

CSC 261/461 – Database Systems

Lecture 3 (Study at Home)

Spring 2018

Study at Home

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

Meaning (Semantics) of SQL Queries

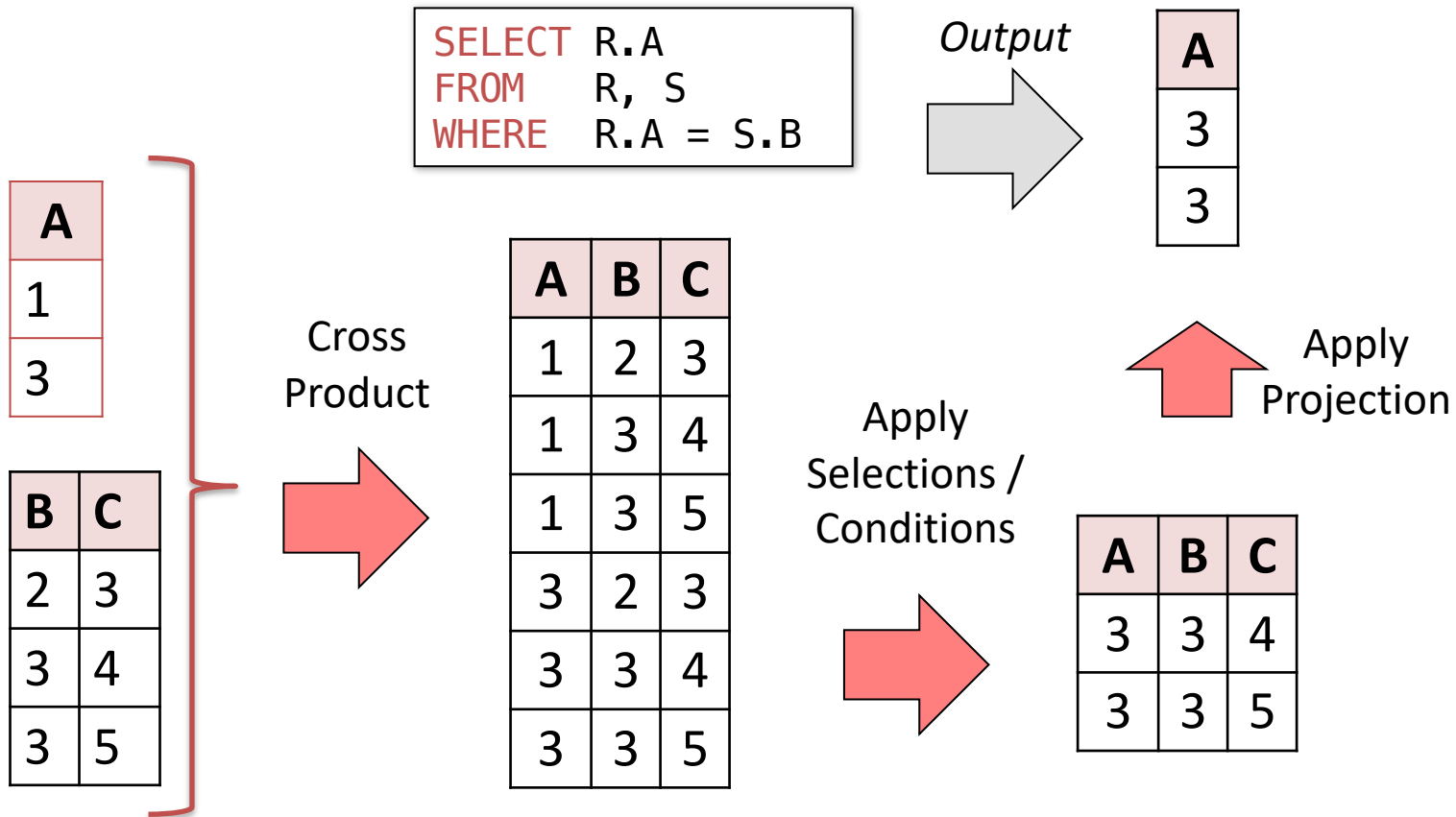
```
SELECT x1.a1, x1.a2, ..., xn.ak  
FROM   R1 AS x1, R2 AS x2, ..., Rn AS xn  
WHERE  Conditions(x1, ..., xn)
```

