SQL: subqueries

• A note for SELECT
• Join tricks
• Unions
• Data types
• Subqueries

A Note for SELECT

• help me debug this code!
SELECT st, COUNT(model)
FROM sales
GROUP BY model
• I got something like this but it looks weird…what’s happening?

<table>
<thead>
<tr>
<th>sales</th>
<th>returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td>model</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>WA</td>
<td>LEAF</td>
</tr>
<tr>
<td>WA</td>
<td>I3</td>
</tr>
<tr>
<td>HI</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>I3</td>
</tr>
<tr>
<td>CA</td>
<td>S60</td>
</tr>
<tr>
<td>CA</td>
<td>SPARKEV</td>
</tr>
<tr>
<td>GA</td>
<td>LEAF</td>
</tr>
<tr>
<td>GA</td>
<td>I3</td>
</tr>
<tr>
<td>OR</td>
<td>LEAF</td>
</tr>
<tr>
<td>OR</td>
<td>I3</td>
</tr>
<tr>
<td>UT</td>
<td>LEAF</td>
</tr>
<tr>
<td>CO</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>SPARKEV</td>
</tr>
</tbody>
</table>

A Note for SELECT

• notice that we aggregate the model
• but the st information that we select is not aggregated
• so we’re basically telling SQL to select a non-aggregated value to be presented for each aggregated row

A Note for SELECT

• If I manually group models...
SELECT st, model
FROM sales
ORDER BY model
• For each model group, there are multiple st values. Our previous query says that we should present only one single st value per model. But it doesn’t specify HOW!
• So our previous query returns a random st for each group — but we usually do not want indeterministic behaviors
A Note for SELECT

- solution 1: specify how st should be selected within each group

```sql
SELECT MIN(st), COUNT(model)
FROM sales
GROUP BY model
```

- For each group, I told SQL to select the first state (alphabetically)

<table>
<thead>
<tr>
<th>st</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>500E</td>
</tr>
<tr>
<td>WA</td>
<td>I3</td>
</tr>
<tr>
<td>OR</td>
<td>I3</td>
</tr>
<tr>
<td>CA</td>
<td>I3</td>
</tr>
<tr>
<td>GA</td>
<td>I3</td>
</tr>
<tr>
<td>UT</td>
<td>LEAF</td>
</tr>
<tr>
<td>OR</td>
<td>LEAF</td>
</tr>
<tr>
<td>GA</td>
<td>LEAF</td>
</tr>
<tr>
<td>WA</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>LEAF</td>
</tr>
<tr>
<td>HI</td>
<td>LEAF</td>
</tr>
<tr>
<td>CO</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>S60</td>
</tr>
<tr>
<td>CA</td>
<td>SPARKEV</td>
</tr>
</tbody>
</table>

A Note for SELECT

- solution 2: aggregate st as well

```sql
SELECT st, model, COUNT(model)
FROM sales
GROUP BY st, model
```

- so now we group by each {st, model} pair/combination

<table>
<thead>
<tr>
<th>st</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>500E</td>
</tr>
<tr>
<td>WA</td>
<td>I3</td>
</tr>
<tr>
<td>OR</td>
<td>I3</td>
</tr>
<tr>
<td>CA</td>
<td>I3</td>
</tr>
<tr>
<td>GA</td>
<td>I3</td>
</tr>
<tr>
<td>UT</td>
<td>LEAF</td>
</tr>
<tr>
<td>OR</td>
<td>LEAF</td>
</tr>
<tr>
<td>GA</td>
<td>LEAF</td>
</tr>
<tr>
<td>WA</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>LEAF</td>
</tr>
<tr>
<td>HI</td>
<td>LEAF</td>
</tr>
<tr>
<td>CO</td>
<td>LEAF</td>
</tr>
<tr>
<td>CA</td>
<td>S60</td>
</tr>
<tr>
<td>CA</td>
<td>SPARKEV</td>
</tr>
</tbody>
</table>

A Note for SELECT

- In some db settings, our problematic query will return an error message like this:

```
Expression #1 of SELECT list is not in GROUP BY clause and contains nonaggregated column 'sales.ST' which is not functionally dependent on columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by
```

