Announcements

Special Lecture
- At Sieg 134 on January 19th from 330-450PM
- Topic: Building SQL Applications
- Important For
  - Programming Assignment
  - Course Project
- Form Groups for Course Project NOW
- Homework Due in a week
- Final: Check Schedule

SQL

Reading: Sec 5 (all subsections, except 5.10)

Selection and Projection

SELECT name, stockPrice
FROM Company
WHERE country like "USA" AND stockPrice > 50

Input schema: Company(sticker, name, country, stockPrice)
Output schema: R(name, stock price)

Removing Duplicates

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

SELECT DISTINCT category
FROM Product
WHERE price > 100

Simple Aggregation

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database
SELECT Sum(price * quantity)
FROM Purchase

Example 1': find total sales of bagels
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
Grouping, Aggregation

Purchase(product, date, price, quantity)

Example 2: find total sales after 9/1 per product.

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUPBY product

First compute the relation (date > “9/1”) then group by product:

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
</tbody>
</table>

Then, aggregate

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>$29.75</td>
</tr>
<tr>
<td>Banana</td>
<td>$12.48</td>
</tr>
</tbody>
</table>

Then compute

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUPBY product

Example

<table>
<thead>
<tr>
<th>Product</th>
<th>SumSales</th>
<th>MaxQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$12.48</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>$29.75</td>
<td>20</td>
</tr>
</tbody>
</table>

For every product, what is the total sales and max quantity sold?

SELECT product, Sum(price * quantity) AS SumSales
Max(quantity) AS MaxQuantity
FROM Purchase
GROUP BY product

Group By and Having

Purchase(product, date, price, quantity)

Example 2: find total sales after 9/1 per product.

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUPBY product
HAVING Sum(quantity) > 10

Queries With GROUP BY and HAVING

SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification

Note: The target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).

Note: The attribute list (i) must be a subset of grouping-list.

Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)
Conceptual Evaluation

The cross-product of relation-list is computed, tuples that fail qualification are discarded, 'unnecessary' fields are deleted, as before.

The remaining tuples are partitioned into groups by the value of attributes in grouping-list.

The group-qualification is then applied to eliminate some groups.

One answer tuple is generated per qualifying group.

Find the age of the youngest sailor with age 18, for each rating with at least 2 such sailors

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>Brunas</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
```

Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.

2nd column of result is unnamed. (Use AS to name it.)

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Meaning (Semantics) of SQL Queries

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

4. Translation to Relational algebra:

\[ \Pi_{a1,...,ak} (\sigma_{Conditions} (R1 \times R2 \times ... \times Rn)) \]

Select-From-Where queries are precisely Select-Project-Join

Joins

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Company (cname, stockPrice, country)
Person (per-name, phoneNumber, city)

Find names of people living in Seattle that bought gizmo products, and the names of the stores they bought from

```
SELECT per-name, store
FROM Person, Purchase
WHERE per-name=buyer AND city="Seattle"
AND product="gizmo"
```

Conceptual Evaluation Strategy

Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

1. Compute the cross-product of relation-list.
2. Discard resulting tuples if they fail qualifications.
3. Delete attributes that are not in target-list.
4. If DISTINCT is specified, eliminate duplicate rows.

This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

Meaning (Semantics) of SQL Queries

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

1. Nested loops:
   \[ \text{Answer} = \{} \]
   \[ \text{for } x1 \text{ in } R1 \text{ do} \]
   \[ \text{for } x2 \text{ in } R2 \text{ do} \]
   \[ \quad \text{......} \]
   \[ \text{for } xn \text{ in } Rn \text{ do} \]
   \[ \quad \text{if } \text{Conditions} \]
   \[ \quad \text{then } \text{Answer} = \text{Answer} \cup \{ (a1, ..., ak) \} \]
   \[ \text{return } \text{Answer} \]
```
Example Instances

<table>
<thead>
<tr>
<th>R1</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
</tbody>
</table>

% We will use these instances of the Sailors and Reserves relations in our examples.

<table>
<thead>
<tr>
<th>S1</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S2</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

Example of Conceptual Evaluation

```
SELECT S.sname
FROM Sailors S1, Reserves R1
WHERE S1.sid=R1.sid AND R1.bid=103
```

A Note on Range Variables

% Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

OR

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid
AND bid=103
```

It is good style, however, to use range variables always!

Find sailors who've reserved at least one boat

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

% Would adding DISTINCT to this query make a difference?

SQL is Tricky!

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

Looking for \( R \cap (S \cup T) \)

But what happens if \( T \) is empty?

Nested Queries

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
FROM Reserves R
WHERE R.bid=103)
```

% A WHERE clause can itself contain an SQL query!

% To find sailors who've not reserved #103, use NOT IN.

