SQL: Data Manipulation Language (DML)

Todd Bacastow
IST 210: Organization of Data
SQL

- SQL = Structured Query Language
  - Data Definition Language (DDL)
  - Data Manipulation Language (DML)
Think in terms of sets and not in terms of "one record at a time" processing

Ask set-oriented questions

For example: "What is in the set of students?"
Principal Form

**SELECT** desired attributes

**FROM** relations (formally, *tuple variables*)

**WHERE** condition about tuples in the relations
Example

- What beers are made by A.B.

Beers (name, addr)

```
SELECT name
FROM Beers
WHERE manf = 'A.B.';
```

Note single quotes for strings.
Use * As List of All Attributes

Beers (name, manf)

SELECT *
FROM Beers
WHERE manf = 'A.B.';

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>A.B.</td>
</tr>
<tr>
<td>BudLite</td>
<td>A.B.</td>
</tr>
<tr>
<td>Michelob</td>
<td>A.B.</td>
</tr>
</tbody>
</table>
Notes

- Condition in WHERE clause can use logical operators AND, OR, NOT and parentheses in the usual way
- Remember: SQL is case insensitive. Keywords like SELECT or AND can be written upper/lower case as you like
- Only inside quoted strings does case matter
  - ‘Joe Johns’ != ‘joe johns’ != ‘Joe JOHNS’
Relational Operators

There are six Relational Operators in SQL:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>Not Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal To</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal To</td>
</tr>
</tbody>
</table>
Multirelation Queries

- List of relations in FROM clause
- Relation-dot-attribute disambiguates attributes from several relations
- Example:
  - Find the beers that the frequenters of Joe’s Bar like.

```
Likes (drinker, beer)
Frequents (drinker, bar)

SELECT beer
FROM Frequents, Likes
WHERE bar='Joe''s' AND Frequent.drinker=Likes.drinker;
```
Multirelation Query Semantics

- Same as for single relation, but start with the product of all the relations mentioned in the FROM clause
- Operational Semantics
  - Consider a tuple variable for each relation in the FROM
  - Imagine these tuple variables each pointing to a tuple of their relation, in all combinations (i.e., a cross product)
  - If the current assignment of tuple-variables to tuples makes the WHERE true, then output the attributes of the SELECT.
Explicit Tuple Variables

Sometimes we need to refer to two or more copies of a relation

- Use tuple variables as aliases of the relations
  - E.g. Find pairs of beers by the same manufacturer
    Beers (name, manf)

    SELECT b1.name, b2.name
    FROM Beers AS b1, Beers AS b2
    WHERE b1.manf=b2.manf AND b1.name<b2.name;

- Note: b1.name<b2.name is needed to avoid producing (Bud,Bud) and to avoid producing in both orders.
Subqueries

- Result of a SELECT-FROM-WHERE query can be used in the WHERE-clause of another query
- Simplest case: subquery returns a single, unary tuple
  - E.g. Find bars that serve Miller at the same price Joe charges for Bud

Sells (bar, beer, price)

SELECT bar
FROM Sells
WHERE beer='Miller' AND price=
(SELECT price
FROM Sells
WHERE bar='Joe”s’ AND beer='Bud');
Subqueries

- **Scoping rule:** an attribute refers to the most closely nested relation with that attribute
- **Parentheses rule:** parentheses around subquery are essential
Union, Intersection, Difference

- “Relation UNION relation” produces the union of the two relations
- Similarly for INTERSECTION, EXCEPT = intersection and set difference
- Deference renders the difference between two relations
Bag Semantics of SQL

- An SQL relation is really a *bag* or *multiset*
  - It may contain the same tuple more than once, although there is no specified order
  - Example: \{1,4,1,3\} is a bag and not a set
    \{1,4,3\} is a set

- Different implementations of SQL handle this in different ways!
Defaults for SQL

- Default for SELECT-FROM-WHERE is **bag**
- Default for UNION, INTERSECTION, and DIFFERENCE is **set**
- Force set semantics with DISTINCT after SELECT
  - Example: Find the different prices charged for beers
    Sells (bar, beer, price)

    ```sql
    SELECT DISTINCT price FROM Sells;
    ```

- Force bag semantics with ALL after UNION, etc.
More on Bag Semantics

- **Bag**
  - **Union**: collective of the element in the two bags
    - Example: \(\{1,2,1\} \cup \{1,2,3\} = \{1,1,1,2,2,3\}\)
  - **Intersection**: take the minimum of the number of occurrences in each bag
    - Example: \(\{1,2,1\} \cap \{1,2,3,3\} = \{1,2\}\)
  - **Difference**: proper-subtract the number of occurrences in the two bags
    - Example: \(\{1,2,1\} - \{1,2,3,3\} = \{1\}\)

