Database Modifications and Transactions

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 Likes2(drinker, beer) the fact that Sally likes Bud.

```
INSERT INTO Likes2
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):

```
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:

name	address	phone
Sally	123 Sesame St	NULL

Inserting Many Tuples

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

  INSERT INTO <relation>

  (<subquery>);

Subquery can be any SELECT statement we have seen with grouping, set operations, other subqueries, etc.

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.

The other drinker

Solution

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

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

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>
WHERE <condition>;
```

Condition can be any WHERE condition we have seen, including IN, ANY, ALL, EXISTS + subquery.

Example: Deletion

- Delete from Likes(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;
```

Transactions

Controlling Concurrent Behavior

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- Compare: An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

Transactions

- *Transaction* = process involving database queries and/or modification.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.
- Optional: weaker forms of transactions are often supported as well.

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - It’s database modifications are now permanent in the database.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges. (max,min)
- Joe decides to stop selling (delete) Bud and Miller, but to sell only Heineken (insert) at $3.50.

Sally’s Program

- Sally executes the following two SQL statements called (max) and (min) to help us remember what they do.

```
(max)     SELECT MAX(price) FROM Sells
          WHERE bar = ’Joe’’s Bar’;
(min)     SELECT MIN(price) FROM Sells
          WHERE bar = ’Joe’’s Bar’;
```

Joe’s Program

- At about the same time, Joe executes the following steps: (del) and (ins).

```
(del)   DELETE FROM Sells
        WHERE bar = ’Joe’’s Bar’;
(ins)   INSERT INTO Sells
        VALUES(’Joe’’s Bar’, ’Heineken’, 3.50);
```

Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe’s Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level ("serializable") = ACID transactions.
- Each DBMS implements transactions in its own way.

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

/* Oracle allows only 1 and 3 and some similar to 2. */

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- Example: If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.