CSC 261/461 – Database Systems
Lecture 5

Fall 2017
MULTISET OPERATIONS IN SQL
UNION

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

\{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?
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**
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:

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

Ex:

\[
\text{Product}(\text{name}, \text{price}, \text{category}, \text{maker})
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{name} \\
\text{FROM} & \quad \text{Product} \\
\text{WHERE} & \quad \text{price} > \text{ALL}( \\
 & \quad \text{SELECT} \quad \text{price} \\
 & \quad \text{FROM} \quad \text{Product} \\
 & \quad \text{WHERE} \quad \text{maker} = 'Gizmo-Works')
\end{align*}
\]

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: \( \text{Product(name, price, category, maker)} \)

```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”
Nested queries as alternatives to INTERSECT and EXCEPT
Correlated Queries

```sql
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

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

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

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

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
```
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)
Grouping and Aggregation

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

Let’s see what this means...

Find total sales after 10/1/2005 per product.
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

```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>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**

```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>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>

<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>
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

- $S$ = 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

```
SELECT S
FROM $R_1, \ldots, R_n$
WHERE $C_1$
GROUP BY $a_1, \ldots, a_k$
HAVING $C_2$
```
General form of Grouping and Aggregation

\[
\begin{align*}
\text{SELECT} & \quad S \\
\text{FROM} & \quad R_1, \ldots, R_n \\
\text{WHERE} & \quad C_1 \\
\text{GROUP BY} & \quad a_1, \ldots, a_k \\
\text{HAVING} & \quad C_2
\end{align*}
\]

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
• 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
```
Group-by vs. 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
```

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!
Topics Covered

1. NULLs
2. Outer Joins
3. With and Case
4. Constraint
5. Schema Change Statements
## Table 7.1 Logical Connectives in Three-Valued Logic

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>TRUE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td></td>
</tr>
</tbody>
</table>
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL

• Can mean many things:
  – Value does not exist
  – Value exists but is unknown
  – Value not applicable
  – Etc.

• The schema specifies for each attribute if can be null (nullable attribute) or not

• Each individual NULL value considered to be different from every other NULL value

• SQL uses a three-valued logic:
  – TRUE, FALSE, and UNKNOWN (like Maybe)

• NULL = NULL comparison is avoided

• How does SQL cope with tables that have NULLs?
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`

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

Now it includes all Persons!
By default, joins in SQL are "inner joins":

```
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!
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!
• 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:

```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

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

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

Note: another equivalent way to write an INNER JOIN!
### LEFT OUTER JOIN:

#### Product

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

#### Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

**SQL Query:**

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