INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen

Summer 2023

SQL: GROUPING AND AGGREGATION

- In relational database design, a **single** table like turtles will typically hold information about **multiple** individual mini-world objects.
  - Column turtle is used to divide the lists of drawing commands into disjoint **groups:**

turtles

turtle	pos	command
τ_1	1	(t,0,10)
τ_1	2	(t,10,0)
τ_1	3	(t,0,-10)
τ_1	4	(t,-10,0)
τ_2	1	(t,5,10)
τ_2	2	(t,5,-10)
τ_2	3	(t,-10,0)
τ_3	1	(t,0,10)
τ_3	2	(f,-5,-5)
τ_3	3	(t,10,0)

SQL: GROUPING AND AGGREGATION

- The SQL SELECT–FROM–WHERE block supports a language construct, GROUP BY, that enables queries to perform **operations per row group.**

GROUP BY

The optional **GROUP BY clause** in a SELECT–FROM–WHERE block condenses into a single row all those rows that share the same value for the listed *grouping expressions*:

```
<SELECT FROM WHERE block>
[ GROUP BY expression [, …] ]
```

The order of the grouping *expressions* is irrelevant.

GROUP BY is performed after row filtering (WHERE) but before result generation: the SELECT clause will thus output row groups, *not* individual rows.

SQL: GROUPING AND AGGREGATION

```
SELECT t.turtle, t.pos, t.command     -- ⚠ SELECT clause not valid SQL – see below
for a fix
FROM   turtles AS t
GROUP BY t.turtle
```

turtles

turtle	pos	command
τ_1	[1, 2, 3, 4]	[(t,0,10), (t,10,0), (t,0,-10), (t,-10,0)]
τ_2	[1, 2, 3]	[(t,5,10), (t,5,-10), (t,-10,0)]
τ_3	[1, 2, 3]	[(t,0,10), (f,-5,-5), (t,10,0)]

- Produces as many groups as there are distinct values for the grouping expressions (one group for each of the values τ_1, τ_2, τ_3 of expresssion t.turtle).
- SELECT: value of grouping expression t.turtle is unambigiuous in each group. **Not so for the non-grouping columns** pos and command.
  - Note: in a sense, after GROUP BY the types of all non-grouping columns change from *t* to *bag(t)*. Problematic for the flat relational model.

