CSC 261/461 – Database Systems
Lecture 3 (Study at Home)

Spring 2017
MW 3:25 pm – 4:40 pm
January 18 – May 3
Dewey 1101

Study at Home

- We will cover this slides in Class
- But, the pace would be faster
- So, please study these slides at home
- Ask question when I present if you have doubt.

Meaning (Semantics) of SQL Queries

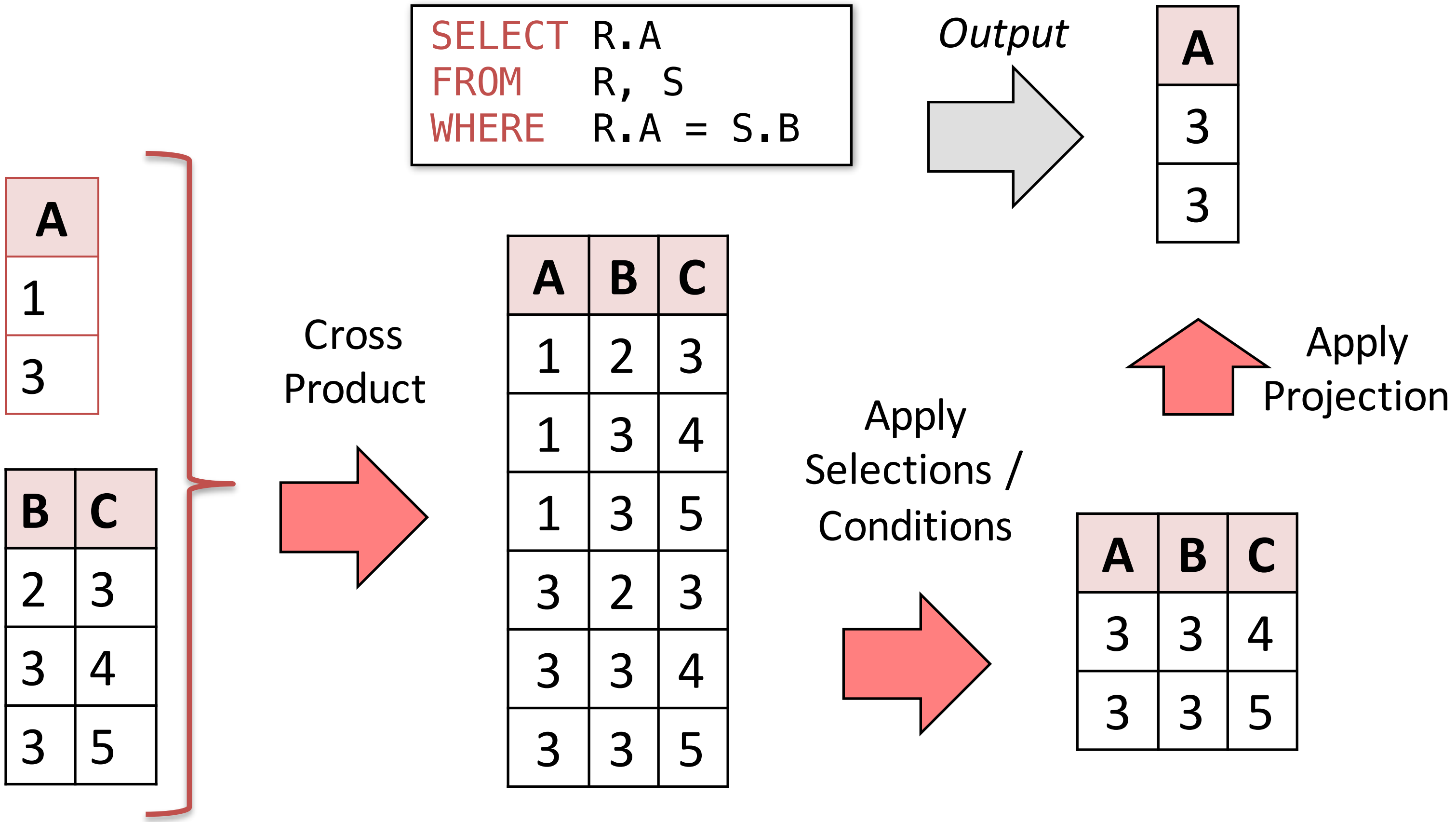
```
SELECT $x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$
FROM   $R_1$ AS $x_1$, $R_2$ AS $x_2$, …, $R_n$ AS $x_n$
WHERE  Conditions($x_1$,…, $x_n$)
```

Almost never the *fastest* way to compute it!

```
Answer = {}
for $x_1$ in $R_1$ do
    for $x_2$ in $R_2$ do
       …..
          for $x_n$ in $R_n$ do
              if Conditions($x_1$,…, $x_n$)
                  then Answer = Answer $\cup$ {($x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$)}
return Answer
```

