Advanced SQL

② The Core of SQL

Winter 2025/26

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 *TaDa* (Chapters 6 to 8)
  - the DuckDB web (Documentation › SQL)

- 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 of T
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          -- 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 ≡ {a:5,   b:'x',   c:true,  d:NULL} ⎫
       t ≡ {a:1,   b:'x',   c:true,  d:10}   ⎬ row values
       ⋮                                     ⋮ (structs)
       t ≡ {a:2,   b:'y',   c:true,  d:40}   ⎭
               ↑      ↑          ↑       ↑
field types:  int    text    boolean   int
```

Row Types (Struct Types)

- All rows `t` in table `T` have **row type** `row(a int, b text, c boolean, d int)`[1]. Abbreviated as `t :: row(...)`. Read symbol `::` as “*has type*.”

- A row type `τ` can also be explicitly defined via

```
CREATE TYPE τ AS row(a int, b text, c boolean, d int);
```

- New row type `τ` is then usable like any builtin type:

```
TABLE duckdb_types;      -- inspect DuckDB's catalog
```

[1] In DuckDB, `struct` is a synonym for `row`: `struct(a int, b text, c boolean, d int)`.

Row Field Access and * ("Star")

- Named **field access** uses dot notation. Assume `t :: τ` and binding `t ≡ {a:5, b:'x', c:true, d: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. DuckDB adds lots of syntatic sugar when it comes to `*` in the `SELECT` clause, see Star Expression in the DuckDB documentation.

Row Comparisons

- **Row comparisons** between rows t_1, t_2 are performed field-by-field and lexicographically (provided that the field types match). Assume $t_1 :: \tau$, $t_2 :: \tau$:

  - $t_1 = t_2 \iff t_1.a = t_2.a$ 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 …

- Thus (here: $\tau \equiv$ `row(a int, b text)`):

  `{a:1, b:'z'} < {a:2, b:'x'}` and
  `{a:2, b:'z'} > {a:2, b:'x'}`.

3 The SELECT Clause

A SELECT **clause** evaluates n expressions e_1, …, e_n:

```
SELECT e₁ AS c₁, …, eₙ AS cₙ
```

- Creates n columns named c_1, …, c_n (case-insensitive).
- In absence of AS c_i, DuckDB derives a column name from e_i. Almost always this is questionable practice 😦.
- Thus: explicitly use AS to name columns unless a name can be sensibly derived from e_i (e.g., when $e_i \equiv$ t.a).
- If a column/table names contains 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** n **provided rows:**

```
VALUES (e11,...,e1k), ..., (en1,...,enk)
```

- If $n > 1$, the rows must agree in arity k and field types (first row value is used to infer and determine types).
- VALUES automatically assigns column names "col<*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   T₁ AS t₁, …, Tₙ AS tₙ        -- 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:

$$T_i \text{ AS } t_i(\underline{c_{i1}, \ldots, c_{ik}})$$

FROM Computes Cartesian Products

```
SELECT …
FROM   T₁ AS t₁, …, Tₙ AS tₙ
```

- This FROM clause generates $|T_1| \times \cdots \times |T_n|$ bindings. Semantics: compute the **Cartesian product** $T_1 \times \cdots \times T_n$, 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}, …, 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   T₁ AS t₁, …, Tₙ AS tₙ      -- 1
WHERE  p                          -- 2
```

- All row variables t_i are in scope in predicate 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 columns yields a **compile-time error**,
- ... a table with > 1 rows yields an **error at runtime.** ⚠[4]

[4] With DuckDB's configuration option scalar_subquery_error_on_multiple_rows set to false, a *random* value v will be selected from the table. This merely hides a potential source of errors. Avoid.