SQL: GROUPING AND AGGREGATION

SELECT after GROUP BY / Aggregrate Functions

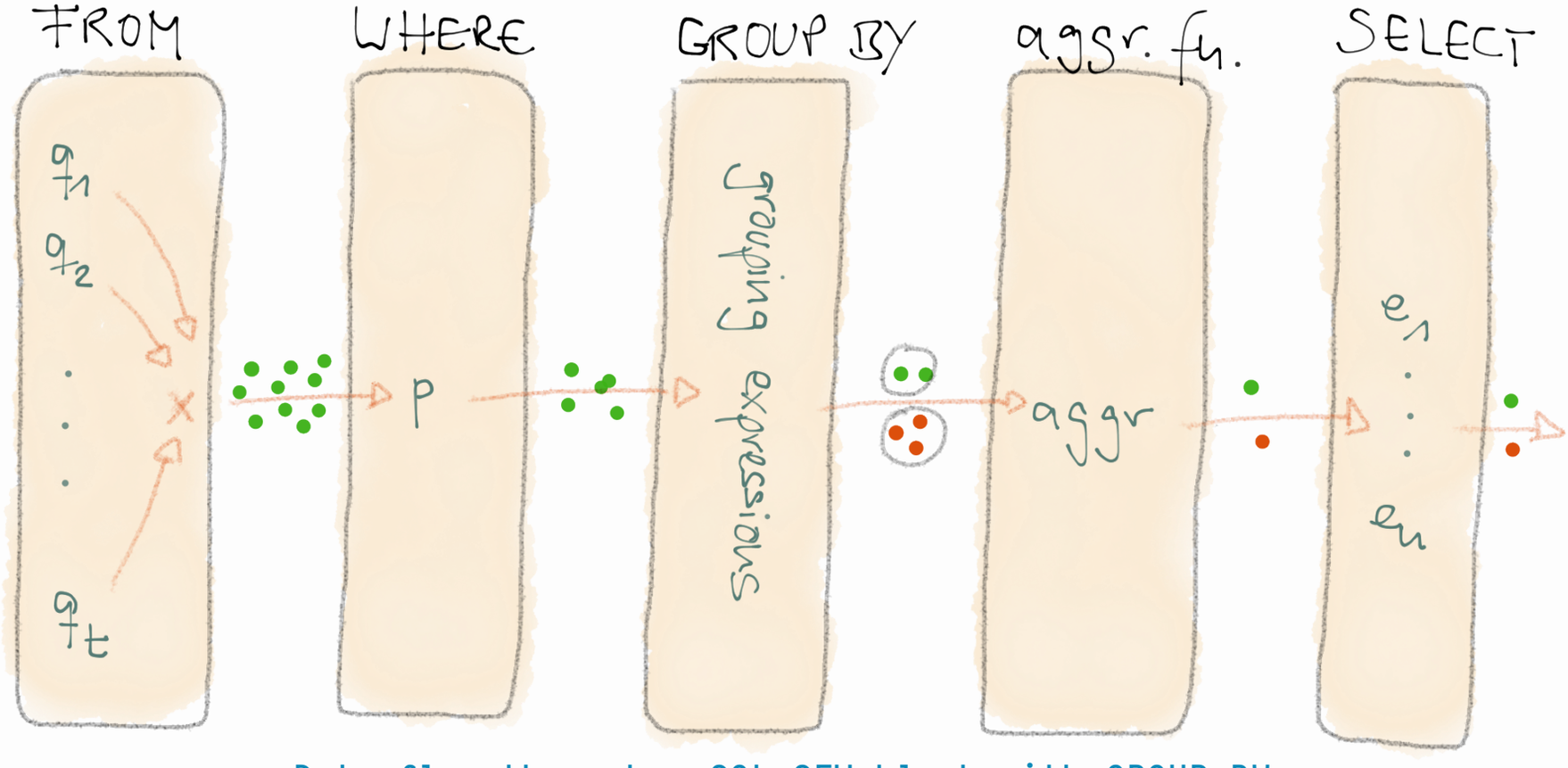
In the SELECT clause of a SFW block with GROUP BY, an output expression may only refer to **aggregates** of *non-grouping columns*.

Aggregate functions (more are listed in the PostgreSQL documentation):

```
count(expression)                          -- number of non-NULL values in group
count(*)                                   -- number of rows in group
sum(expression)                            -- sum of non-NULL values in group
avg(expression)                            -- average of non-NULL values in group
{max | min}(expression)                    -- maximum / minimum of values in group
array_agg(expression ORDER BY expression) -- array of all values in group
{bool_and | bool_or}(expression)           -- conjunction/disjunction of values
```

- Aggregate functions reduce *bag(t)* values in non-grouping columns to an atomic value.

SQL: GROUPING AND AGGREGATION



Data flow through a SQL SFW block with GROUP BY

SQL: GROUPING AND AGGREGATION

R

G	A	B
X	1	true
X	2	false
X	4	true
…	…	…
Y	8	true
Y	16	true
…	…	…
Z	32	true
Z	▢	▢

```
SELECT r.G, count(r.A) AS "count", count(*) AS "count*", sum(r.A) AS "sum",
            avg(r.A) AS "avg", max(r.A) AS "max", min(r.A) AS "min",
            array_agg(r.A ORDER BY r.A) AS "array", bool_and(r.B) AS "every", bool_or(r.B) AS
"some"
FROM   R AS r
GROUP BY r.G
```

G	count	count*	sum	avg	max	min	array	every	some
X	3	3	7	2.333	4	1	{1,2,4}	f	t
Y	2	2	24	12.000	16	8	{8,16}	t	t
Z	1	2	32	32.000	32	32	{32,▢}	t	t