Note: this is a *multiset* union

An example of SQL semantics



```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

A
1
3

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Output

A
3
3

Note the *semantics* of a join

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

1. Take **cross product**:

$$X = R \times S$$

Recall: Cross product (A X B) is the set of all unique tuples in A,B

Ex: {a,b,c} X {1,2}
= {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

2. Apply **selections / conditions**:

$$Y = \{(r,s) \in X \mid r.A == r.B\}$$

= Filtering!

3. Apply **projections** to get final output:

$$Z = (y.A,) \; for \; y \in Y$$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*

- Not actually how the DBMS executes it under the covers

A Subtlety about Joins

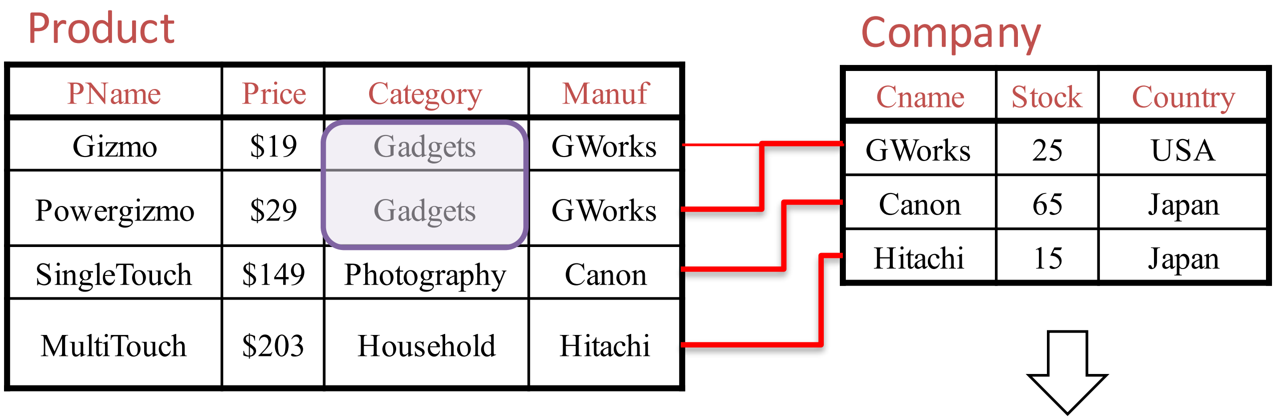
```
Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)
```

Find all countries that manufacture some product in the 'Gadgets' category.

```sql
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=CName AND Category='Gadgets'
```

A subtlety about Joins



Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Country
?
?

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
   AND Category=‘Gadgets’
```

What is the problem ?
What’s the solution ?

1. SET OPERATORS & NESTED QUERIES

What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries
3. ACTIVITY: Set operator subtleties

An Unintuitive Query

TABLE R

A
1
2
3
4
5

TABLE S

A

TABLE T

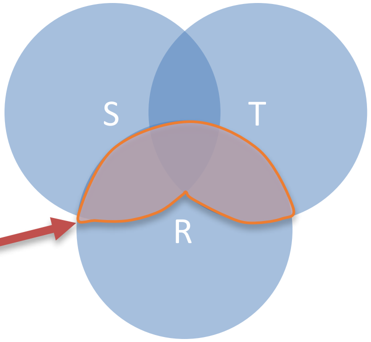
A
1
4
7
10

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



Computes R ∩ (S ∪ T)

But what if S = ϕ?

Go back to the semantics!

An Unintuitive Query

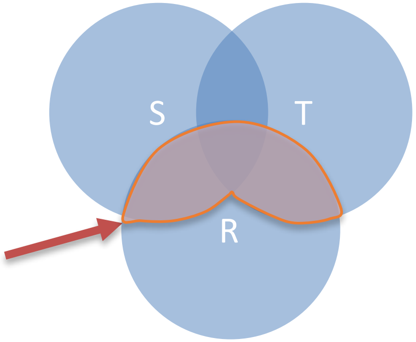
```sql
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

- Recall the semantics!
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection
- If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

Must consider semantics here.
Are there more explicit way to do set operations like this?

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



R ∩ (S ∪ T)

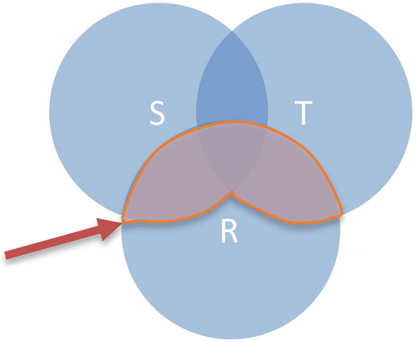
- Semantics:
    1. Take cross-product
    2. Apply selections / conditions
    3. Apply projection

Joins / cross-products are just **nested for loops** (in simplest implementation)!

If-then statements!

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



R ∩ (S ∪ T)

