>>

SQL Queries (iv): Grouping

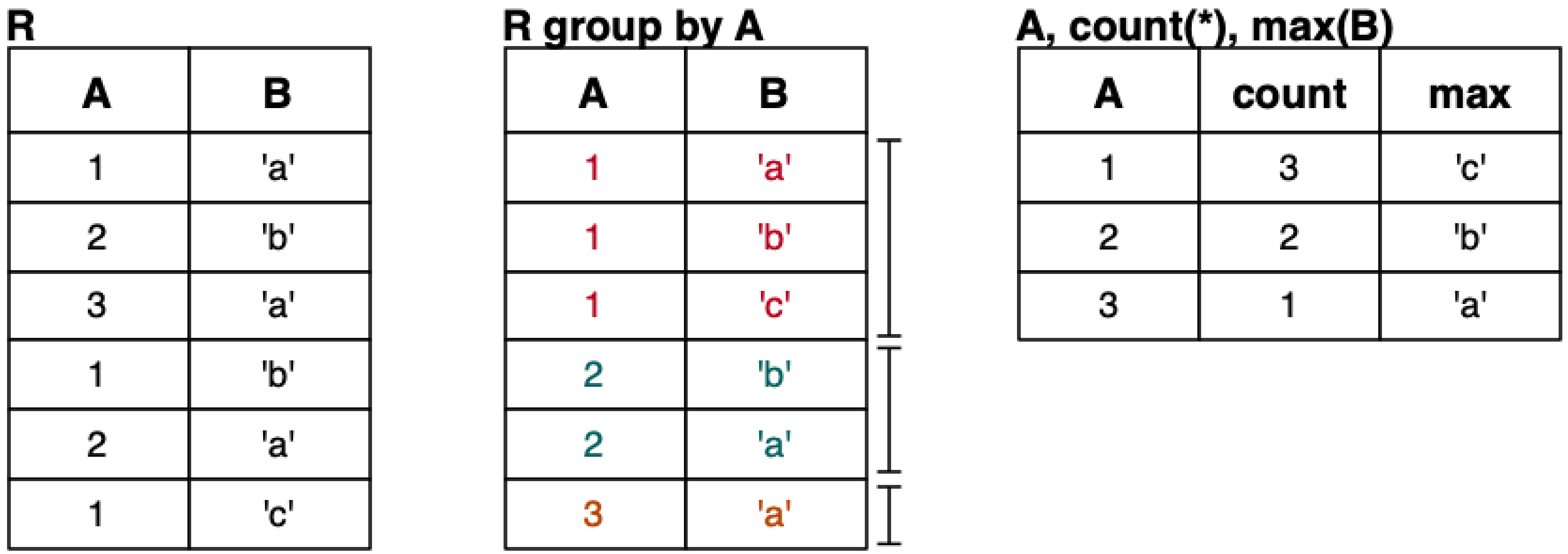
- Grouping
- Restrictions on **`SELECT`** Lists
- Filtering Groups
- Partitions

∧ >>

❖ Grouping

`SELECT-FROM-WHERE` can be followed by **`GROUP BY`** to:

- partition result relation into groups (according to values of specified attribute)
- summarise (aggregate) some aspects of each group
- output one tuple per group, with grouping attribute and aggregates



R

A	B
1	'a'
2	'b'
3	'a'
1	'b'
2	'a'
1	'c'

R group by A

A	B
1	'a'
1	'b'
1	'c'
2	'b'
2	'a'
3	'a'

A, count(*), max(B)

A	count	max
1	3	'c'
2	2	'b'
3	1	'a'

<< ^ >>

❖ Grouping (cont)

Example: How many different beers does each brewer make?

```
SELECT   brewer, COUNT(name) as nbeers
FROM     Beers
GROUP BY brewer;

       brewer          | nbeers
-----------------------+--------
 West City             |      1
 James Squire          |      5
 Yullis                |      1
 Hop Nation            |      4
 Anderson Valley       |      1
 Beatnik               |      1
 Boatrocker            |      3
 Kizakura              |      1
 ...
```

<< ^ >>

❖ Grouping (cont)

GROUP BY is used as follows:

```
SELECT   attributes/aggregations
FROM     relations
WHERE    condition
GROUP BY attributes
```

Semantics:

1. apply product and selection as for **SELECT-FROM-WHERE**
2. partition result into groups based on values of *attributes*
3. apply any aggregation separately to each group

Grouping is typically used in queries involving the phrase "for each".