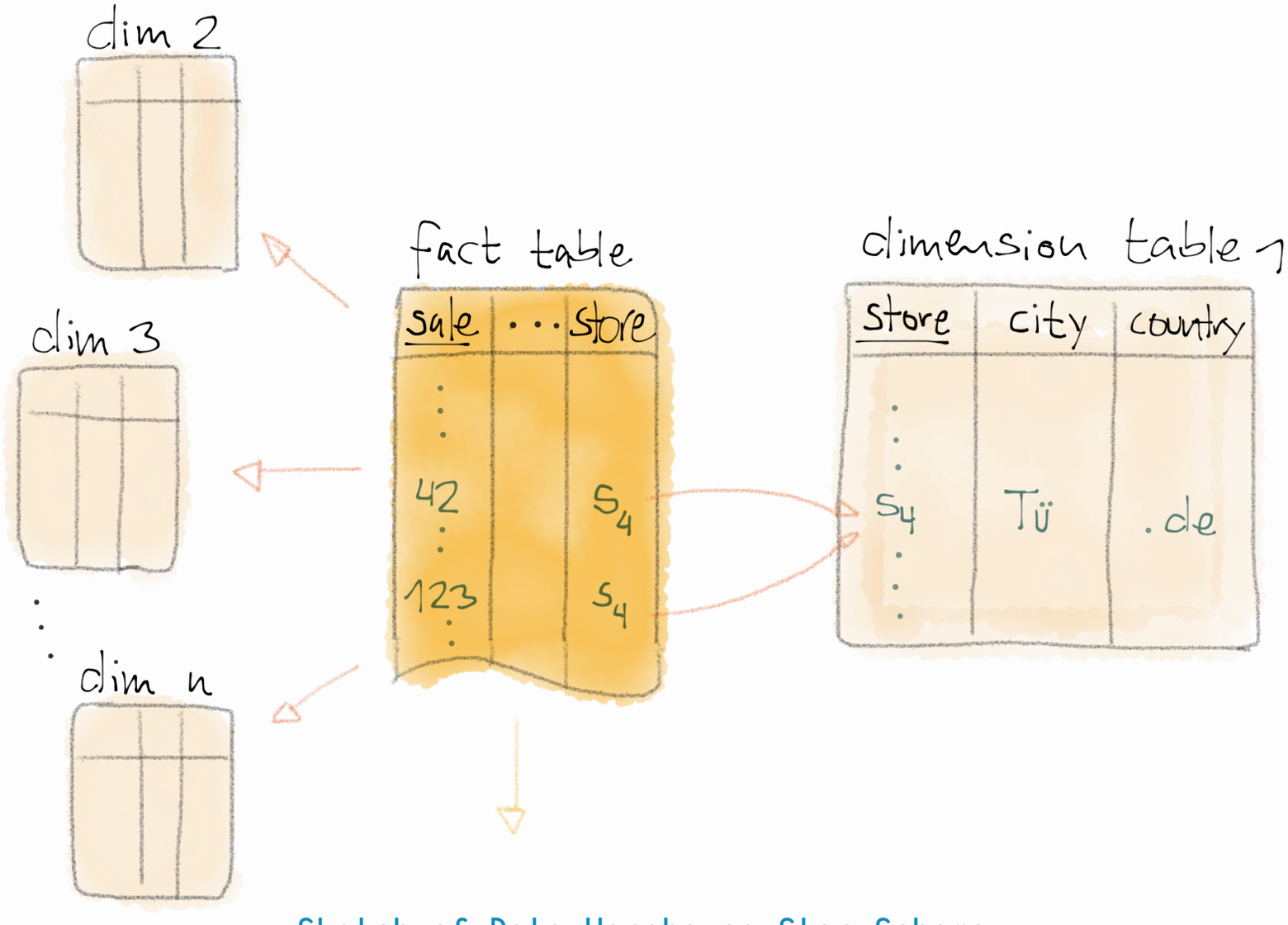
DATA WAREHOUSES

- Grouping and aggregation are essential operations for **Data Warehouses (DW)**, i.e., database applications that provide archival storage and report generation facilities for business data (think *Amazon.com*).

  1. Data about business transactions (e.g., product sales, flight bookings) are periodically moved into the DW's **fact table** (typically in batches, in periods of low system load).
     This data is considered archived and will be deleted in the operational DBMS that faces users (e.g., on the Web).
  2. To facilitate report generation, the DW "surrounds" the fact table by **dimension tables** that provide details about the product and location/date of sale. The fact table uses foreign keys to refer to the dimension tables.
  3. It is expected that the (potentially) huge fact table undergoes frequent insertions while the comparably small dimension tables are stable.

