇒ Steps ❹ and ❺ process groups (*not* individual rows). This affects expressions $\langle p \rangle$ and the $\langle e_j \rangle$.

GROUP BY Partitions Rows

```
SELECT ...
FROM   ...
GROUP BY A₁
HAVING ...
```

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

	A_1	A_2	…
	⋮	⋮	⋮
the x_i group	x_i	y_{i1}	⋮
	x_i	y_{i2}	⋮
the x_j group	x_j	y_{j1}	⋮
	x_j	y_{j2}	⋮
	⋮	⋮	⋮

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)[9]

```
        👍   👎                          ⁂              👍
        ▼    ▼                           ▼              ▼
SELECT t.b, t.d   ┆  SELECT the(t.b) AS b, SUM(t.d) AS "Σd"
FROM    T AS t    ┆  FROM    T AS t
GROUP BY t.b      ┆  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.)

[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 "∑d",
       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	∑d	∀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 $\langle g_i \rangle$ form a set—order is irrelevant.

- Grouping on a **key** effectively puts each row in its own singleton group. (Typically a query smell 💩.)

- Expressions $\langle e \rangle$ that are **functionally dependent** on the $\langle g_i \rangle$ are constant within a group (and thus can be used in SELECT).

  - 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** $\langle q_2 \rangle$ -- $\cup^+$ (bag union)
$\langle q_1 \rangle$ **INTERSECT ALL** $\langle q_2 \rangle$ -- $\cap^+$ (bag intersection)
$\langle q_1 \rangle$ **EXCEPT ALL** $\langle q_2 \rangle$ -- $\setminus^+$ (bag difference)

- Row types (width, field types) of the $\langle q_i \rangle$ 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:

class	herbivore?	legs	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 ‹e₁›, ..., ‹eₘ›
FROM   ‹T›                                  -- Gᵢ: grouping criteria
GROUP BY GROUPING SETS (G₁,...,Gₙ)          --     sets in (…)
```

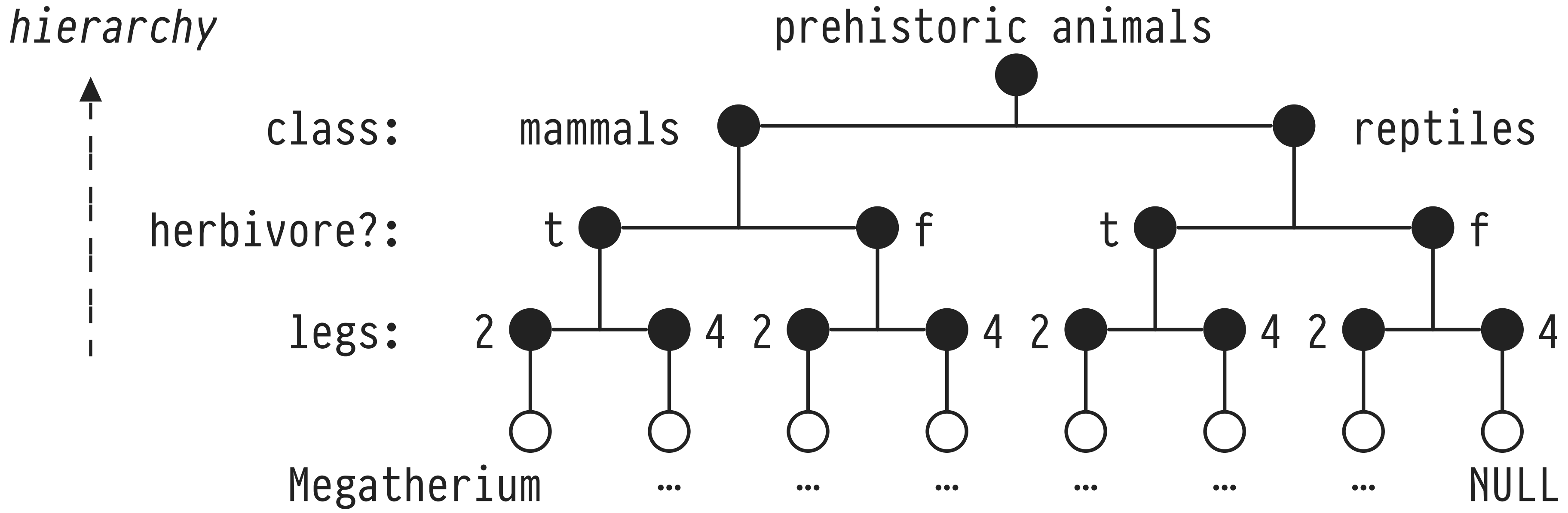
- Yields n GROUP BY queries q_i, glued together by UNION ALL. If non-aggregate ‹e_j› $\notin G_i$, ‹e_j› ≡ NULL in q_i.