Almost never the *fastest* way
to compute it!

```
Answer = {}  
for x1 in R1 do  
  for x2 in R2 do  
    ....  
    for xn in Rn do  
      if Conditions(x1, ..., xn)  
        then Answer = Answer  $\cup$  {(x1.a1, x1.a2, ..., xn.ak)}  
return Answer
```

Note: this is a *multiset* union

An example of SQL semantics



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 ($A \times B$) is the set of all unique tuples in A, B

Ex: $\{a, b, c\} \times \{1, 2\}$
 $= \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$

2. Apply **selections / conditions**:

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

=
Filtering!

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

$$Z = (y.A,) \text{ for } y \in Y$$

= 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.

```
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



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

Country
?
?

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

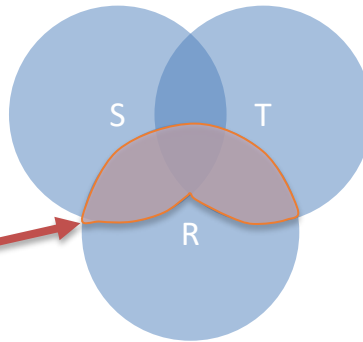
A
1
4
7
10

```
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
 $R \cap (S \cup T)$
Or
 $(R \cap S) \cup (R \cap T)$

But what if $S = \phi$?

Go back to the semantics!

An Unintuitive Query

```
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 = $\{\}$!

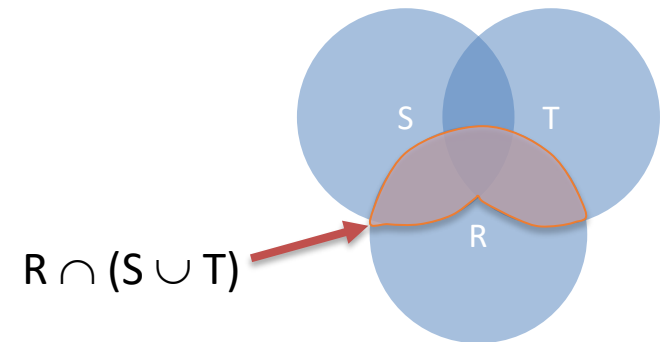
Must consider semantics here.
Are there more explicit way to do set operations like this?

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
```

- Semantics:

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

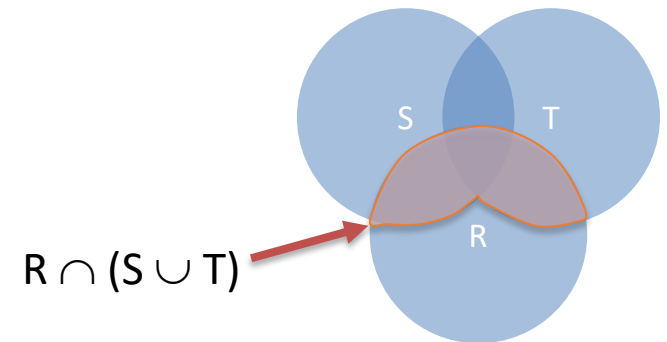


*Joins / cross-products are just **nested for loops** (in simplest implementation)!*

*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
```



```
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)
```

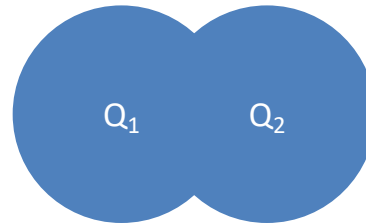
Can you see now what happens if $S = []$?

MULTISET OPERATIONS IN SQL

UNION

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

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



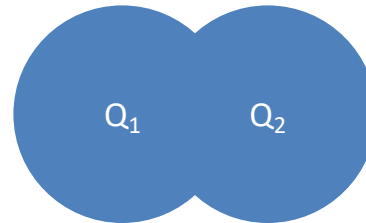
Why aren't there duplicates?

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
```

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

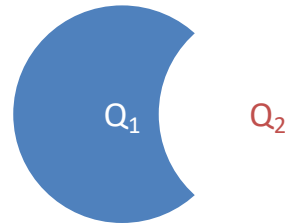


*ALL indicates
Multiset
operations*

EXCEPT

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

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



NESTED QUERIES

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

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\text{EXISTS } R$

ANY and ALL not supported by SQLite.