- The resulting DW schema (fact table + dimension tables) is known as **star schema**.

STAR SCHEMA



Sketch of Data Warehouse Star Schema

STAR SCHEMA OF THE LEGO STORES DW

- **Fact table** sales:

sales

sale	set	date	store	items	price
sale ID	*p*	*d*	*s*	*# of items sold*	*price of single item*

- **Dimension table** sets and **hierarchical dimension tables** stores, dates:

sets

set	name	cat	x	y	z	weight	year	img
p								

dates

date	day	day_of_week	month	quarter	year
d					

stores

store	city	state	country
s			

SQL QUERIES AGAINST THE LEGO STORES DW

1. Sales and turnover by country:

```
SELECT s.country, count(*) AS sales, sum(f.items * f.price) AS turnover
FROM   sales AS f,                          -- ⎫  from fact table navigate to
       stores AS s                          -- ⎬  dimension table(s) to
WHERE  f.store = s.store                    -- ⎭  collect required details of sale
GROUP BY s.country;
```

2. Sales by date (granularity: 1 year)

```
SELECT d."year", count(*) AS sales
FROM   sales AS f, dates AS d
WHERE  f."date" = d."date"
GROUP BY d."year";
```

3. Sales by date (granularity: 3 months)?

```
  …
```

SQL: GROUPING AND AGGREGATION

Consider tables $R(\underline{a_1}, a_2, \ldots, a_k)$ and $S(b_1, b_2, \ldots, b_r)$.

1. Describe the expected result:

```
SELECT r.a₁, count(*)
FROM   R AS r
GROUP BY r.a₁
```

```
SELECT s.b₁, s.b₂, …, s.bᵣ
FROM   S AS s
GROUP BY s.b₁, s.b₂, …, s.bᵣ
```

2. Do the results differ? Why?

```
SELECT r.a₁, count(*)
FROM   R AS r, S AS s
WHERE  r.a₁ = s.b₁
GROUP BY r.a₁
```

```
SELECT r.a₁, count(*)
FROM   R AS r, S AS s
WHERE  r.a₁ = s.b₁
GROUP BY r.a₁, r.aᵢ     -- 1 ⩽ i ⩽ k
```

