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
```

Cross Product

Apply Selections / Conditions

Output

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

```
Output
A
3
3
```

```
Apply Projection
```

```
A  B  C
3  3  4
3  3  5
```
Note the **semantics** of a join

**SELECT**  
**FROM**  
**WHERE**

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

1. **Take cross product:**
   
   $$X = R \times S$$

2. **Apply selections / conditions:**
   
   $$Y = \{(r, s) \in X \mid r.A = r.B \}$$

3. **Apply projections** to get final output:
   
   $$Z = (y.A,) \text{ for } y \in Y$$

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

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

Filtering!

= 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

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**

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**Company**

<table>
<thead>
<tr>
<th>Cname</th>
<th>Stock</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

**SQL Query**

```
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

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

What does it compute?

<table>
<thead>
<tr>
<th>TABLE R</th>
<th>TABLE S</th>
<th>TABLE T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A

1

4

7

10
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?

```python
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

\[
\begin{align*}
\text{SELECT} & \quad \text{R.A} \\
\text{FROM} & \quad \text{R, S} \\
\text{WHERE} & \quad \text{R.A} = \text{S.A} \\
\text{UNION ALL} & \\
\text{SELECT} & \quad \text{R.A} \\
\text{FROM} & \quad \text{R, T} \\
\text{WHERE} & \quad \text{R.A} = \text{T.A}
\end{align*}
\]

\[
\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}
\]

ALL indicates Multiset operations
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: Sub-queries Returning Relations

Another example:

```
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:

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

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”

ANY and ALL not supported by SQLite.
Subqueries Returning Relations

You can also use operations of the form:

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

Ex:

```sql
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)
```

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

\( <> \) means \(!=\)
Nested queries as alternatives to INTERSECT and EXCEPT

\[
\text{(SELECT R.A, R.B FROM R) INTERSECT (SELECT S.A, S.B FROM S)}
\]

\[
\text{SELECT R.A, R.B FROM R WHERE EXISTS (}
\text{\quad (SELECT * FROM S WHERE R.A=S.A AND R.B=S.B))}
\]

\[
\text{(SELECT R.A, R.B FROM R) EXCEPT (SELECT S.A, S.B FROM S)}
\]

\[
\text{SELECT R.A, R.B FROM R WHERE NOT EXISTS (}
\text{\quad (SELECT * FROM S WHERE R.A=S.A AND R.B=S.B))}
\]
Correlated Queries

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
1. Aggregation operators

2. GROUP BY

3. GROUP BY: with HAVING, semantics
**Aggregation**

Except COUNT, all aggregations apply to a single attribute.

```
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

• SQL supports several **aggregation** operations:
• 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

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

What do these mean?
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>

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

50 (= 1*20 + 1.50*20)
Select product, SUM(price * quantity) AS TotalSales from Purchase where date > '10/1/2005' group by 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
```

<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**

```
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**
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>
HAVING Clause

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

```
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

Whereas WHERE clauses condition on individual tuples...
General form of Grouping and Aggregation

- $S = \text{Can ONLY contain attributes } a_1, \ldots, a_k \text{ and/or aggregates over other attributes}$
- $C_1 = \text{is any condition on the attributes in } R_1, \ldots, R_n$
- $C_2 = \text{is any condition on the aggregate expressions}$
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 vs. Nested Query

- Find authors who wrote ≥ 10 documents:

```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)

```sql
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!
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.