```
output = {}

for r in R:
   for s in S:
      for t in T:
         if r[‘A’] == s[‘A’] or r[‘A’] == t[‘A’]:
            output.add(r[‘A’])
return list(output)
```

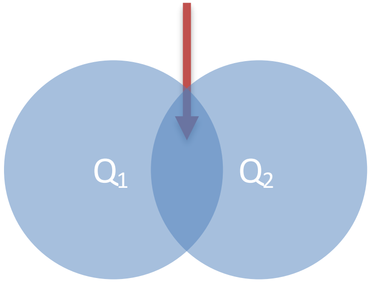
Can you see now what happens if S = []?

MULTISET OPERATIONS IN SQL

Explicit Set Operators: INTERSECT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

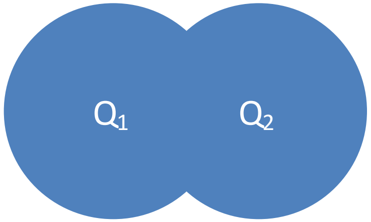
$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$



UNION

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION
SELECT R.A
FROM    R, T
WHERE   R.A=T.A
```

$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$



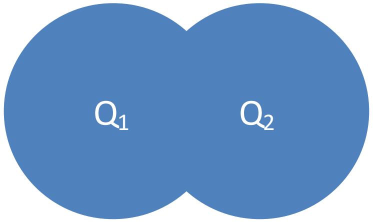
Why aren’t there duplicates?

What if we want duplicates?

UNION ALL

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION ALL
SELECT R.A
FROM    R, T
WHERE   R.A=T.A
```

$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$

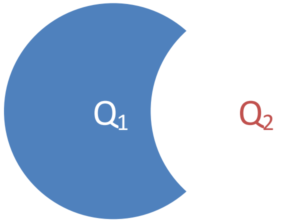


ALL indicates Multiset operations

EXCEPT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$\{r.A \mid r.A = s.A\} \setminus \{r.A | r.A = t.A\}$



Nested queries: Sub-queries Returning Relations

Another
example
:

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT DISTINCT c.city
FROM   Company c
WHERE  c.name  IN (
       SELECT pr.maker
      FROM   Purchase p, Product pr
      WHERE  p.product = pr.name
         AND p.buyer = ‘Joe Blow‘)
```

“Cities where one can find companies that manufacture products bought by Joe Blow”

Subqueries Returning Relations

ANY and ALL not supported by SQLite.

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex:

```
Product(name, price, category, maker)
```

```
SELECT name
FROM   Product
WHERE  price > ALL(
       SELECT price
     FROM   Product
     WHERE  maker = ‘Gizmo–Works’)
```

Find products that are more expensive than all those produced by “Gizmo-Works”

Subqueries Returning Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex:

```
Product(name, price, category, maker)
```

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = ‘Gizmo-Works’
   AND EXISTS(
        SELECT p2.name
      FROM   Product p2
      WHERE  p2.maker <> ‘Gizmo-Works’
           AND p1.name = p2.name)
```

<> means !=

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”

Nested queries as alternatives to INTERSECT and EXCEPT

```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```

⇨

```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
         SELECT *
         FROM S
       WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B
 FROM   R)
EXCEPT
(SELECT S.A, S.B
 FROM   S)
```

⇨

```
SELECT R.A, R.B
FROM   R
WHERE NOT EXISTS(
         SELECT *
       FROM S
       WHERE R.A=S.A AND R.B=S.B)
```

Correlated Queries

```
Movie(title, year, director, length)
```

```
SELECT DISTINCT title
FROM   Movie AS m
WHERE  year <> ANY(
             SELECT  year
       FROM     Movie
       WHERE  title =  m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

Basic SQL Summary

- SQL provides a high-level declarative language for manipulating data (DML)
- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed.

2. AGGREGATION & GROUP BY

What you will learn about in this section

1. Aggregation operators

2. GROUP BY

3. GROUP BY: with HAVING, semantics

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

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

Note: Same as COUNT(*). Why?

We probably want:

```sql
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:

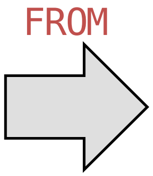
1. Compute the FROM and WHERE clauses

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

SELECT

Product	TotalSales
Bagel	50
Banana	15

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        R_1,…,R_n
WHERE       C_1
GROUP BY    a_1,…,a_k
HAVING      C_2
```

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 $R_1,…,R_n$
2. GROUP BY the attributes $a_1,…,a_k$
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:

```
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!

Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.
- Thanks to YouTube, especially to Dr. Daniel Soper for his useful videos.