Union, Intersection, Difference

"Relation `UNION` relation" produces the union of the two relations.

- Similarly for `INTERSECT`, `EXCEPT` = intersection and set difference.
  - ✧ But: in Oracle 7.3.2 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.

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

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

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 (unlike a list).
- Example: $\{1, 2, 1, 3\}$ is a bag and not a set.

Bag Union

Sum the times an element appears in the two bags.

- Example: $\{1, 2, 1\} \cup \{1, 2, 3\} = \{1, 1, 1, 2, 2, 3\}$.

Bag Intersection

Take the minimum of the number of occurrences in each bag.

- Example: $\{1, 2, 1\} \cap \{1, 2, 3\} = \{1, 2\}$.

Bag Difference

Proper-subtract the number of occurrences in the two bags.

- Example: $\{1, 2, 1\} - \{1, 2, 3\} = \{1\}$.

Laws for Bags Differ From Laws for Sets

- Some familiar laws continue to hold for bags.
  - ✧ Examples: union and intersection are still commutative and associative.
- But other laws that hold for sets do *not* hold for bags.

Example

$R \cap (S \cup T) \equiv (R \cap S) \cup (R \cap T)$ holds for sets.

- Let R, S, and T each be the bag $\{1\}$.
- Left side: $S \cup T = \{1, 1\}$; $R \cap (S \cup T) = \{1\}$.
- Right side: $R \cap S = R \cap T = \{1\}$; $(R \cap S) \cup (R \cap T) = \{1, 1\} \neq \{1\}$.

Forcing Set/Bag Semantics

- Default for select-from-where is bag; default for union, interesection, 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.

Example

Find the different prices charged for beers.

```
Sells(bar, beer, price)

SELECT DISTINCT price
FROM Sells;
```

- Force bag semantics with `ALL` after `UNION`, etc.

Aggregations

Sum, avg, min, max, and count apply to attributes/columns. Also, count(*) 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 (presumably each bar that sells Bud) once.

Problem

What would we do if `Sells` were a bag?

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 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.

Example

Find the average sales price for each beer.

```
Sells(bar, beer, price)

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

Example

Find, for each drinker, the average price of Bud at the bars they frequent.

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

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

- Note: grouping occurs after the product and selection.

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.

Example

The following might seem a tempting way to find the bar that sells Bud the cheapest:

```
Sells(bar, beer, price)

SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Bud';
```

- **But it is illegal in SQL2.**

Problem

How would we find that bar?

`HAVING` **clauses**

- "`HAVING` condition" eliminates groups when condition is false.
- 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.

Example

Find the average price of those beers that are either served in at least 3 bars or manufactured by Anheuser-Busch.

```
Beers(name, manf)
Sells(bar, beer, price)

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

DB Modifications

Modification = insert + delete + update.

Insertion of a Tuple

`INSERT INTO` relation `VALUES` (list of values).

- Inserts the tuple = list of values, associating values with attributes in the order the attributes were declared.
    - ❖ Forget the order? List the attributes as arguments of the relation.

Example

```
        Likes(drinker, beer)
```

Insert the fact that Sally likes Bud.

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

Insertion of the Result of a Query

`INSERT INTO` relation (subquery).

Example

Create a (unary) table of all Sally's potential buddies, i.e., the people who frequent bars that Sally also frequents.

```
Frequents(drinker, bar)

CREATE TABLE PotBuddies(
    name char(30)
);

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

Deletion

`DELETE FROM` relation `WHERE` condition.

- Deletes all tuples satisfying the condition from the named relation.

Example

Sally no longer likes Bud.

```
Likes(drinker, beer)

DELETE FROM Likes
WHERE drinker = 'Sally' AND
    beer = 'Bud';
```

Example

Make the `Likes` relation empty.

```
DELETE FROM Likes;
```

Example

Delete all beers for which there is another beer by the same manufacturer.

```
Beers(name, manf)

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

- Note alias for relation from which deletion occurs.

- Semantics is tricky. If A.B. makes Bud and BudLite, does deletion of Bud make BudLite *not* satisfy the condition?
    - SQL2 formal semantics says "no." Oracle implements this deletion properly (i.e., both get deleted).
- General principle: all conditions in modifications *should* be evaluated by the system before any mods due to that mod command occur.
    - Especially important when subqueries are used, because they can refer to the relation being modified.

Updates

`UPDATE` relation `SET` list of assignments `WHERE` condition.

Example

Drinker Fred's phone number is 555-1212.

```
Drinkers(name, addr, phone)

UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```

Example

Make $4 the maximum price for beer.

- Updates many tuples at once.

```
Sells(bar, beer, price)

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