Announcements

- Assignment 2 is due now!
- Assignment 3 is due next Tuesday!
- Start early!
- Midterm in class on November 1.
  - in class; open book/notes.

Outline

- Multirelation SQL queries
- Subqueries
  - ANY, ALL, EXISTS, IN
- Aggregation
  - GROUP BY
  - HAVING

Multirelation Queries

- List of relations in FROM clause.
- Relation-dot-attribute disambiguates attributes from several relations.
- Example: Find the beers that the frequenters of Spoon like.
  - \( \text{Likes}(\text{drinker, beer}) \text{ Frequents}(\text{drinker, bar}) \)

```sql
SELECT beer
FROM Frequents, Likes
WHERE bar = 'Spoon' AND Frequents.drinker = Likes.drinker;
```

Formal Semantics

- Same as for single relation, but start with the product of all the relations mentioned in the FROM clause:
  - Apply selection (for bags) – WHERE clause
  - Apply projection (extended) – SELECT 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 (e.g., nested loops).
- 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.
- Example: Find pairs of beers by the same manufacturer.

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

Explicit Tuple Variables

- SQL permits AS between relation and its tuple variable
- Note that b1.name < b2.name is needed to avoid producing (Bud, Bud) and to avoid producing a pair in both orders.

Examples

- Find all bars that sell two different beers at the same price.
- Find all bars that sell three different beers at the same price.
- Find all drinkers that frequent a bar that serves their favorite beer.

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 (like a constant).

Example

- Find bars that serve Miller at the same price Spoon charges for Bud

```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND price = (SELECT price
FROM Sells
WHERE bar = 'Spoon' AND beer = 'Bud');
```

The IN Operator

- Tuple IN relation is true iff the tuple is in the relation.
- Find the name and manufacturer of beers that Leo likes

```
Beers(name, manf) and Likes(drinker, beer).
```

```
SELECT *
FROM Beers
WHERE name IN (SELECT beer
FROM Likes
WHERE drinker = 'Leo');
```
The EXISTS operator

- \( \text{EXISTS}(\text{relation}) \) is true iff the relation is nonempty.
- Find the beers that are the unique beer by their manufacturer:
  
  ```sql
  SELECT name
  FROM Beers b1
  WHERE NOT EXISTS
    (SELECT *
     FROM Beers
     WHERE manf = b1.manf AND
     name <> b1.name);
  ```

Correlated Subquery

- Scoping rule: to refer to outer \( \text{Beers} \) in the inner subquery, we need to give the outer a tuple variable, \( b1 \) in this example.
- A subquery that refers to values from a surrounding query is called a correlated subquery.
- A correlated subquery must be evaluated (by the system) for every tuple in the outer query.

Quantifiers

- ANY and ALL behave as existential and universal quantifiers, respectively.
- Find the beer(s) sold for the highest price, given \( \text{Sells} \)
  
  ```sql
  SELECT beer
  FROM Sells
  WHERE price >= ALL
    (SELECT price
     FROM Sells);
  ```

Example

- Find the beer(s) not sold for the lowest price, given \( \text{Sells} \)
  
  ```sql
  (SELECT * FROM Likes)
  INTERSECT
  (SELECT drinker, beer
   FROM Sells, Frequents
   WHERE Frequents.bar = Sells.bar);
  ```

Union, Intersection, Difference

- \( (\text{subquery}) \ \text{UNION} \ (\text{subquery}) \) produces the union of the two relations.
- Similarly for INTERSECT, EXCEPT = intersection and set difference.
- No supported by MySQL but you can write an equivalent query.

Example

- Find the drinkers and beers such that the drinker likes the beer and frequents a bar that serves it.
  
  ```sql
  (SELECT * FROM Likes)
  INTERSECT
  (SELECT drinker, beer
   FROM Sells, Frequents
   WHERE Frequents.bar = Sells.bar);
  ```
Forcing Set/Bag Semantics

- Default for select-from-where is bag; default for union is set.
  - Why? Saves time of not comparing tuples as we generate them.
- Force set semantics with DISTINCT after SELECT.
  - But make sure the extra time is worth it.
- Force bag semantics with ALL after UNION.

Example

- Find the different prices charged for beers.
  SELECT DISTINCT price
  FROM Sells;
- Find all beers liked by Leo or Jim.

Aggregations

- Sum, avg, min, max, and count apply to attributes/columns.
- Count(*) applies to tuples.
- Use these in lists following SELECT.
- Find the average price of Bud.
  SELECT AVG(price)
  FROM Sells
  WHERE beer = 'Bud';
- Counts each tuple (for each bar that sells Bud) once.

Eliminating Duplicates Before Aggregation

- Find the number of different prices at which Bud is sold.
  SELECT COUNT(DISTINCT price)
  FROM Sells
  WHERE beer = 'Bud';
- DISTINCT may be used in any aggregation, but typically only makes 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.
- Find the average price for each beer.
  SELECT beer, AVG(price)
  FROM Sells
  GROUP BY beer;

Example

- 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;
- Note: grouping occurs after the \( \times \) and \( \sigma \) operations.
Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of a SELECT clause must either be aggregated or appear in a group-by clause.
- The following might seem a tempting way to find the bar that sells Bud the cheapest:
  ```sql
  SELECT bar, MIN(price)
  FROM Sells
  WHERE beer = 'Bud';
  ```
- But it is illegal in SQL.
- How would we find that bar?

HAVING Clauses

- HAVING clauses are selections on groups, just as WHERE clauses are selections on tuples.
- Condition can use the tuple variables or relations in the FROM and their attributes, just like the WHERE can.
  - But the t.v.’s range only over the group.
  - And the attribute better make sense within a group; i.e., be one of the grouping attributes.

Example

- Find the average price of those beers that are either served in at least 3 bars or manufactured by Anheuser-Busch.
  ```sql
  SELECT beer, AVG(price)
  FROM Sells
  GROUP BY beer
  HAVING COUNT(*) >= 3 OR
  beer IN (SELECT name
    FROM Beers
    WHERE manf = 'Anheuser-Busch');
  ```

Another Example

- Find, for each manufacturer, the beer with highest average price.