- lesson: we can only select values that are in an aggregate function or in the GROUP BY clause

Recap: Join

- two equivalent ways to write a join expression

```sql
select *
from car, sales
where car.model = sales.model;
```

```sql
select *
from car
join sales
on car.model = sales.model;
```
Types of joins

• (INNER) JOIN: Select records that have matching values in both tables.
• LEFT (OUTER) JOIN: Select records from the first (left-most) table with matching right table records.
• RIGHT (OUTER) JOIN: Select records from the second (right-most) table with matching left table records.
• FULL (OUTER) JOIN: Select all records that match either left or right table records.

source: http://www.dofactory.com/sql/join

Practices...

• Find all car models that are not sold in any state.
  • I’ll first do a join so that I can get both the model and the sales info in one table...
  
  SELECT *
  FROM car
  JOIN sales
  ON car.model = sales.model;

Error: Column 'model' in field list is ambiguous
 Practices…

• Find all car models that are not sold in any state.

• There are two columns called model! Which one?
  • Also, to see “NOT sold” I realize I can match the two
tables, and find out values that don’t match up
  • which translates to doing a left or right outer join,
and observing those NULL values
  • Which join? Well, I want all records of car models,
even if they don’t have matches in the sales table
(which contains the state information), so it’s a left
join

Practices…

• Find all car models that are not sold in any state.
• A left join...
  SELECT *
  FROM car
  LEFT JOIN sales
  ON car.model = sales.model;

Error: Column 'model' in field list is ambiguous
Practices...

- Find all car models that are not sold in any state.
- ...and finally select only the model names

```sql
SELECT car.model
FROM car
LEFT JOIN sales
ON car.model = sales.model
WHERE sales.model IS NULL;
```

Aliasing tables

- Table aliasing comes up often in joins because we’ll have to distinguish between the tables. When the table names gets really long, we will give them aliases.
- Table aliases come right after the table name with no keywords (The ‘AS’ keyword is for column names!)
- Example: find all car models that are not sold in any state.

```sql
SELECT c.model
FROM car c
LEFT JOIN sales s
ON c.model = s.model
WHERE s.model IS NULL;
```

Different flavors of SQL

- Multiple standard revisions and multiple flavors/implementations exist
- Revisions of the standard: successively introduced additional features, mostly backwards-compatible with older standards
- Common implementations: Oracle, MySQL, SQLite, Postgres, MSQL (Microsoft), Microsoft Access
- Implementations comply with the standard, but people almost never completely implement everything in the standard, e.g. FULL JOIN not implemented in MySQL but is implemented in PostgreSQL

Market share: Source: JoinVision E-Services GmbH, July 2006
UNIONs

- stack a dataset on top of another dataset with the same schema
- Restrictions
  - Both tables must have the same number of columns
  - The columns must have the same data types in the same order as the first table
- UNION appends distinct values
- UNION ALL appends all values (probably used more often)

UNIONs: example

- can simulate a FULL OUTER JOIN using unions
- example: the User and Likes database

```sql
SELECT users.name, likes.like
FROM users
LEFT OUTER JOIN likes
ON users.id = likes.user_id
UNION
SELECT users.name, likes.like
FROM users
RIGHT OUTER JOIN likes
ON users.id = likes.user_id
```

JOIN tricks — self joins

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Salary</th>
<th>ManagerId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>70000</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Henry</td>
<td>80000</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Sam</td>
<td>60000</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>Max</td>
<td>90000</td>
<td>NULL</td>
</tr>
</tbody>
</table>

- Find people who earn more than their managers. In our example the query should return

```
Employee
Joe
```

JOIN tricks — self joins

- so the names of employees and the names of managers are linked by Id and ManagerId
- to find the name of X’s manager, we’ll have to go to X’s ManagerId field, and find the other person Y, whose Id is X’s ManagerId
- to translate that relationship into an SQL query, we need two copies of the Employee table

```sql
SELECT *
FROM Employee table1
JOIN Employee table2
ON table1.ManagerId = table2.Id
```

table1 and table2 are both aliases!
JOIN tricks — self joins
• just a join

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Salary</th>
<th>ManagerId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>70000</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Henry</td>
<td>80000</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Sam</td>
<td>60000</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>Max</td>
<td>90000</td>
<td>NULL</td>
</tr>
</tbody>
</table>

