More SQL

Relations as Bags
Grouping and Aggregation
Database Modification

Union, Intersection, and Difference

Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
- (subquery) UNION (subquery)
- (subquery) INTERSECT (subquery)
- (subquery) EXCEPT (subquery)

Example

From relations Likes(drinker, beer), Sells(bar, beer, price) and Frequent(drinker, bar), find the drinkers and beers such that:
- The drinker likes the beer, and
- The drinker frequents at least one bar that sells the beer.
Solution

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

The drinker frequents a bar that sells the beer.

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 in relational algebra, it is easier to avoid eliminating duplicates.

Just work tuple-at-a-time.

When doing 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`...

Example: DISTINCT

- From `Sells(bar, beer, price)`, find all the different prices charged for beers:
  
  ```sql
  SELECT DISTINCT price
  FROM Sells;
  ```

- Notice that without `DISTINCT`, each price would be listed as many times as there were bar/beer pairs at that price.

Example: ALL

- Using relations `Frequents(drinker, bar)` and `Likes(drinker, beer)`:
  
  ```sql
  (SELECT drinker FROM Frequents) EXCEPT ALL
  (SELECT drinker FROM Likes);
  ```

- Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.
Join Expressions

- SQL provides a number of expression forms that act like varieties of join in relational algebra.
  - But using bag semantics, not set semantics.
- These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Products and Natural Joins

- Natural join is obtained by:
  
  R NATURAL JOIN S;
- Product is obtained by:
  
  R CROSS JOIN S;
- Example:
  
  Likes NATURAL JOIN Serves;
- Relations can be parenthesized subexpressions, as well.

Theta Join

- R JOIN S ON <condition> is a theta-join, using <condition> for selection.
- Example: using Drinkers(name, addr) and Frequents(drinker, bar):
  
  Drinkers JOIN Frequents ON name = drinker;
  
  gives us all (d, a, d, b) quadruples such that drinker d lives at address a and frequents bar b.
Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
  - Optional NATURAL in front of OUTER.
  - Optional ON <condition> after JOIN.
  - Optional LEFT, RIGHT, or FULL before OUTER.
    - LEFT = pad dangling tuples of R only.
    - RIGHT = pad dangling tuples of S only.
    - FULL = pad both; this choice is the default.

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

- DISTINCT inside an aggregation causes duplicates to be eliminated before the aggregation.
- Example: find the number of different prices charged for Bud:
  ```sql
  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-NUL values in a column, then the result of the aggregation is NULL.

Example: Effect of NULL’s

- ```sql
  SELECT count(*)
  FROM Sells
  WHERE beer = 'Bud';
  ```
  The number of bars that sell Bud.

- ```sql
  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;
  ```

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 drinker-bar-price of Bud tuples first, then group by drinker.
Restriction on SELECT Lists With Aggregation

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

Illegal Query Example

- You might think you could find the bar that sells Bud the cheapest by:
  
  ```sql
  SELECT bar, MIN(price)
  FROM Sells
  WHERE beer = 'Bud';
  ```
  
  - But this query is illegal in SQL.
  - Why? Note bar is neither aggregated nor 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.
Requirements on HAVING Conditions

- These conditions may refer to any relation or tuple-variable in the FROM clause.
- They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
  - A grouping attribute, or
  - Aggregated.

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');
```
Database Modifications

- A modification command does not return a result as a query does, but it changes the database in some way.
- There are three kinds of modifications:
  - Insert a tuple or tuples.
  - Delete a tuple or tuples.
  - 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.
- There are two reasons to do so:
  - We forget the standard order of attributes for the relation.
  - 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):

```
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```

Inserting Many Tuples

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

```
INSERT INTO <relation>
( <subquery> )
```

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 Frequents d1, Frequents 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:
  ```sql
  DELETE FROM <relation>
  WHERE <condition>;
  ```

---

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:
  ```sql
  DELETE FROM Likes
  WHERE drinker = 'Sally' AND
  beer = 'Bud';
  ```
Example: Delete all Tuples

- Make the relation Likes empty:
  
  ```sql
  DELETE FROM Likes;
  ```

- Note no WHERE clause needed.

Example: Delete Many Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.
  
  ```sql
  DELETE FROM Beers b
  WHERE EXISTS (SELECT name FROM Beers WHERE manf = b.manf AND name <> b.name);
  ```

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

- The answer is that 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 in the original relation.
  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;