Lecture 5: SQL Part IV

Today's Lecture

1. Aggregation & Group By
   - ACTIVITY: Fancy SQL Part I

2. Advanced SQL-izing
   - ACTIVITY: Fancy SQL Part II

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

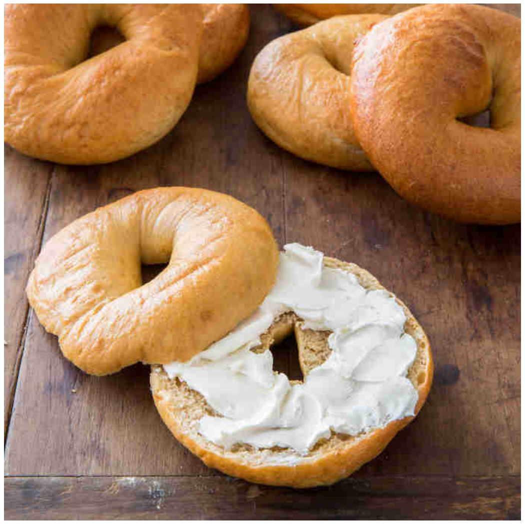
```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

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
```

FROM

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 →

<table>
<tr><th>Product</th><th>Date</th><th>Price</th><th>Quantity</th></tr>
<tr><td rowspan="2">Bagel</td><td>10/21</td><td>1</td><td>20</td></tr>
<tr><td>10/25</td><td>1.50</td><td>20</td></tr>
<tr><td rowspan="2">Banana</td><td>10/3</td><td>0.5</td><td>10</td></tr>
<tr><td>10/10</td><td>1</td><td>10</td></tr>
</table>

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

SELECT →

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
```

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

HAVING clauses contains conditions on **aggregates**

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

General form of Grouping and Aggregation

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

Why?

- 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 C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
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 ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE (
     SELECT Count(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 <u>much</u> more efficient!

Activity-5-1.ipynb

2. 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 (سورها)

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

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

Find all companies that make <u>some</u> products with price < 100

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

Existential: easy ! ☺

سور وجودی: آسان

(وجود دارد)

Quantifiers

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

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

A universal quantifier is of the form "for all"

Find all companies with products all having price < 100

Equivalent

Find all companies that make only products with price < 100

Universal: hard ! ☹

سور عمومی : سخت

)به ازای هر(أ

A little bit of logic

- De Morgan's Laws
  - $\neg(p(a) \wedge p(b)) \Leftrightarrow \neg p(a) \vee \neg p(b)$
  - $\neg(p(a) \vee p(b)) \Leftrightarrow \neg p(a) \wedge \neg p(b)$
- Generalized De Morgan's Laws
  - $\neg(p(a_1) \wedge \ldots \wedge p(a_n)) \Leftrightarrow \neg p(a_1) \vee \ldots \vee \neg p(a_n)$
  - $\neg(p(a_1) \vee \ldots \vee p(a_n)) \Leftrightarrow \neg p(a_1) \wedge \ldots \wedge \neg p(a_n)$,
  - where $\mathbf{A}=\{a_1,\ldots,a_n\}$

A little bit of logic

- Since:
  - $p(a_1) \wedge \ldots \wedge p(a_n) \Leftrightarrow \forall x \in \mathbf{A}\ (p(x))$
  - $p(a_1) \vee \ldots \vee p(a_n) \Leftrightarrow \exists x \in \mathbf{A}\ (p(x))$,
- Generalized De Morgan's Laws become:
  - $\neg(\forall x \in \mathbf{A}\ (p(x))) \Leftrightarrow \exists x \in \mathbf{A}\ (\neg p(x))$
  - $\neg(\exists x \in \mathbf{A}\ (p(x))) \Leftrightarrow \forall x \in \mathbf{A}\ (\neg p(x))$
- and also:
  - $\forall x \in \mathbf{A}\ (p(x)) \Leftrightarrow \neg(\ \exists x \in \mathbf{A}\ (\neg p(x))\ )$
  - $\exists x \in \mathbf{A}\ (p(x)) \Leftrightarrow \neg(\ \forall x \in \mathbf{A}\ (\neg p(x))\ )$

A little bit of logic

- Set inclusion: $\mathbf{A} \subseteq \mathbf{B}$
  - $\forall x \in \mathbf{A}\ (x \in \mathbf{B}) \Leftrightarrow \neg(\exists x \in \mathbf{A}\ (x \notin \mathbf{B})) \Leftrightarrow$
  - $\neg(\exists x \in \mathbf{A}\ \neg(\exists y \in \mathbf{B}\ (x=y)))$
- Set equality: $\mathbf{A=B} \Leftrightarrow \mathbf{A} \subseteq \mathbf{B} \wedge \mathbf{B} \subseteq \mathbf{A}$

- Implication
  - $p(a) \rightarrow p(b) \Leftrightarrow \neg p(a) \vee p(b)$
  - $\neg\ (p(a) \rightarrow p(b)) \Leftrightarrow p(a) \wedge \neg p(b)$

A little bit of logic

- Min aggregate: $a = \min(\mathbf{A})$, where $a \in \mathbf{A}$
    - $\forall x \in \mathbf{A}\ (a \leq x) \Leftrightarrow \neg(\exists x \in \mathbf{A}\ (a > x))$

- Max aggregate: $a = \max(\mathbf{A})$, where $a \in \mathbf{A}$
    - $\forall x \in \mathbf{A}\ (x \leq a) \Leftrightarrow \neg(\exists x \in \mathbf{A}\ (x > a))$

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 = 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= NULL then x="Joe" is UNKNOWN

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- 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

رفتار ناخواسته:

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

بعضی از افراد برگردانده نمیشوند.

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, 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
```

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Note: another equivalent way to write an INNER JOIN!

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-5-2.ipynb

Summary

SQL is a rich programming language
that handles the way data is processed
declaratively