Announcements

- Assignment 2 is due now!
- Assignment 3 is due next Tuesday!
- Midterm on October 31.
  - in class; open book/notes.

Outline

- Continue with SQL.
- Subqueries
  - ANY, ALL, EXISTS, IN
- Aggregation
  - GROUP BY
  - HAVING

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
  ```sql
  SELECT bar
  FROM Sells
  WHERE beer = 'Miller' AND price =
  (SELECT price
   FROM Sells
   WHERE bar = 'Spoon' AND beer = 'Bud');
  ```
- Scoping rule: an attribute refers to the most closely nested relation with that attribute.
- Parentheses around subquery are essential.

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

- **EXISTS(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 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 Sells(bar, beer, price)
  ```sql
  SELECT beer
  FROM Sells
  WHERE price >= ALL
  (SELECT price
   FROM Sells);
  ```

Example

- Find the beer(s) not sold for the lowest price, given Sells(bar, beer, price).

Union, Intersection, Difference

- (subquery) **UNION** (subquery) produces the union of the two relations.
- Similarly for INTERSECT, EXCEPT = intersection and set difference.
- But: in Oracle set difference is MINUS, not EXCEPT.

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, intersection, and difference is set.
  - Why? Saves time of not comparing tuples as we generate them.
  - But we need to sort anyway when we take intersection or difference. (Union seems to be thrown in for good measure!)
- Force set semantics with DISTINCT after SELECT.
  - But make sure the extra time is worth it.
- Force bag semantics with ALL after UNION, etc.

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 × and σ 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.