Scalar Subqueries: Atomic Values

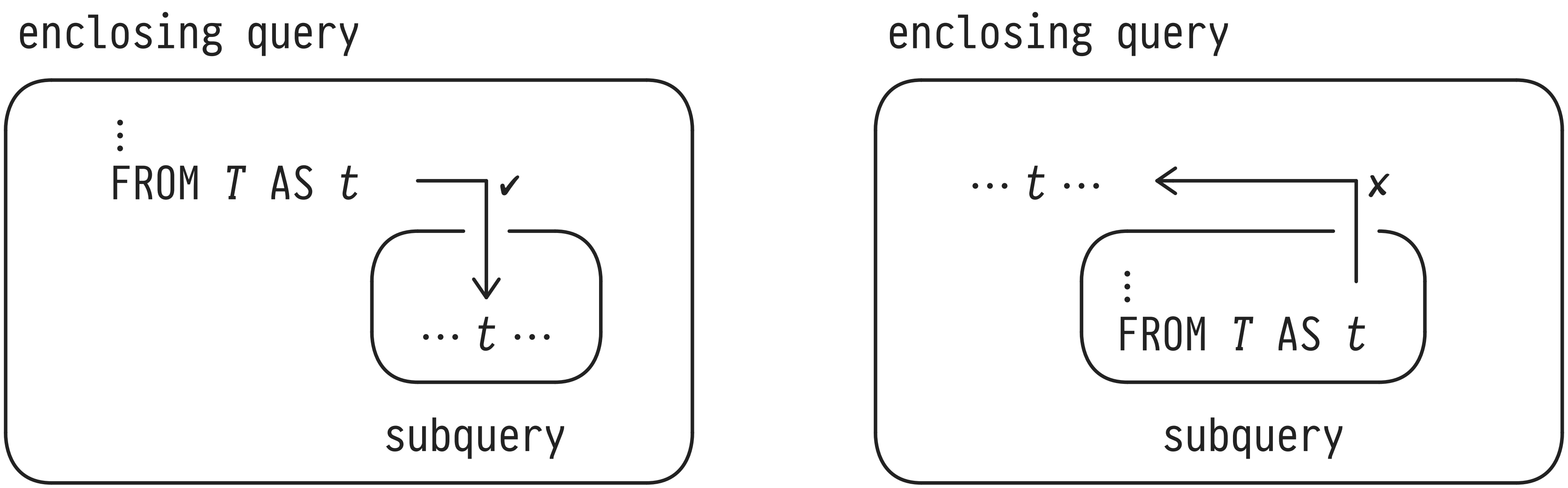
```
         generate single column
                    ↓
SELECT 2 + (SELECT t.d AS _
            FROM   T AS t
            WHERE  t.a = 2)  AS "The Answer"

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

- **Compile-time error** for `SELECT t.a, t.d` (> 1 column)
- **Runtime error** for `WHERE t.a > 2` (> 1 row)
- 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.

Row Variable Scoping

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

```
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 …                 -- 3
FROM   …                 -- 1
WHERE  …                 -- 2
ORDER BY e1, …, en       -- 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:

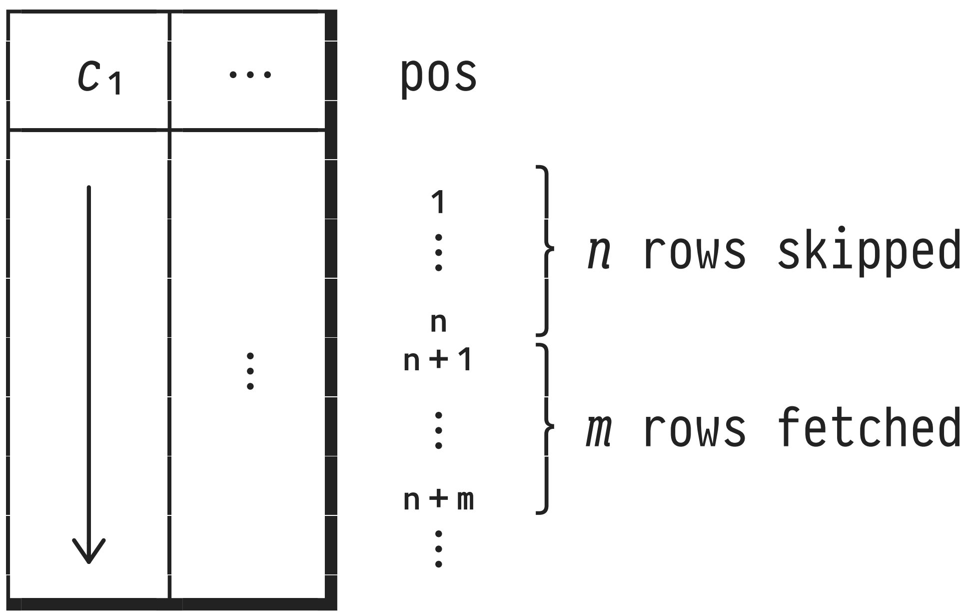
```
 may yield rows in any order
   ↓
SELECT t1.*
FROM   (SELECT t2.*                     guaranteed row order
        FROM   T AS t2                  inside the subquery only
        ORDER BY t2.a) AS t1;
```

- ⚠️ 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 LIMIT 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:**

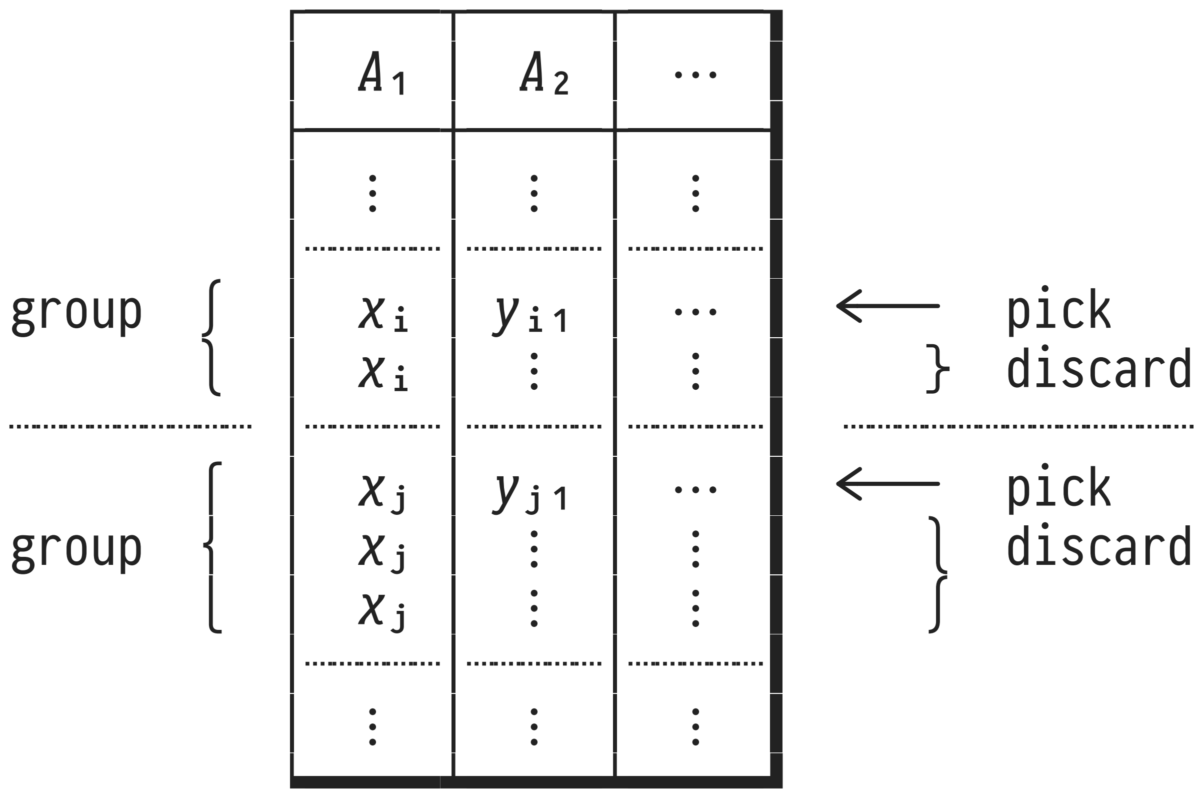
```
                     prefix of ORDER BY clause
SELECT DISTINCT ON ❹ (e₁, …, eₙ) c₁, …, cₖ          -- ❷
FROM   …                                             -- ❶
ORDER BY e₁, …, eₙ, eₙ₊₁, …, eₘ                      -- ❸
```