JOIN table1
...ON table1.ManagerId = table2.Id

SELECT table1.Name AS 'Employee'
FROM Employee table1
JOIN Employee table2
ON table1.ManagerId = table2.Id
WHERE table1.Salary > table2.Salary

JOIN tricks — inequality joins
• the ON clause specifies a matching rule for the two tables, so it doesn’t have to be an equal
• example: find pairs of car model names, in which the first name is shorter than the next

SELECT car1.model, car2.model
FROM car AS car1
JOIN car AS car2
ON length(car1.model) < length(car2.model)

JOIN tricks — multiple keys
• If the Sales table also have a man column, we could

SELECT *
FROM car
JOIN sales
ON car.model = sales.model
AND car.man = sales.man
JOIN tricks — multiple keys

• Why?
  • accuracy
  • speed
  • has to do with implementations — “indexes”
  • might hit one of the indexes and get a better speed

Datatypes

• why datatypes?
  • easier for data manipulation
  • easier to store, and saves space on disk!
  • it’s important because functions usually act on just one or several types of data, e.g. SUM() for numerical data
  • depending on the implementation of the database, datatypes vary, but more or less the same
  • in MySQL 5.7, there are 5 major categories (numeric, text, date/time, spatial, JSON) — each has a handful of datatypes

Datatypes: numeric

• Integer Types (Exact Value): INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT
• Fixed-Point Types (Exact Value): DECIMAL, NUMERIC
• Floating-Point Types (Approximate Value): FLOAT, DOUBLE
• Bit-Value Type: BIT

• the difference is in the size. e.g. TINYINT ranges from -128 to 127, while INT ranges from -2147483648 to 2147483647

Datatypes: string

• CHAR, VARCHAR
  • fixed and variable length string
• BINARY, VARBINARY
  • fixed and variable length binary
• BLOB (Binary Large Objects), TEXT
  • binary and characters of bigger size
• ENUM
  • a list of possible values (e.g. ‘XS’, ‘S’, ‘M’, ‘L’, ‘XL’)
• SET
  • like enum but stores more than 1 choice
Datatypes: date and time

- **DATE**
  - YYYY-MM-DD
- **DATETIME**
  - YYYY-MM-DD HH:MI:SS
- **TIMESTAMP**
  - Also YYYY-MM-DD HH:MI:SS, but stored as the number of seconds since ‘1970-01-01 00:00:00’ UTC (i.e. the Unix epoch)
- **TIME**
  - HH:MI:SS
- **YEAR**
  - A year in two-digit or four-digit format.

Datatypes: date and time

Date and time are tricky to get right! How do you...

- compare dates?
- take differences?
- group into weeks or months?
- convert string to dates?

Datatypes: others

- **Spatial**
  - GEOMETRY, POINT, LINESTRING, POLYGON etc
- **JSON**


Subqueries

- a feature that allows performing operations in more than 1 step
- e.g. I want to the sum of price, battery, mpge, miles and weight values for each row, and find the average of those sums
- Subqueries can be used in several places within a query – FROM, WHERE, JOIN...

<table>
<thead>
<tr>
<th>MODEL</th>
<th>MAN</th>
<th>PRICE</th>
<th>BATTERY</th>
<th>MPGE</th>
<th>MILES</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>I3</td>
<td>BMW</td>
<td>42375</td>
<td>22</td>
<td>123</td>
<td>80</td>
<td>2700</td>
</tr>
<tr>
<td>BCE</td>
<td>MB</td>
<td>42375</td>
<td>28</td>
<td>84</td>
<td>80</td>
<td>3935</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>CHEVY</td>
<td>27495</td>
<td>20</td>
<td>119</td>
<td>82</td>
<td>2989</td>
</tr>
<tr>
<td>LEAF</td>
<td>NISSAN</td>
<td>31820</td>
<td>24</td>
<td>115</td>
<td>75</td>
<td>3340</td>
</tr>
<tr>
<td>560</td>
<td>TESLA</td>
<td>71070</td>
<td>60</td>
<td>89</td>
<td>208</td>
<td>4647</td>
</tr>
<tr>
<td>500E</td>
<td>FIAT</td>
<td>32650</td>
<td>24</td>
<td>116</td>
<td>87</td>
<td>2980</td>
</tr>
</tbody>
</table>
Subqueries in FROM

