Transactions, Views, Indexes

- Introduction to Transactions: Controlling Concurrent Behavior
- Virtual and Materialized Views
- Indexes: Speeding Accesses to Data

Introduction to Transactions

- *Transaction* = process involving database queries and/or modification.
  - A transaction is a collection of 1+ operations that must be executed atomically.
    - So either all are executed or none are
- Normally with some strong requirements regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.
- Depending on the implementation, a transaction may start:
  - Implicitly, with the execution of a SELECT, UPDATE, … statement, or
  - Explicitly, with a BEGIN TRANSACTION statement
- Transaction finishes with a COMMIT or ROLLBACK statement

Why Transactions?

- Database systems are normally accessed by many users or processes at the same time.
  - This is the case for both queries and modifications.
- A DMBS needs to keep processes from troublesome interactions.
- As well, we have been assuming that operations are carried out atomically, and that the hardware or software can't fail in the middle of a modification.
  - Clearly this is an unrealistic assumption.

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time from the same account.
  - 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 person's changes get lost.

ACID Properties

- Properly executed transactions are said to meet the *ACID* criteria.
- *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.
- Weaker forms of transactions are often supported as well.

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - Its database modifications are now permanent in the database.
- Prior to a COMMIT, other transactions (commonly) cannot see the partial or tentative changes.
  - I.e. the database system might *lock* the changed items until the COMMIT is executed.

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.
- One use of ROLLBACK: Abort a database modification via a trigger.

Example: Interacting Processes

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

Sally's Program

- Sally executes the following two SQL statements called (min) and (max) 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}	{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

- Locking or isolating part of the database can be expensive.
- The *isolation level* controls the extent to which a transaction is exposed to the effects of other concurrent transactions.
- 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

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.
- If a transaction T is SERIALIZABLE, then it reads only changes made by committed transactions.
- As well, no value read or written by T is changed by another transaction until T completes.

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.
- With READ COMMITTED a transaction T reads only changes made by committed transactions, and no value written by T can be changed by other transactions until T is complete.
  - A value read by T may be changed by another transaction while T is in progress.

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.
- Common terminology:
  - *Dirty data* is data written by a transaction that has not yet committed.
  - A *dirty read* is a read of dirty data written by another transaction.

Views

- A *view* is a relation *defined* in terms of stored tables (called *base tables*) and other views.
- Two kinds:
  1. *Virtual* = not stored in the database
     - Just a query for constructing the relation.
  2. *Materialized* = actually constructed and stored.

Declaring Views

- Declare by:

  CREATE [MATERIALIZED] VIEW <name> AS <query>;

- Default is virtual.

Example: View Definition

- CanDrink(customer, beer) is a view "containing" the customer-beer pairs such that the customer frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
        SELECT customer, beer
        FROM Frequents, Sells
        WHERE Frequents.bar = Sells.bar;
```

- Renaming attributes:

```
CREATE VIEW CanDrink(person, beverage) AS
        SELECT customer, beer
        FROM Frequents, Sells
        WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

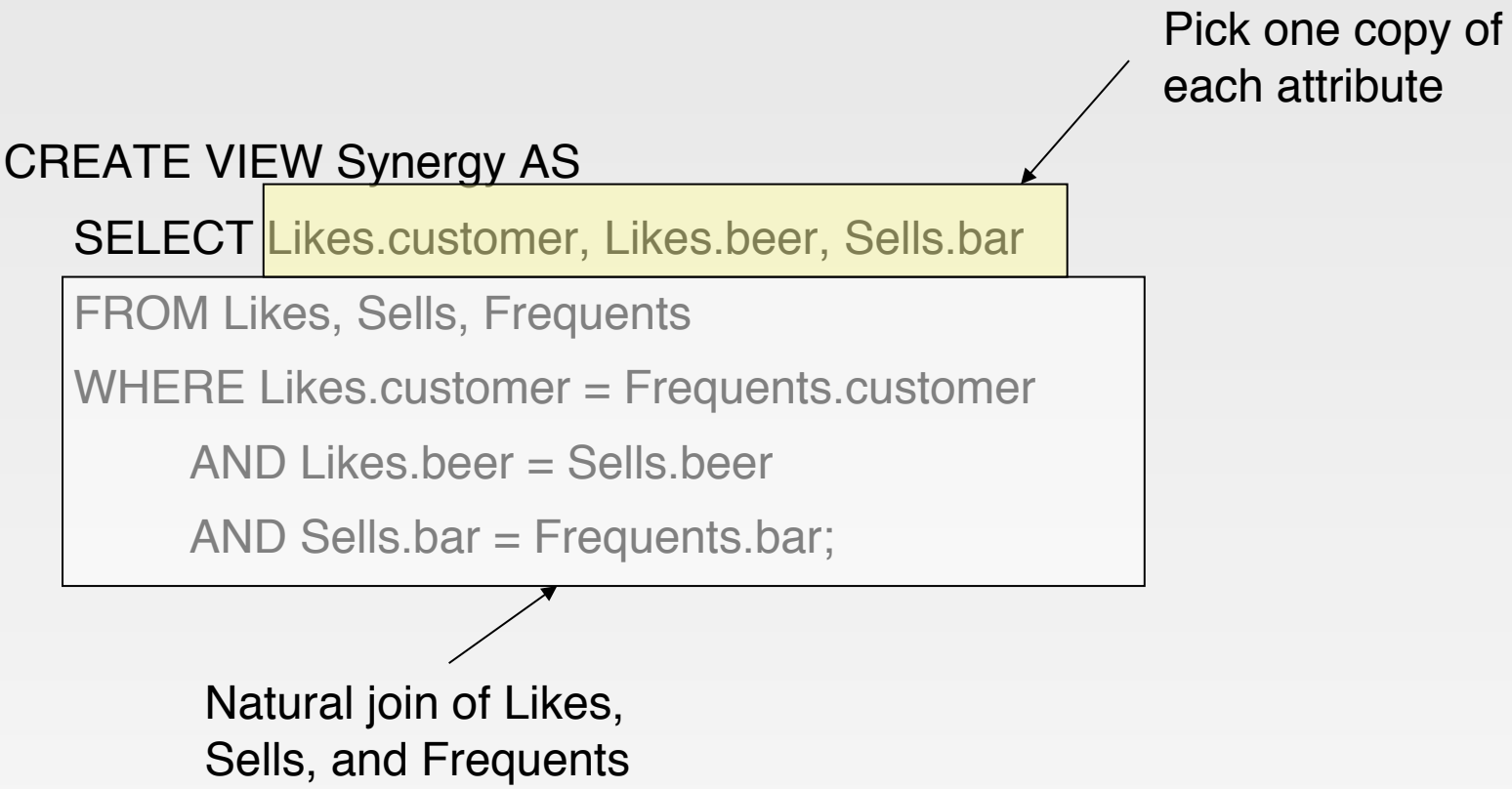
- Can query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

```
SELECT beer FROM CanDrink
WHERE customer = 'Sally';
```