1. Sort rows in $e_1, \ldots, e_n, e_{n+1}, \ldots, e_m$ order.
2. Rows with identical $e_1, \ldots, e_n$ values form one **group.**
3. From each of these groups, pick **the first row** in $e_{n+1}, \ldots, e_m$ order.

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

DISTINCT ON: Group, Then Pick First in Each Group

```
SELECT DISTINCT ON (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.[5]
- 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.

[5] 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 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:[6] which value of *e* should we pick?

[6] 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 $(\emptyset^{agg}, z^{agg}, \oplus^{agg})$:
  - $\emptyset^{agg}$ (*empty*): aggregate of the empty value collection
  - z^{agg} (*zero*): aggregate value initialiser
  - $\oplus^{agg}$ (*merge*): add value to existing aggregate

```
a ← ∅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 = ∅agg            --   once we see first non-NULL value:
│ │ └ a ← zagg             --     initialize aggregate
└ └ a ← ⊕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[7]	0	$a + x$
avg[8]	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	$a \vee x$
⋮	⋮	⋮	⋮

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

[7] 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.

[8] 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

$p \equiv$ true

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

$p \equiv$ 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` $e_1, \ldots, e_n$ argument that defines row order:[9]

```
--                  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'

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

Filtered and Unique Aggregates

```
SELECT agg(e) FILTER (p)  -- or: … FILTER (WHERE p)
FROM   …
```

- FILTER clause alters aggregate semantics (see *):

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

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

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

Evaluating Expressions on Minimal/Maximal Rows

In math, **arg max** finds the *argument* x_m that maximises function f:

$$x_m = \operatorname*{arg\,max}_{x \in S} f(x) \iff f(x_m) = \max_{x \in S} f(x)$$

- In SQL, aggregate arg_max (arg_min) evaluates expression e_1 for the row with maximal (minimal) e_2:

```
SELECT arg_max(e₁, e₂)
FROM   ...
```

  - **Q:** Convenient but not essential. Can you simulate arg_max using only SQL constructs introduced so far?

11 ¦ Forming Groups of Rows

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

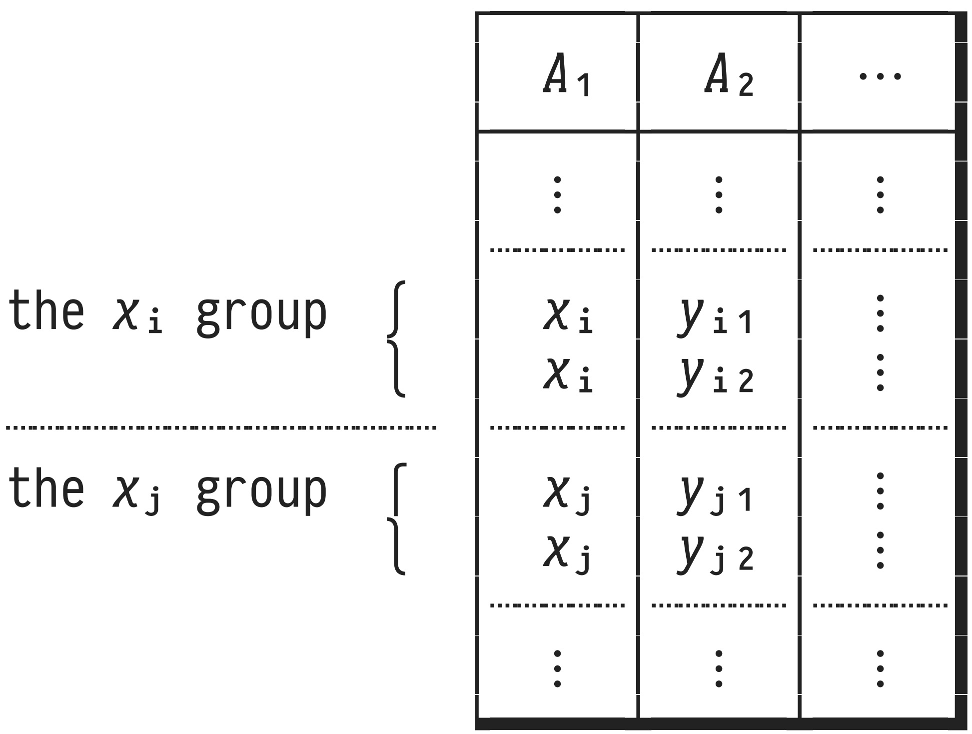
```
SELECT e₁, …, eₘ         -- 5
FROM   …                 -- 1
WHERE  …                 -- 2
GROUP BY g₁, …, gₙ       -- 3
HAVING p                 -- 4
```

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

GROUP BY Partitions Rows

```
SELECT   …
FROM     …
GROUP BY A1
HAVING   …
```

SELECT … and HAVING …: evaluated once per **group** (**not** per **row**)



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)

```
       ✓    ✗                  ⁂                 ✓
       ↓    ↓                  ↓                 ↓
SELECT t.b, t.d ┊ SELECT any_value(t.b) AS b, sum(t.d) AS "∑d"
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!
  $\Rightarrow$ 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.
- (⁂ aggregate any_value(*e*) picks one arbitrary value among the equal *e* values.[10])

[10] Indeed, in DuckDB: any_value(*e*) ≡ arbitrary(*e*).

Aggregates Return One Result 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	∑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 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 *could* be used in SELECT).

  Since SQL does not know about the FD, either

  1. explicitly add e to the set of g_i (this will *not* affect the grouping), or
  2. use any_value(e).

12 ┆ Bag and Set Operations

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

```
q₁ UNION ALL     q₂     -- ∪⁺ (bag union)
q₁ INTERSECT ALL q₂     -- ∩⁺ (bag intersection)
q₁ EXCEPT ALL    q₂     -- \⁺ (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).

- 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 (…)
```