- **NOTE**: Frequently the default for UNION, INTERSECTION, and DIFFERENCE is **set**
Example

- Find the drinkers and beers such that the drinker likes the beer and frequents a bar that serves it

  Likes (drinker, beer)
  Sells (bar, beer, price)
  Frequents (drinker, bar)

  Likes INTERSECT
  (SELECT drinker, beer
   FROM Sells, Frequents
   WHERE Frequents.bar = Sells.bar);
Aggregations

- SUM, AVG, MIN, MAX, and COUNT apply to attributes/columns.
- COUNT(*) also applies to tuples
- Use these in lists following SELECT
- Example: Find the average price of Bud
  Sells (bar, beer, price)

SELECT AVG (price)
FROM Sells
WHERE beer = 'Bud';

Counts each tuple once (including possible duplicates)
Eliminating Duplicates before Aggregation

- Find the number of different prices at which Bud is sold

Sells (bar, beer, price)

SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud';

- DISTINCT may be used in any aggregation, but typically only make sense with COUNT
Grouping

- Follow SELECT-FROM-WHERE by GROUP BY and a list of attributes.
- The relation that is the result of the FROM and WHERE clauses is grouped according to the values of these attributes, and aggregations take place only within a group.
- Example: find the average sales price for each beer.

Sells (bar, beer, price)

SELECT beer, AVG(price) FROM Sells
GROUP BY beer;
Simple Join

Consider:

Likes (drinker, beer)
Sells (bar, beer, price)

SELECT drinker, bar
FROM Likes, Sells
WHERE Likes.beer=Sells.beer;

- What’s happening here????
Other ‘Joins’ in SQL2

- R NATURAL JOIN S
- R CROSS JOIN S
- R JOIN S ON condition
- R OUTER JOIN S
Alternative JOIN Constructs

- SQL2 provides alternative ways to specify joins:

  FROM Likes.beer JOIN Sells.beer ON Likes.beer=Sells.beer

  FROM Likes JOIN Sells USING beer

  FROM Likes NATURAL JOIN Sells

- The first produces a table with two identical beer columns, remaining two produce table with single beer column.
Outerjoin

- Outerjoin = natural join with *dangling* tuples padded with NULLs and included in the result
- A tuple is dangling if it doesn’t join with any other tuple

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

OUTERJOIN

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

=  

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Modifiers of OUTERJOIN

- Optional NATURAL in front
- Optional ON condition at end
- Optional LEFT, RIGHT, FULL before OUTER
  - LEFT = Pad dangling tuples of R ONLY
  - RIGHT = Pad dangling tuples of S ONLY
## Other Examples

### Employees:

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Hansen, Ola</td>
</tr>
<tr>
<td>02</td>
<td>Svendson, Tove</td>
</tr>
<tr>
<td>03</td>
<td>Svendson, Stephen</td>
</tr>
<tr>
<td>04</td>
<td>Pettersen, Kari</td>
</tr>
</tbody>
</table>

### Orders:

<table>
<thead>
<tr>
<th>Prod_ID</th>
<th>Product</th>
<th>Employee_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>Printer</td>
<td>01</td>
</tr>
<tr>
<td>657</td>
<td>Table</td>
<td>03</td>
</tr>
<tr>
<td>865</td>
<td>Chair</td>
<td>03</td>
</tr>
</tbody>
</table>
Who has ordered a product, and what did they order?

SELECT Employees.Name, Orders.Product
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID

Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hansen, Ola</td>
<td>Printer</td>
</tr>
<tr>
<td>Svendson, Stephen</td>
<td>Table</td>
</tr>
<tr>
<td>Svendson, Stephen</td>
<td>Chair</td>
</tr>
</tbody>
</table>
Who ordered a printer?

```
SELECT Employees.Name
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID
AND Orders.Product='Printer'
```

Result

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hansen, Ola</td>
</tr>
</tbody>
</table>
Who has ordered a product, and what did they order?

```sql
SELECT Employees.Name, Orders.Product
FROM Employees
INNER JOIN Orders ON Employees.Employee_ID=Orders.Employee_ID
```

The INNER JOIN returns all rows from both tables where there is a match. If there are rows in Employees that do not have matches in Orders, those rows will **not** be listed.

**Result**

<table>
<thead>
<tr>
<th>Name</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hansen, Ola</td>
<td>Printer</td>
</tr>
<tr>
<td>Svendson, Stephen</td>
<td>Table</td>
</tr>
<tr>
<td>Svendson, Stephen</td>
<td>Chair</td>
</tr>
</tbody>
</table>