Triggers on Views

- In general, it is impossible to modify a virtual view, because a virtual view doesn't exist.
- But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- Example: View Synergy has (customer, beer, bar) triples such that the bar serves the beer, the customer frequents the bar and likes the beer.

Example: The View



Interpreting a View Insertion

- We cannot insert into Synergy --- it is a virtual view.
- But we can use an INSTEAD OF trigger to turn a (customer, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.
  - Sells.price will have to be NULL.

The Trigger

```
CREATE TRIGGER ViewTrig
    INSTEAD OF INSERT ON Synergy
    REFERENCING NEW ROW AS n
    FOR EACH ROW
    BEGIN
        INSERT INTO LIKES VALUES(n.customer, n.beer);
        INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
        INSERT INTO FREQUENTS VALUES(n.customer, n.bar);
    END;
```

Some Views are not Updateable

- Example:

  CREATE VIEW BeerPrices AS

  SELECT beer, price

  FROM Sells

- Recall: SELLS has attributes bar, beer, prices
  - Primary key of SELLS is bar, beer

Materialized Views

- Problem: each time a base table changes, the materialized view may change.
  - Cannot afford to recompute the view with each change.
- Solution (sometimes): Periodic reconstruction of the materialized view, which is otherwise “out of date.”

Example: Data for Sales Analysis

- Sales of items require frequent modification of a database.
- If aggregated data (for example) is to be used to analyse sales, then materialised views may be used.
- May be updated once a day
  - Justification: Things don't change much over 24 hours.

Example: A Data Warehouse

- Canadian Tire stores every sale at every store in a database.
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales.
- The warehouse is used by analysts to predict trends and move goods around.

- We'll see more on data warehouses later…

Indexes

- An *index* for an attribute (or attributes) of a relation is a data structure used to speed access to tuples of a relation, given values of the attribute(s).
- In a DBMS it is a balanced search tree with giant nodes (a full disk page) called a *B-tree*.
- Can make query answering and joins involving the attribute much faster.
- On the other hand, modifications are more complex and take longer.
- Covered in depth in CMPT454

Declaring Indexes

- No standard!
- Typical syntax:

```
CREATE INDEX BeerInd   ON Beers(manf);
CREATE INDEX SellInd   ON Sells(bar, beer);
```

Using Indexes

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.
- Example: use BeerInd and SellInd to find the prices of beers manufactured by Pete's and sold by Joe. (next slide)
- With the indices, just retrieve tuples satisfying these conditions
  - Clearly, can result in huge savings (vs. retrieving all tuples from the mentioned relations)

Using Indexes --- (2)

```
SELECT price
FROM Beers, Sells
WHERE manf = 'Pete''s' AND
        Beers.name = Sells.beer AND
        bar = 'Joe''s Bar';
```

1. Use BeerInd to get all the beers made by Pete's.
2. Then use SellInd to get prices of those beers, with bar = 'Joe''s Bar'

Database Tuning

- A major problem in making a database run fast is deciding which indexes to create.
- Recall:
  - Pro: An index speeds up queries that can use it.
  - Con: An index slows down modifications on its relation because the index must be modified too.
- The key for a relation is usually the most useful attribute to have an index on:
  - Queries in which a value for a key is specified are common.
  - For a given key value there is only one tuple. Thus the index returns at most one tuple, requiring just 1 page from the relation instance to be retrieved.

Example: Tuning

- Suppose the only things we did with our beers database was:
  1. Insert new facts into a relation (10%).
  2. Find the price of a given beer at a given bar (90%).
- Then
  - SellInd on Sells(bar, beer) would be wonderful, but
  - BeerInd on Beers(manf) would be harmful.

Tuning Advisors

- Use a *tuning advisor* to help determine appropriate indexes.
  - A major research thrust.
  - Hand tuning is difficult and inaccurate.

Tuning Advisors --- (1)

- An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload or constraints.

Tuning Advisors --- (2)

- The advisor generates candidate indexes and evaluates each on the workload.
- Feed each sample query to the query optimizer, which assumes only this one index is available.
- Measure the improvement/degradation in the average running time of the queries.
- Problem: Indexes are not independent – the choice of one may affect the performance of another.
  - In practice, a *greedy* algorithm works well.

End of Transactions, Views, Indexes