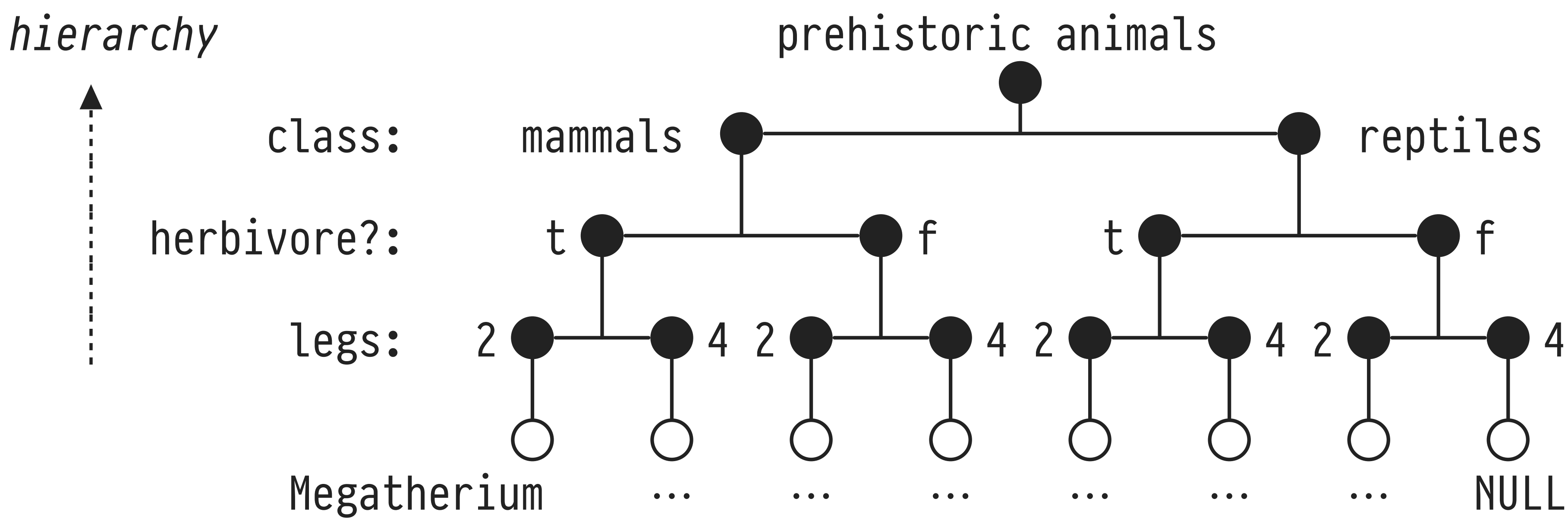
- Yields n GROUP BY queries q_i, glued together by UNION ALL. If non-aggregate $e_j \notin G_i$, $e_j \equiv$ NULL in q_i.

Hierarchical Dimensions: ROLLUP

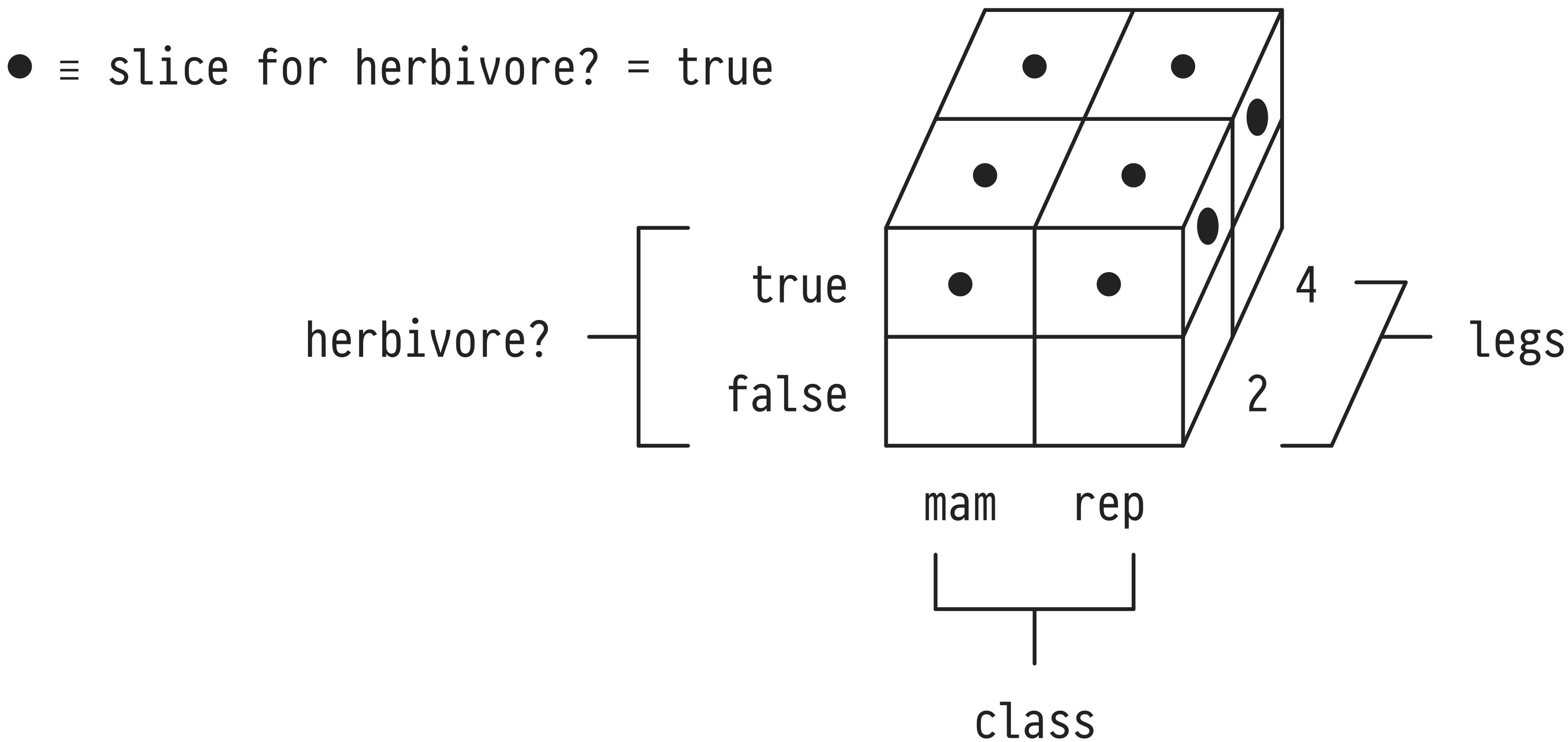
- **Group along a path** from any node G_n up to the root:

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

hierarchy



Analyze All Dimension Combinations: CUBE



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

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.
- (DuckDB's “friendly SQL”: admit FROM before SELECT.)

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
 T₁(c₁₁,…,c₁,ₖ₁) AS [NOT MATERIALIZED] --  ⎫
   (q₁),                                --  ⎪ Query qᵢ may
   ⋮                                    --  ⎬ refer to tables
 Tₙ(cₙ₁,…,cₙ,ₖₙ) AS [NOT MATERIALIZED] --  ⎪ T₁,…,Tᵢ₋₁
   (qₙ)                                 --  ⎭
 q                           -- q may refer to all tables Tᵢ
```

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

SQL With WITH — Sample Uses

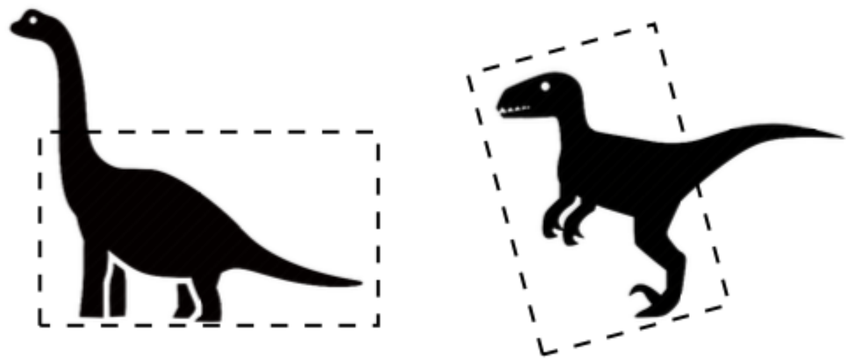
1. **Define queries in stages,** intermediate results in tables T_i. May use $q \equiv$ TABLE T_i to debug stage i.
2. MATERIALIZED: **avoid recomputation** of common subqueries.[11]
3. **Bundle a query with test data:**

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

[11] DuckDB performs materialization by default. Modifier NOT MATERIALIZED will *inline* q_i at its use sites (if q_i is “simple” and inlining is semantically sound).

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	0.447
4	0.201

Transient Table bodies

Dinosaur Body Shapes

- **Query Plan:**[12]

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*

[12] In this course, *query plan* refers to a "plan of attack" for a query problem, not `EXPLAIN` output.