• Example

SELECT sub.*
FROM ( 
    SELECT *
    FROM car
    WHERE price < 35000
) sub
WHERE sub.battery = 24

Subqueries in FROM

• The database runs the inner query/subquery first:

\[
\begin{align*}
\text{SELECT} & \quad \text{*} \\
\text{FROM} & \quad car \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• The inner query runs just like other queries, and the database treats this inner query as an independent query

• The inner query produces a result table

Subqueries in FROM

• After the inner query is done, the database runs the outer query using that result table produced from the inner query

SELECT sub.*
FROM sub
WHERE sub.battery = 24

Subqueries in FROM

• the subquery here has a name, written right outside/after the parenthesis

\[
\begin{align*}
\text{... ( SELECT} & \quad \text{*} \\
\text{FROM} & \quad \text{car} \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• note that it’s the same way that we give an alias to a normal table:

SELECT c.model
FROM car c
LEFT JOIN sales s
ON c.model = s.model

Subqueries in FROM

• The database runs the inner query/subquery first:

\[
\begin{align*}
\text{SELECT} & \quad \text{*} \\
\text{FROM} & \quad \text{car} \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• The inner query runs just like other queries, and the database treats this inner query as an independent query

• The inner query produces a result table

Subqueries in FROM

• After the inner query is done, the database runs the outer query using that result table produced from the inner query

SELECT sub.*
FROM sub
WHERE sub.battery = 24

Subqueries in FROM

• the subquery here has a name, written right outside/after the parenthesis

\[
\begin{align*}
\text{... ( SELECT} & \quad \text{*} \\
\text{FROM} & \quad \text{car} \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• note that it’s the same way that we give an alias to a normal table:

SELECT c.model
FROM car c
LEFT JOIN sales s
ON c.model = s.model

Subqueries in FROM

• The database runs the inner query/subquery first:

\[
\begin{align*}
\text{SELECT} & \quad \text{*} \\
\text{FROM} & \quad \text{car} \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• The inner query runs just like other queries, and the database treats this inner query as an independent query

• The inner query produces a result table

Subqueries in FROM

• After the inner query is done, the database runs the outer query using that result table produced from the inner query

SELECT sub.*
FROM sub
WHERE sub.battery = 24

Subqueries in FROM

• the subquery here has a name, written right outside/after the parenthesis

\[
\begin{align*}
\text{... ( SELECT} & \quad \text{*} \\
\text{FROM} & \quad \text{car} \\
\text{WHERE} & \quad \text{price} < 35000
\end{align*}
\]

• note that it’s the same way that we give an alias to a normal table:

SELECT c.model
FROM car c
LEFT JOIN sales s
ON c.model = s.model
Subqueries in WHERE

• Find the price of the second most expensive car
  • we do this by first finding the highest price $P$, and then selecting the highest price that’s less than $P$

```sql
SELECT MAX(price)
FROM car
WHERE price < (SELECT MAX(price)
               FROM car)
```

Subqueries in WHERE

• The database runs the inner query first — the highest price

```sql
SELECT MAX(price)
FROM car
```

Subqueries in WHERE

• ...then the outer query — the highest price within prices that’s less than the absolute highest price

```sql
SELECT MAX(price)
FROM car
WHERE price < [our result from the inner query]
```

• Our query works because the result of the subquery is only one cell!

Subqueries in WHERE

• When then inner query contains multiple cells, use IN:

```sql
SELECT AVG(price)
FROM car
WHERE price IN (SELECT price
                 FROM car
                 WHERE price < 35000)
```
In conclusion...

• There are 4 types of joins to integrate information from two tables.
• UNION and UNION ALL stitches two parts of the dataset together.
• Common data types include numerics, strings and date/times.
• Subquery runs before the outer query, so that the outer query can use the result from the subquery as the underlying table.

Stay tuned for next week

• subqueries
• inserting, deleting and updating records
• constraints for tables
• (maybe) window functions
• course eval