<< ∧ >>

❖ Restrictions on **SELECT** Lists

When using grouping, every attribute in the **SELECT** list must:

- have an aggregation operator applied to it OR
- appear in the **GROUP-BY** clause

Incorrect Example: Find the styles associated with each brewer

```
SELECT   brewer, style
FROM     Beers
GROUP BY brewer;
```

PostgreSQL's response to this query:

```
ERROR: column beers.style must appear in the GROUP BY
         clause or be used in an aggregate function
```

<< ∧ >>

❖ Filtering Groups

In some queries, you can use the **`WHERE`** condition to eliminate groups.

Example: Average beer price by suburb excluding hotels in The Rocks.

```
SELECT   b.addr, AVG(s.price)
FROM     Sells s join Bars b on (s.bar=b.name)
WHERE    b.addr <> 'The Rocks'
GROUP BY b.addr;
```

For conditions on whole groups, use the **`HAVING`** clause.

<< ∧ >>

❖ Filtering Groups (cont)

HAVING is used to qualify a **GROUP-BY** clause:

```
SELECT   attributes/aggregations
FROM     relations
WHERE    condition1     (on tuples)
GROUP BY attributes
HAVING   condition2;    (on group)
```

Semantics of **HAVING**:

1. generate the groups as for **GROUP-BY**
2. discard groups not satisfying **HAVING** condition
3. apply aggregations to remaining groups

<< ^ >>

❖ Filtering Groups (cont)

Example: Number of styles from brewers who make at least 5 beers?

```
SELECT   brewer, count(name) as nbeers,
         count(distinct style) as nstyles
FROM     Beers
GROUP BY brewer
HAVING   count(name) > 4
ORDER BY brewer;

      brewer        | nbeers | nstyles
--------------------+--------+---------
 Bentspoke          |      9 |       7
 Carlton            |      5 |       2
 Frenchies          |      5 |       5
 Hawkers            |      5 |       5
 James Squire       |      5 |       4
 One Drop           |      9 |       7
 Sierra Nevada      |      5 |       5
 Tallboy and Moose  |      5 |       5
```

`distinct` required, otherwise **`nbeers=nstyles`** for all brewers

<< ^ >>

❖ Filtering Groups (cont)

Alternative formulation of division using **`GROUP-BY`** and **`HAVING`**

Example: Find bars that each sell all of the beers Justin likes.

```
SELECT DISTINCT S.bar
FROM   Sells S, Likes L on (S.beer = L.beer)
WHERE  L.drinker = 'Justin'
GROUP  BY S.bar
HAVING count(S.beer) =
           (SELECT count(beer) FROM Likes
            WHERE drinker = 'Justin');
```

<< ∧ >>

❖ Partitions

Sometimes it is useful to

- partition a table into groups
- compute results that apply to each group
- use these results with individual tuples in the group

Comparison with **`GROUP-BY`**

- **`GROUP-BY`** produces one tuple for each group
- **`PARTITION`** augments each tuple with group-based value(s)
- can use other functions than aggregates (e.g. ranking)
- can use attributes other than the partitioning ones

<< ^ >>

❖ Partitions (cont)

Syntax for **PARTITION**:

```
SELECT attr1, attr2, ...,
       aggregate1 OVER (PARTITION BY attri),
       aggregate2 OVER (PARTITION BY attrj), ...
FROM   Table
WHERE  condition on attributes
```

Note: the *condition* cannot include the *aggregate* value(s)

<< ^ >>

❖ Partitions (cont)

Example: show each city with daily temperature and temperature range

Schema: *Weather(city,date,temperature)*

```
SELECT city, date, temperature
       min(temperature) OVER (PARTITION BY city) as lowest,
       max(temperature) OVER (PARTITION BY city) as highest
FROM   Weather;
```

Output: *Result(city, date, temperature, lowest, highest)*

<< ^

❖ Partitions (cont)

Example showing **GROUP BY** and **PARTITION** difference:

```
SELECT city, min(temperature) max(temperature)
FROM   Weather  GROUP BY city
```

Result: one tuple for each city *Result(city,min,max)*

```
SELECT city,  date,  temperature as temp,
       min(temperature) OVER (PARTITION BY city),
       max(temperature) OVER (PARTITION BY city)
FROM   Weather;
```

Result: one tuple for each temperature measurement.

Produced: 27 Feb 2021