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:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- EXISTS R

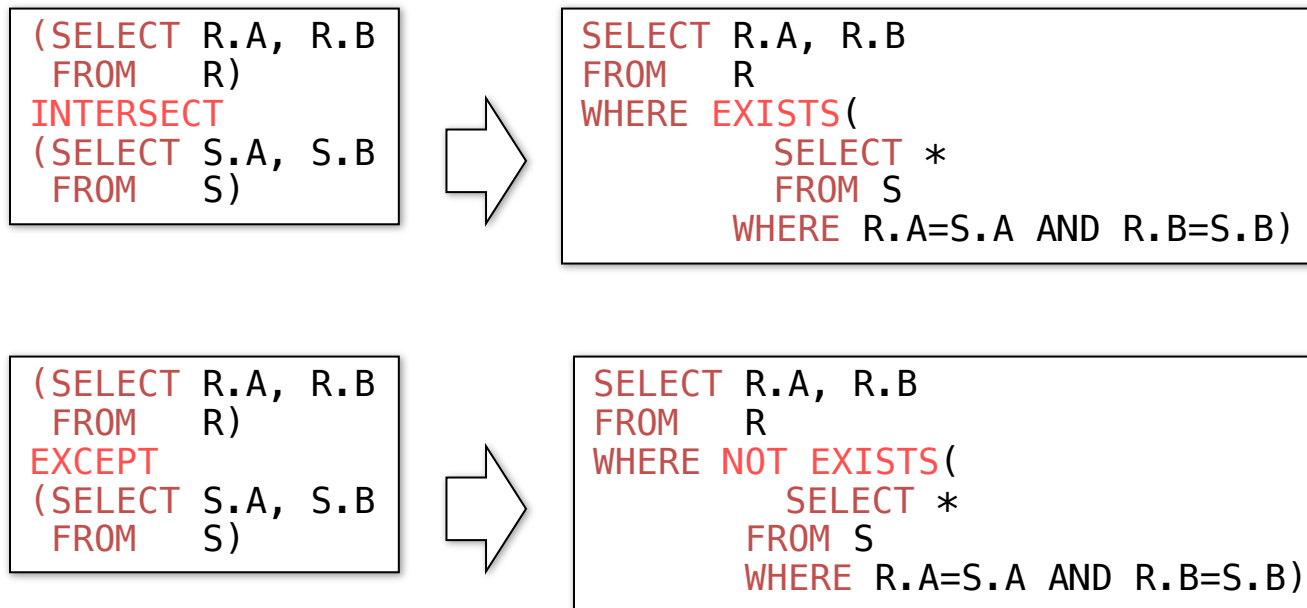
Ex: `Product(name, price, category, maker)`

```
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



MYSQL does not support INTERSECT and EXCEPT operators

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

```
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
```

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

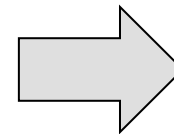
What do these mean?

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...

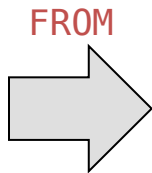
Grouping and Aggregation

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
```



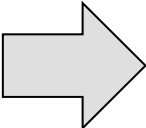
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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



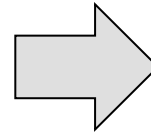
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

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, \dots, R_n
WHERE	C_1
GROUP BY	a_1, \dots, a_k
HAVING	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

SELECT	S
FROM	R_1, \dots, R_n
WHERE	C_1
GROUP BY	a_1, \dots, a_k
HAVING	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 vs. 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.