Lecture 4:
Advanced SQL – Part II
Announcements!

1. Problem Set #1 is released!
   • We will discuss some of the questions at the end of this lecture

2. Project group assignments
   • Does everybody have a team?

3. Ask questions, Go to office hours, Engage on Piazza
Lecture 4: Advanced SQL – Part II
Today’s Lecture

1. Aggregation & GROUP BY
   • ACTIVITY: Fancy SQL Part I

2. Advance SQL-izing
   • ACTIVITY: Fancy SQL Part II

3. Problem Set #1 Overview
1. Aggregation & GROUP BY
What you will learn about in this section

1. Aggregation operators

2. GROUP BY

3. GROUP BY: with HAVING, semantics

4. ACTIVITY: Fancy SQL Pt. I
Aggregation

- SQL supports several aggregation operations:
  - SUM, COUNT, MIN, MAX, AVG

```
SELECT AVG(price)
FROM Product
WHERE maker = "Toyota"
```

```
SELECT COUNT(*)
FROM Product
WHERE year > 1995
```

Except COUNT, all aggregations apply to a single attribute
Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category) FROM Product WHERE year > 1995
```

Note: Same as COUNT(*).
Why?

We probably want:

```
SELECT COUNT(DISTINCT category) FROM Product WHERE year > 1995
```
More Examples

Purchase(product, date, price, quantity)

SELECT SUM(price * quantity) FROM Purchase

What do these mean?

SELECT SUM(price * quantity) FROM Purchase WHERE product = 'bagel'
Simple Aggregations

Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

SELECT SUM(price * quantity) FROM Purchase WHERE product = 'bagel'

50 (= 1*20 + 1.50*20)
Grouping and Aggregation

Let’s see what this means…

SELECT product, 
    SUM(price * quantity) AS TotalSales 
FROM Purchase 
WHERE date > '10/1/2005' 
GROUP BY product 

Find total sales after 10/1/2005 per product.
Grouping and Aggregation

Semantics of the query:

1. Compute the `FROM` and `WHERE` clauses

2. Group by the attributes in the `GROUP BY`

3. Compute the `SELECT` clause: grouped attributes and aggregates
1. Compute the **FROM** and **WHERE** clauses

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
2. Group by the attributes in the **GROUP BY**

```sql
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

GROUP BY Product

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
3. Compute the **SELECT** clause: grouped attributes and aggregates

```sql
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
GROUP BY v.s. Nested Queries

**SELECT**
```
product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

**SELECT DISTINCT**
```
x.product,
(SELECT Sum(y.price*y.quantity)
FROM Purchase y
WHERE x.product = y.product
    AND y.date > '10/1/2005') AS TotalSales
FROM Purchase x
WHERE x.date > '10/1/2005'
```
HAVING Clause

SELECT  product, SUM(price*quantity)
FROM    Purchase
WHERE   date > '10/1/2005'
GROUP BY product
HAVING  SUM(quantity) > 100

Whereas WHERE clauses condition on individual tuples...

HAVING clauses contains conditions on aggregates

Same query as before, except that we consider only products that have more than 100 buyers
General form of Grouping and Aggregation

\[
\text{SELECT } S \quad \text{FROM } R_1, \ldots, R_n \\
\text{WHERE } C_1 \\
\text{GROUP BY } a_1, \ldots, a_k \\
\text{HAVING } C_2
\]

- $S$ = Can ONLY contain attributes $a_1, \ldots, a_k$ and/or aggregates over other attributes
- $C_1$ = is any condition on the attributes in $R_1, \ldots, R_n$
- $C_2$ = is any condition on the aggregate expressions
General form of Grouping and Aggregation

```
SELECT S
FROM R₁,...,Rₙ
WHERE C₁
GROUP BY a₁,...,aₖ
HAVING C₂
```

Evaluation steps:
1. Evaluate **FROM-WHERE**: apply condition $C₁$ on the attributes in $R₁,...,Rₙ$
2. **GROUP BY** the attributes $a₁,...,aₖ$
3. **Apply condition** $C₂$ to each group (may have aggregates)
4. Compute aggregates in $S$ and return the result
Group-by v.s. Nested Query

- Find authors who wrote \( \geq 10 \) documents:
- Attempt 1: with nested queries

```sql
SELECT DISTINCT Author.name
FROM Author
WHERE COUNT(
    SELECT Wrote.url
    FROM Wrote
    WHERE Author.login = Wrote.login)
    > 10
```

