

# 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

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

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

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

*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?

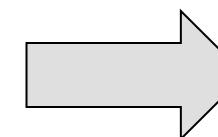
```
SELECT SUM(price * quantity)  
FROM Purchase  
WHERE product = 'bagel'
```

# Simple Aggregations

## Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

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



50 (= 1\*20 + 1.50\*20)

# Grouping and Aggregation

Purchase(product, date, price, quantity)

```
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.

Let's see what this means...

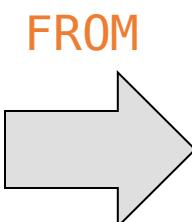
# Grouping and Aggregation

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
```



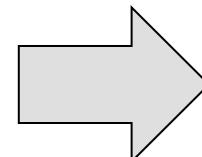
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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

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

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

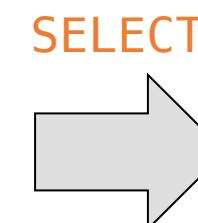


Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

### 3. Compute the **SELECT** clause: grouped attributes and aggregates

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

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10



Product	TotalSales
Bagel	50
Banana	15

# GROUP BY v.s. Nested Quereis

```
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
```

HAVING clauses contains conditions on **aggregates**

Same query as before, except that we consider only products that have more than 100 buyers

Whereas WHERE clauses condition on *individual tuples*...

# General form of Grouping and Aggregation

<b>SELECT</b>	S
<b>FROM</b>	$R_1, \dots, R_n$
<b>WHERE</b>	$C_1$
<b>GROUP BY</b>	$a_1, \dots, a_k$
<b>HAVING</b>	$C_2$

Why?

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

# General form of Grouping and Aggregation

<b>SELECT</b>	S
<b>FROM</b>	$R_1, \dots, R_n$
<b>WHERE</b>	$C_1$
<b>GROUP BY</b>	$a_1, \dots, a_k$
<b>HAVING</b>	$C_2$

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition  $C_1$  on the attributes in  $R_1, \dots, R_n$
2. **GROUP BY** the attributes  $a_1, \dots, a_k$
3. **Apply condition  $C_2$  to each group (may have aggregates)**
4. Compute aggregates in S and return the result

# Group-by v.s. Nested Query

```
Author(login, name)  
Wrote(login, url)
```

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

```
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
```

This is  
SQL by  
an expert

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
```

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

Find all companies that make some products with price < 100

Existential: easy ! 😊

# Quantifiers

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

Find all companies  
with products all  
having price < 100

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

↓ Equivalent

Find all companies  
that make only  
products with price  
< 100

A universal quantifier is of  
the form “for all”

Universal: hard ! 😞

# NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - 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  $x = \text{NULL}$  then  $4*(3-x)/7$  is still NULL
- *For boolean operations*, in SQL there are three values:

**FALSE**        =        0

**UNKNOWN**    =        0.5

**TRUE**          =        1

- If  $x = \text{NULL}$  then  $x = \text{"Joe"}$  is UNKNOWN

# Null Values

- $C1 \text{ AND } C2 = \min(C1, C2)$
- $C1 \text{ OR } C2 = \max(C1, C2)$
- $\text{NOT } C1 = 1 - C1$

```
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 IS NULL
- x 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**”:

```
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
```

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  $(a, \text{NULL})$ !
- 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:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Note: another equivalent way to write an  
INNER JOIN!



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

# LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

# 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  
*declaratively*

# Problem Set #1: SQL Über 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

```
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.”

Precipitation		
station_id	day	precipitation
16102	1	10
16102	4	10
16102	24	30

```
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”

Edges		
edge_id	src	trg
1	A	B
2	B	C
3	C	D

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

**Some more examples:** <https://www.fusionbox.com/blog/detail/graph-algorithms-in-a-database-recursive-ctes-and-topological-sort-with-postgres/620/>