% To understand semantics of nested queries, think of a nested loops evaluation: For each Sailors tuple, check the qualification by computing the subquery.
**Nested Queries with Correlation**

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

*EXISTS is another set comparison operator, like IN.
*If UNIQUE is used, and * is replaced by R.bid, finds sailors
  with at most one reservation for boat #103. (UNIQUE checks
  for duplicate tuples; * denotes all attributes. Why do we
  have to replace * by R.bid?)*
*Illustrates why, in general, subquery must be re-computed for
each Sailors tuple.*

**Example: Subqueries Returning Relations**

Find companies who manufacture products bought by Joe Blow.

```
SELECT Company.name
FROM Company, Product
WHERE Company.name=maker
AND Product.name IN
(SELECT product
FROM Purchase
WHERE buyer = "Joe Blow");
```

Here the subquery returns a set of values

**Example: Subqueries Returning Relations**

```
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name=maker
AND Product.name = product
AND buyer = "Joe Blow"
```

Equivalent to:

```
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name=maker
AND Product.name = product
AND buyer = "Joe Blow"
```

Is this query equivalent to the previous one?

**Example: Subqueries Returning Relations**

```
SELECT name
FROM Product
WHERE price > ALL (SELECT price
FROM Purchase
WHERE maker=“Gizmo-Works”)
```

Product ( pname, price, category, maker)
Find products that are more expensive than all those produced
By “Gizmo-Works”

```
SELECT name
FROM Product
WHERE price > ALL (SELECT price
FROM Purchase
WHERE maker=“Gizmo-Works”)
```

You can also use:  
\( x > \text{ALL} R \)
\( x > \text{ANY} R \)
\( \text{EXISTS}\ R \)

**Example: Conditions on Tuples**

```
SELECT Company.name
FROM Company, Product
WHERE Company.name=maker
AND (Product.name,price) IN
(SELECT product, price
FROM Purchase
WHERE buyer = “Joe Blow”)
```

```
SELECT Company.name
FROM Company, Product
WHERE Company.name=maker
AND (Product.name,price) IN
(SELECT product, price
FROM Purchase
WHERE buyer = “Joe Blow”)
```

You can also use:  
\( x > \text{ALL} R \)
\( x > \text{ANY} R \)
\( \text{EXISTS}\ R \)
Example: Correlated Queries

Movie (title, year, director, length)
Find movies whose title appears more than once.

```
SELECT title
FROM Movie AS x
WHERE year < ANY
  (SELECT year
   FROM Movie
   WHERE title = x.title);
```

Note (1) scope of variables (2) this can still be expressed as single SFW

Example: Complex Correlated Query

Product (pname, price, category, maker, year)
Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT pname, maker
FROM Product AS x
WHERE price > ALL (SELECT price
  FROM Product AS y
  WHERE x.maker = y.maker AND y.year < 1972);
```

Example: Removing Duplicates

```
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name=maker
  AND (Product.name,price) IN
    (SELECT product, price
     FROM Purchase
     WHERE buyer = "Joe Blow");
```

Example: Union, Intersection, Difference

```
(SELECT name
  FROM Person
  WHERE City="Seattle")
UNION

(SELECT name
  FROM Person, Purchase
  WHERE buyer=name AND store="The Bon")
```

Similarly, you can use INTERSECT and EXCEPT. You must have the same attribute names (otherwise: rename).

Find sid's of sailors who've reserved a red or a green boat

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND (B.color='red' OR B.color='green')
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='green'
```

Union All Etc.

The UNION, INTERSECTION and EXCEPT operators operate as sets, not bags.

```
(SELECT name
  FROM Person
  WHERE City="Seattle")
UNION ALL

(SELECT name
  FROM Person, Purchase
  WHERE buyer=name AND store="The Bon")
```
**Defining Views**

Views are relations, except that they are not physically stored.

They are used mostly in order to simplify complex queries and to define conceptually different views of the database to different classes of users.

View: purchases of telephony products:

```
CREATE VIEW telephony-purchases AS
SELECT product, buyer, seller, store
FROM Purchase, Product
WHERE Purchase.product = Product.name
AND Product.category = "telephony"
```

**A Different View**

```
CREATE VIEW Seattle-view AS
SELECT buyer, seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND
Person.name = Purchase.buyer
```

We can later use the views:

```
SELECT name, store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
Product.category = "shoes"
```

What’s really happening when we query a view??

**Null Values and Outerjoins**

```
If x=Null then 4*(3-x)/7 is still NULL
```

```
If x=Null then x="Joe" is UNKNOWN
```

Three boolean values:

- FALSE = 0
- UNKNOWN = 0.5
- TRUE = 1

```
SELECT *
FROM Person
WHERE (age < 25) AND
(height > 6 OR weight > 190)
```

Rule in SQL: include only tuples that yield TRUE.

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

Some Persons are not included!
Null Values and Outerjoins

Can test for NULL explicitly:

\[ \exists x \text{ IS NULL} \]
\[ \exists x \text{ IS NOT NULL} \]

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

Now it includes all Persons

---

Null Values and Outerjoins

Explicit joins in SQL:

```sql
Product(name, category)
Purchase(prodName, store)
```

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

Same as:

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

But Products that never sold will be lost!

---

Null Values and Outerjoins

Left outer joins in SQL:

```
Product(name, category)
Purchase(prodName, store)
```

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

---

Null Values and Outerjoins

```
<table>
<thead>
<tr>
<th>Product</th>
<th>Category</th>
<th>Purchase</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gismo</td>
<td>gadget</td>
<td>Gismo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>
```

---

Null Values and Outerjoins

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gismo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>-</td>
</tr>
</tbody>
</table>
```