>>

SQL Queries (iv): Grouping

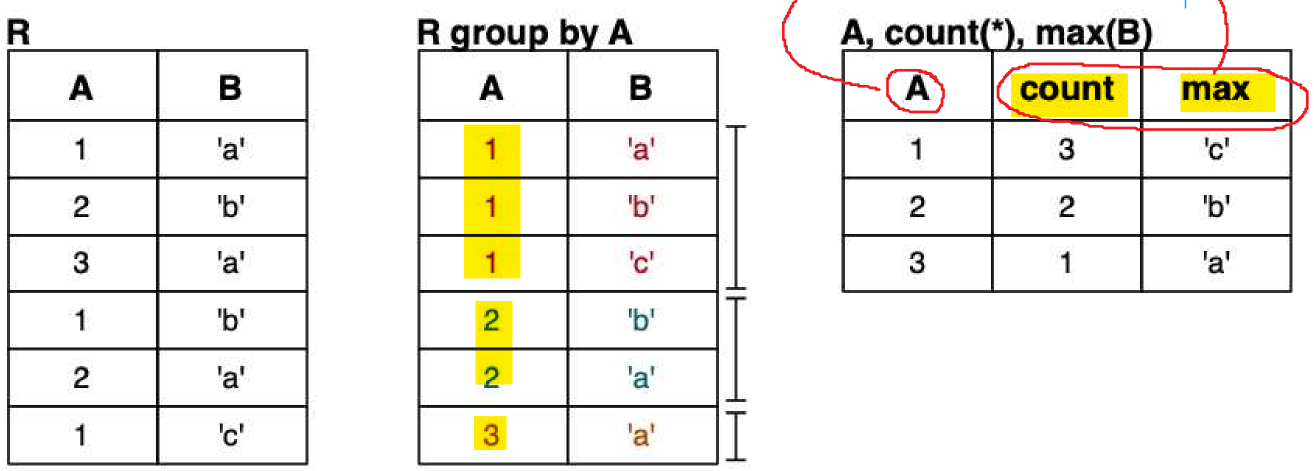
- Grouping
- Restrictions on **SELECT** Lists
- Filtering Groups
- Partitions

∧ >>

❖ 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;
```

without group by brewer there would be multiple rows of the same brewer name, each with a count of 1 next to them (unless there was a beer with a duplicated name, then the count would be n)

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

group by would get something like;

```
Boatrocker | Toohey's new
Boatrocker | Guinness
Boatrocker | carlton draught
Yullis     | Guinness
...
```

the count then gets applied to each group separately, hence we get 3 for boatrocker, 1 for yullis etc.

<< ∧ >>

❖ 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;
```

it's not 100% clear why this doesn't work, you'd think it would just list;
brewer1 | style1
brewer1 | style 2
brewer2 | style 1
etc.

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.

here the avg aggregate function allows us to group by addr

```
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');
```

starting with FROM: we get tuples like (somebar, somebeer, somedrinker, somebeer), where somebeer is the same beer in each instance because it is joined on s.beer = l.beer

now apply the WHERE: we are left with tuples of the form (somebar, somebeer, justin, somebeer), again where somebeer is the same beer.

we then GROUP those tuples BY the bar name i.e.

bar1, beer1, justin, beer1
bar2, beer1, justin, beer1
bar2, beer3, justin, beer3
bar2, beer6, justin, beer6
bar3, beer2, justin, beer2
bar3, beer4, justin, beer4
bar4, beer5, justin, beer5

note that the beerN is always the same within a tuple and that the drinker name is always justin and that the tuples are grouped by the bar name.

We then eliminate all groups which don't meet the having condition i.e. where the number of beers at the current bar group doesn't equal the number of beers that justin likes. For example, say justin likes 3 beers in total. Then the only remaining group will be bar2. If he liked 4+ in total then no groups would survive. Note that it is impossible for him to like < 3 in total because bar2 has 3, so we know he likes at least 3.

<< ∧ >>

❖ 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 attr_1, attr_2, ...,
       aggregate_1 OVER (PARTITION BY attr_i),
       aggregate_2 OVER (PARTITION BY attr_j), ...
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)*

i.e. split up all the tuples by city, take the min of all those city's temperatures as the lowest, and max for the 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: 5 Oct 2020