SQL: GROUPING AND AGGREGATION

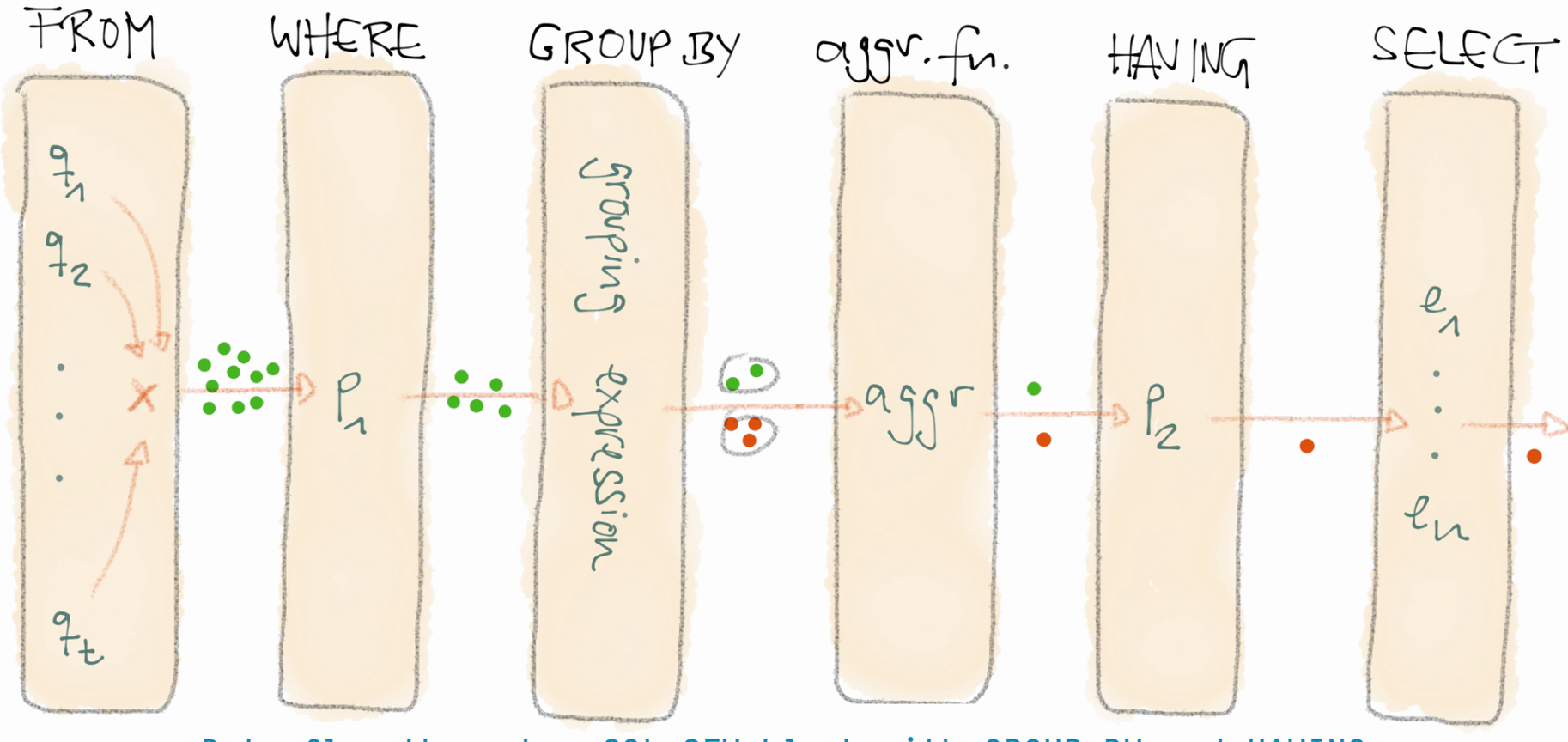
GROUP BY … HAVING

If GROUP BY is followed by the optional **HAVING clause**, row groups that fail to satisfy *condition* do not contribute to the query result:

```
<SELECT FROM WHERE block>
[  GROUP BY expression [, …]
 [ HAVING condition ]
]
```

HAVING is evaluated after GROUP BY: Boolean expression *condition* may refer to **grouping columns** as well as to **aggregates of non-grouping columns.**

SQL: GROUPING AND AGGREGATION



Data flow through a SQL SFW block with GROUP BY and HAVING

SQL: GROUPING AND AGGREGATION

(Super) Key Test

- Find the overworked employees (whose longest meetings last 1+ hours):

```
SELECT a.person
FROM   calendar AS c, attendees AS a
WHERE  c.appointment = a.appointment
GROUP BY a.person
HAVING max(c.stop - c.start) >= '01:00:00';
```

- Do the columns $a_1,\ldots,a_i$ form a key for table R?

```
SELECT DISTINCT 'The tested columns do not form a (super) key for table R'
FROM   R AS r
GROUP BY r.a1, …, r.ai
HAVING count(*) > 1
```

SQL: AGGREGATION WITHOUT GROUPING

Aggregation without GROUP BY

The SELECT clause of a SFW block may *exclusively* contain aggregate output expressions.

In this case, all rows processed by SELECT form **a single group.** The result table will contain exactly **one row** (of aggregate values).

- Count the rows in table R:

```
SELECT count(*) AS cardinality
FROM R
```

- A bit of LEGO set weight statistics (is avg1 = avg2?):

```
SELECT max(s.weight) AS "max", min(s.weight) AS "min",
       avg(s.weight) AS "avg1", sum(s.weight)/count(s.weight) AS "avg2"
FROM   sets AS s
```

SQL: AGGREGATION AND NULL

Since the **interaction of grouping and aggregation with SQL's NULL value** can be counterintuitive, we summarize the behavior here.

- ❶ Aggregations over an **empty row set** yield
  - 0 for `count(*)` and `count(`*`expression`*`)`
  - NULL for all other aggregation functions (⚠️ this includes `sum()`)
- ❷ All rows that yield **NULL for the grouping expression** form one group ("*the NULL group*")
- ❸ All aggregation functions (except `count(*)` and `array_agg()`) **ignore NULL** values. If a group contains all NULL values, see ❶ above.

[End of SQL Grouping and Aggregation Diversion.]