Introduction to SQL

Multirelation Queries
Subqueries

Slides are reused by the approval of Jeffrey Ullman’s
Our Running Example

All our SQL queries will be based on the following database schema.
- Underline indicates key attributes.

- **Beers**(*name*, *manf*)
- **Bars**(*name*, *addr*, *license*)
- **Drinkers**(*name*, *addr*, *phone*)
- **Likes**(*drinker*, *beer*)
- **Sells**(*bar*, *beer*, *price*)
- **Frequents**(*drinker*, *bar*)
Bag Semantics

♦ Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is set semantics.
  ♦ That is, duplicates are eliminated as the operation is applied.
Motivation: Efficiency

◆ When doing projection, it is easier to avoid eliminating duplicates.
  ♦ Just work tuple-at-a-time.

◆ For intersection or difference, it is most efficient to sort the relations first.
  ♦ At that point you may as well eliminate the duplicates anyway.
Controlling Duplicate Elimination

- Force the result to be a set by `SELECT DISTINCT . . .`
- Force the result to be a bag (i.e., don’t eliminate duplicates) by `ALL`, as in `... UNION ALL ...`
Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(*) counts the number of tuples.
Example: Aggregation

From `Sells(bar, beer, price)`, find the average price of Bud:

```
SELECT AVG(price)
FROM Sells
WHERE beer = 'Bud';
```
Eliminating Duplicates in an Aggregation

◆ Use DISTINCT inside an aggregation.
◆ Example: find the number of different prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud';
```
NULL’s Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.
  - Exception: COUNT of an empty set is 0.
Example: Effect of NULL’s

SELECT count(*)
FROM Sells
WHERE beer = 'Bud';

The number of bars that sell Bud.

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

The number of bars that sell Bud at a known price.
Grouping

◆ We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.

◆ The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.
Example: Grouping

◆ From `Sells(bar, beer, price)`, find the average price for each beer:

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

<table>
<thead>
<tr>
<th>beer</th>
<th>AVG(price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>2.33</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: Grouping

From `Sells(bar, beer, price)` and `Frequents(drinker, bar)`, find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'Bud' AND Frequents.bar = Sells.bar
GROUP BY drinker;
```

Compute all drinker-bar-price triples for Bud. Then group them by drinker.
Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.
HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.
Example: HAVING

From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete’s.
Solution

SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR beer IN (SELECT name
FROM Beers
WHERE manf = 'Pete''s');

Beer groups with at least 3 non-NULL bars and also beer groups where the manufacturer is Pete’s.

Beers manufactured by Pete’s.
Requirements on HAVING Conditions

- Anything goes in a subquery.
- Outside subqueries, they may refer to attributes only if they are either:
  1. A grouping attribute, or
  2. Aggregated

(same condition as for SELECT clauses with aggregation).
Database Modifications

- A modification command does not return a result (as a query does), but changes the database in some way.
- Three kinds of modifications:
  1. Insert a tuple or tuples.
  2. Delete a tuple or tuples.
  3. Update the value(s) of an existing tuple or tuples.
Insertion

◆ To insert a single tuple:

    INSERT INTO <relation>
    VALUES ( <list of values> );

◆ Example: add to Likes(drinker, beer) the fact that Sally likes Bud.

    INSERT INTO Likes
    VALUES('Sally', 'Bud');
Specifying Attributes in INSERT

- We may add to the relation name a list of attributes.

- Two reasons to do so:
  1. We forget the standard order of attributes for the relation.
  2. We don’t have values for all attributes, and we want the system to fill in missing components with NULL or a default value.
Example: Specifying Attributes

Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```sql
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```
Adding Default Values

- In a CREATE TABLE statement, we can follow an attribute by DEFAULT and a value.
- When an inserted tuple has no value for that attribute, the default will be used.
Example: Default Values

CREATE TABLE Drinkers (  
  name CHAR(30) PRIMARY KEY,  
  addr CHAR(50)  
    DEFAULT '123 Sesame St.',  
  phone CHAR(16)  
);
Example: Default Values

```
INSERT INTO Drinkers(name) VALUES('Sally');
```

Resulting tuple:

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>123 Sesame St</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Inserting Many Tuples

We may insert the entire result of a query into a relation, using the form:

\[
\text{INSERT INTO } \langle \text{relation} \rangle \\
( \langle \text{subquery} \rangle )
\]
Example: Insert a Subquery

Using `Frequents(drinker, bar)`, enter into the new relation `PotBuddies(name)` all of Sally’s “potential buddies,” i.e., those drinkers who frequent at least one bar that Sally also frequents.
Solution

```
INSERT INTO PotBuddies
(SELECT d2.drinker
FROM Frequent d1, Frequent d2
WHERE d1.drinker = 'Sally' AND d2.drinker <> 'Sally' AND d1.bar = d2.bar
);
```

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same.
Deletion

To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>
WHERE <condition>;
```
Example: Deletion

- Delete from \texttt{Likes}((\texttt{drinker, beer}) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND beer = 'Bud';
```
Example: Delete all Tuples

♥ Make the relation Likes empty:

DELETE FROM Likes;

♥ Note no WHERE clause needed.
Example: Delete Some Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS ( 
  SELECT name FROM Beers
  WHERE manf = b.manf AND
  name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.
Semantics of Deletion --- (1)

Suppose Anheuser-Busch makes only Bud and Bud Lite.

Suppose we come to the tuple $b$ for Bud first.

The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.

Now, when $b$ is the tuple for Bud Lite, do we delete that tuple too?
Semantics of Deletion --- (2)

Answer: we *do* delete Bud Lite as well.

The reason is that deletion proceeds in two stages:

1. Mark all tuples for which the WHERE condition is satisfied.
2. Delete the marked tuples.
Updates

- To change certain attributes in certain tuples of a relation:

  UPDATE <relation>
  SET <list of attribute assignments>
  WHERE <condition on tuples>;
Example: Update

Change drinker Fred’s phone number to 555-1212:

```
UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```
Example: Update Several Tuples

Make $4 the maximum price for beer:

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```