This is SQL by a novice.
Group-by v.s. Nested Query

• Find all authors who wrote at least 10 documents:
• Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name
FROM Author, Wrote
WHERE Author.login = Wrote.login
GROUP BY Author.name
HAVING COUNT(Wrote.url) > 10
```

No need for **DISTINCT**: automatically from **GROUP BY**
Group-by vs. Nested Query

Which way is more efficient?

• Attempt #1- *With nested*: How many times do we do a SFW query over all of the Wrote relations?

• Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **much** more efficient!
Activity-4-1.ipynb
3. Advanced SQL-izing
What you will learn about in this section

1. Quantifiers

2. NULLs

3. Outer Joins

4. ACTIVITY: Fancy SQL Pt. II
Quantifiers

Product(name, price, company)
Company(name, city)

SELECT DISTINCT Company.cname
FROM Company, Product
WHERE Company.name = Product.company
AND Product.price < 100

Find all companies that make some products with price < 100

An existential quantifier is a logical quantifier (roughly) of the form “there exists”

Existential: easy 😊
Quantifiers

Product(name, price, company)
Company(name, city)

SELECT DISTINCT Company.cname
FROM Company
WHERE Company.name NOT IN(
    SELECT Product.company
    FROM Product
    WHERE price >= 100)

A universal quantifier is of the form “for all”

Equivalent

Find all companies with products all having price < 100

Find all companies that make only products with price < 100

Universal: hard 😞
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL

• Can mean many things:
  • Value does not exist
  • Value exists but is unknown
  • Value not applicable
  • Etc.

• The schema specifies for each attribute if can be null (*nullable* attribute) or not

• How does SQL cope with tables that have NULLs?
Null Values

• *For numerical operations*, NULL -> NULL:
  • If x = NULL then $4^{*}(3-x)/7$ is still NULL

• *For boolean operations*, in SQL there are three values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>

  • If x = NULL then x = “Joe” is UNKNOWN
Null Values

• C1 AND C2 = min(C1, C2)
• C1 OR C2 = max(C1, C2)
• NOT C1 = 1 – C1

```sql
SELECT * 
FROM Person 
WHERE (age < 25) 
    AND (height > 6 AND weight > 190)

Won’t return e.g. (age=20 height=NULL weight=200)!
```

Rule in SQL: include only tuples that yield TRUE (1.0)
Null Values

Unexpected behavior:

```
SELECT * 
FROM Person 
WHERE age < 25 OR age >= 25
```

Some Persons are not included!
Null Values

Can test for NULL explicitly:

- $x \text{ IS NULL}$
- $x \text{ IS NOT NULL}$

```
SELECT * 
FROM Person 
WHERE age < 25 OR age >= 25 
OR age IS NULL
```

Now it includes all Persons!
RECAP: Inner Joins

By default, joins in SQL are "inner joins":

```
SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

Both equivalent: Both INNER JOINS!
Inner Joins + NULLS = Lost data?

By default, joins in SQL are “inner joins”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!
Outer Joins

• An **outer join** returns tuples from the joined relations that don’t have a corresponding tuple in the other relations
  • I.e. If we join relations A and B on $a.X = b.X$, and there is an entry in A with $X=5$, but none in B with $X=5$...
    • A LEFT OUTER JOIN will return a tuple (a, NULL)!

• Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```

Now we’ll get products even if they didn’t sell
INNER JOIN:

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

**SELECT** `Product.name, Purchase.store`

**FROM** `Product`

**INNER JOIN** `Purchase` on `Product.name = Purchase.prodName`

Note: another equivalent way to write an INNER JOIN!
# LEFT OUTER JOIN:

## Products

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

## Purchases

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

```sql
SELECT Product.name, Purchase.store
FROM Product
    LEFT OUTER JOIN Purchase
    ON Product.name = Purchase.prodName
```
Other Outer Joins

• Left outer join:
  • Include the left tuple even if there’s no match

• Right outer join:
  • Include the right tuple even if there’s no match

• Full outer join:
  • Include the both left and right tuples even if there’s no match
Activity-4-2.ipynb
Summary

SQL is a rich programming language that handles the way data is processed \textit{declaratively}.
Problem Set #1: SQL Uber Alles
Problems in PS#1

1. Linear algebra in SQL

2. Precipitation data and nested queries

3. The traveling SQL salesman: Graph traversals in SQL
Linear algebra in SQL

1. Simple joins with aggregations

2. Hint 1: Using aliases leads to clean SQL

3. Hint 2: SQL supports many operations over numeric attributes (in the SELECT part of an SFW query)

   i  INT: Row index
   j  INT: Column index
   val INT: Cell value

   ```sql
   SELECT MAX(A.val*B.val)
   FROM   A, B
   WHERE  A.i = B.i AND A.j = B.j
   ```
Precipitation data and nested queries

1. Aggregates inside nested queries. Remember SQL is **compositional**

2. Hint 1: Break down query description to steps (subproblems)

3. Hint 2: Whenever in doubt always go back to the definition
Precipitation data and nested queries

Example:

“Using a single SQL query, find all of the stations that had the highest daily precipitation (across all stations) on any given day.”

```
SELECT station_id, day
FROM precipitation,
     (SELECT day AS maxd, MAX(precipitation) AS maxp
      FROM precipitation
      GROUP BY day)
WHERE day = maxd AND precipitation = maxp
```
The traveling SQL salesman: Graph traversals in SQL

1. Views: Details in the description. Nothing more than temp aliases for queries. Remember: SQL is compositional!

2. Self-joins are very powerful
The traveling SQL salesman: Graph traversals in SQL

Example:
“Find all paths of size two in a directed graph”

```
SELECT e1.src, e1.trg, e2.trg
FROM edges AS e1, edges AS e2,
WHERE e1.trg = e2.src
```

<table>
<thead>
<tr>
<th>edge_id</th>
<th>src</th>
<th>trg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>