Hierarchical Dimensions: ROLLUP

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

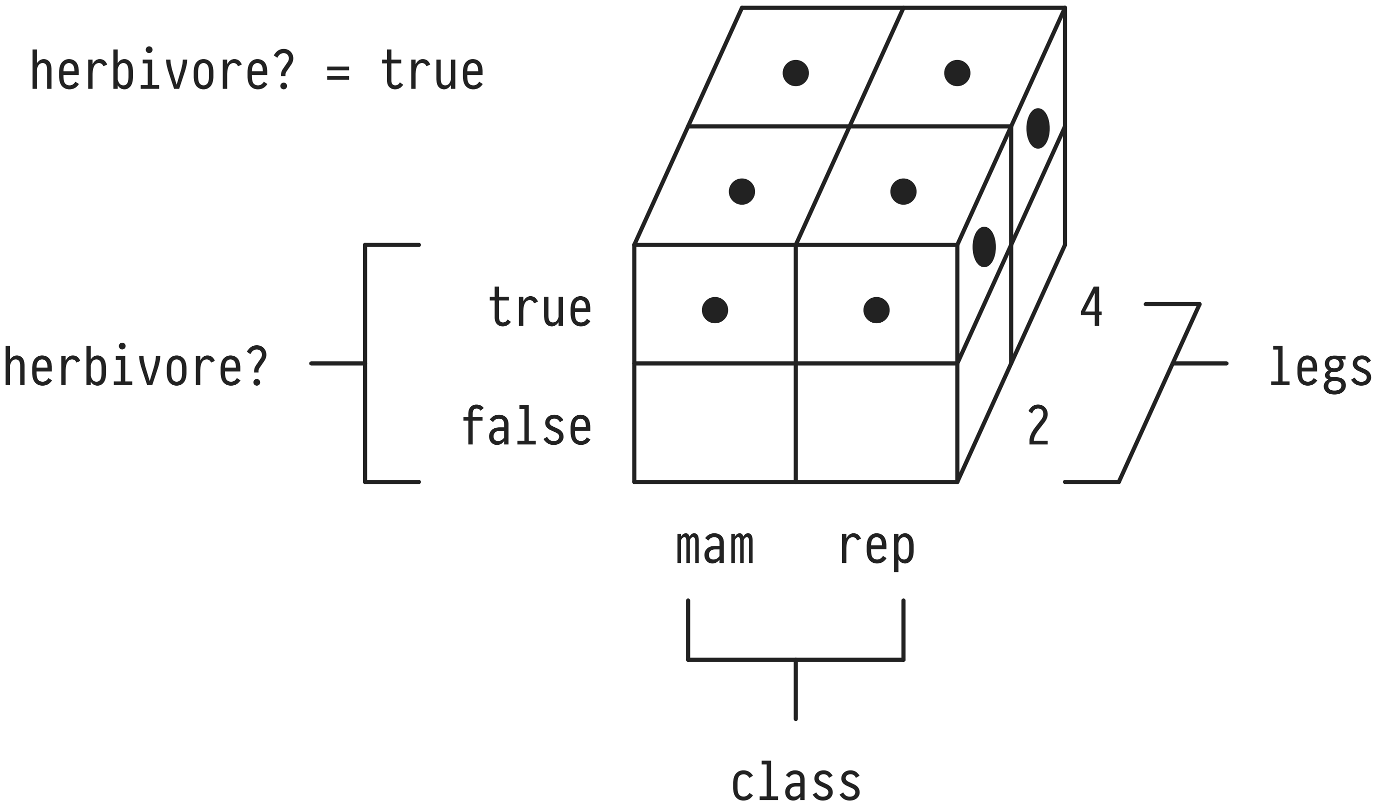
$$\begin{aligned}\text{ROLLUP }(G_1,\ldots,G_n) \equiv \text{GROUPING SETS }(&(G_1,\ldots,G_{n-1},G_n)\\ ,&(G_1,\ldots,G_{n-1}),\ \ldots\\ ,&(G_1),\\ ,&())\end{aligned}$$ ⚠

