Biostats 597E

Week 2 - Introduction to SQL

Summarizing Data

We can use aggregate function to obtain summary statistics of columns such as count, average, and sum.

How many records?

```
select count(*) as n
  from countries
```

Largest population?

```
select max(population) as max_population
  from countries
```

Aggregate Functions

Function	Use
COUNT	count number of rows or nonmissing values of a column
MAX	maximun value for a column
MIN	minimun value for a column
AVG	average value for a column
SUM	total value for a column

Different SQL system might have other aggregate functions. For example SAS has **VAR** function to calculate variance.

Count Function

- We can use **COUNT** with **DISTINCT** to obtain count of unique values of a column

```
select count(distinct continent)
  from countries
```

- Count function only counts non-missing values

COUNT(*) and **COUNT(var)** may give different values if var has missing values.

```
select count(*) as num_rows,
       count(Area) as num_nonmissing_area
  from continents
```

Missing Values

Missing values are generally excluded from the aggregate function. Be careful while dealing data with missing values.

What happens to below code

```
select avg(Area) as mean1,
       sum(Area)/count(*) as mean2
  from continents
```

Calculate Variance

We can calculate variance without VAR function. Example: calculate variance of **AvgLow** in **worldtemps** data

```
select (sum(AvgLow*AvgLow) - count(AvgLow)*avg(AvgLow) *
        avg(AvgLow))/(count(AvgLow)-1) as variance
  from worldtemps
```

Note: We should use **count(AvgLow)** instead of **count(*)**

Grouping Data

We can use **GROUP BY** clause to compute statistics by specified groups.

Example: We want to calculate total population by continent in **countries** data.

```
select continent, sum(population) as total_population
  from countries
  where continent != ''
  group by continent
```

Group by multiple columns

```
select Location, Type, sum(Area) as Area
  from features
  where Type in ('Desert', 'Lake')
  group by Location, Type
```

Filtering Grouped Data

We can filter grouped data using **HAVING** clause

```
select Type, count(*) as Number
  from features
  group by Type
  having Type in ('Island', 'Ocean', 'Sea')
```

Continents with at least 20 countries?

```
select continent, sum(population) as total_population
  from countries
  where continent != ''
  group by continent
  having count(name) > 20
  order by total_population desc
```

WHERE vs. HAVING

- **HAVING** is typically used to specify conditions for including or excluding groups of rows. **WHERE** is used specify conditions for including or excluding individual rows from a table

- **HAVING** must follow the **GROUP BY** clause if used with a **GROUP BY**. On the other hand, **WHERE** must precede the **GROUP BY** clause if used with **GROUP BY**.

- **HAVING** is processed after the **GROUP BY** clause and any aggregate functions. On the other hand, **WHERE** is processed before a **GROUP BY** and any aggregate functions.

Filter rows in original table with **WHERE** ====> Compute aggregate statistics by groups =====> Filter groups with **HAVING**

Using Subqueries

We can nest a subquery (enclosed in parentheses) as part of another query expression. It selects rows from one table based on values in another table.

- Countries with population greater than twice average country population

```
select name, population,
  from countries
  where population > 2 * (select avg(population) from countries)
```

- Which states in US have more people than Belgium?

```
select name
  from unitedstates
  where population > (select population from countries where
                 name = 'Belgium')
```

Using Subqueries

We can use the value of subquery for calculation

- Find the ratio of each country's population over total

```
select name,
  population / (select sum(population) from countries) as ratio
  from countries
```

We can use a sub-query that returns multiple values. This usually works with **IN** operator.

- Population of major oil production countries

```
select name, population
  from countries
  where name in (select country from oilprod)
```

Selecting Data From Multiple Tables

We can specify multiple tables in the **FROM** clause.

What will return? The **cartsian product** of the two tables.

- Create test tables

```
create table one (X INT, Y INT);
insert into one values (1, 2), (2, 3);
create table two (X INT, Y INT);
insert into two values (2, 5), (3, 6), (4, 9)
```

- See the cartsian products

```
select *
  from one, two
```

Inner Join

- Cartisan product usually returns too many rows.
- Usually only matched rows are needed, for example matched ids
- We can use **WHERE** to select only matched rows after join

```
select *
  from one, two
  where one.X = two.X
```

We usually use table alias to refer to different tables

```
select t1.X, t1.Y, t2.Y as Y2
  from one as t1, two as t2
  where t1.X = t2.X
```

as for table alias is optional

Examples

- Oil production and reserve for countries

```
select t1.*, t2.Barrels
  from oilprod t1, oilrsrvs t2
  where t1.Country = t2.Country
```

We notice there are less rows returned than original data due to unmatches.

- Find Coordinates of State Capitals

```
select t1.Name, t1.Capital, t3.Latitude, t3.Longitude
  from unitedstates t1, postalcodes t2, uscitycoords t3
  where t1.Name = t2.Name and t2.Code = t3.State and
  t1.Capital = t3.City
```

Self-join

We can join a table to itself to find relations within the table

- Latitude difference between each pair of cities in state CA

```
select t1.City as City1, t2.City as City2,
  (t1.Latitude - t2.Latitude)**2
from uscitycoords t1, uscitycoords t2
where t1.State = "CA" and t2.State = "CA" and t1.City < t2.City
```

Many to Many Join

Many to many join means each joining id may have multiple records in both tables. This can generate side effects. We need to make sure to understand what to expect.

```
insert into one values (2, 9), (3, 3);
insert into two values (2, 11);


select *
  from one t1, two t2
  where t1.X = t2.X
```

The result is **cartsian product** within matched records.

Think about why **cartesian product** is not a issue for one to one or one to many join?