hierarchy



Analyze All Dimension Combinations: CUBE

- slice for herbivore? = true



$$\texttt{CUBE}\ (G_1,\ldots,G_n) \equiv \texttt{GROUPING SETS}\ \left.\begin{array}{l}((G_1,\ldots,G_n)\\ \quad ,\ \vdots\\ \quad ,())\end{array}\right\}$$

all 2^n subsets considered

14 ¦ SQL Evaluation vs. Reading Order

```
SELECT DISTINCT ON (‹es› ❼) ‹es› ❸, ‹aggs› ❻
FROM    ‹qs›    ❶
WHERE   ‹p›     ❷
GROUP BY ‹es›   ❹
HAVING ‹p›      ❺

  UNION / EXCEPT / INTERSECT ❽  } repeated 0 or more times,
  ⋮                             } all evaluated before ❾

ORDER BY ‹es›   ❾
OFFSET ‹n›      ❿
LIMIT  ‹n›      ❿
```

- Reading order is: (❼,❸,❻,❶⚠,❷,❹,❺,❽)$^+$,❾,❿.

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:

$$
\begin{array}{ll}
\textbf{WITH} & \\
\left.\begin{array}{l}
\quad\langle T_1\rangle(\langle c_{1\,1}\rangle,\ldots,\langle c_{1,k1}\rangle)\ \textbf{AS}\ (\\
\qquad\langle q_1\rangle\ ),\\
\quad\vdots\\
\quad\langle T_n\rangle(\langle c_{n\,1}\rangle,\ldots,\langle c_{n,kn}\rangle)\ \textbf{AS}\ (\\
\qquad\langle q_n\rangle\ )
\end{array}\right\} & \begin{array}{l}\text{Query } \langle q_i\rangle \text{ may refer to}\\ \text{tables } \langle T_1\rangle,\ldots,\langle T_{i-1}\rangle\end{array}\\
\langle q\rangle & \langle q\rangle\text{ may refer to all }\langle T_i\rangle
\end{array}
$$

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

SQL With WITH — Sample Uses

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.

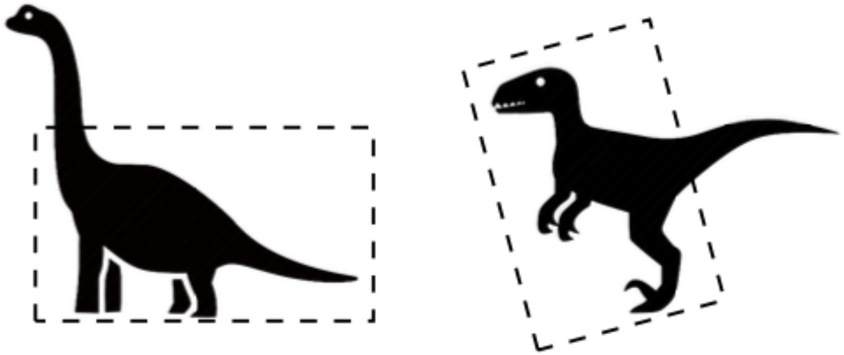
2. **Bundle a query with test data:**

```
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 $\langle T_i \rangle$ 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:

species	height	length	legs
Gallimimus	2.4	5.5	?

Dinosaur Body Shapes

species	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
)
⋮
```

legs	shape
2	0.201
4	0.447

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* the closest
       - Use the locomotion 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.