# Transactions, Views, Indexes

CONTROLLING CONCURRENT BEHAVIOR

VIRTUAL AND MATERIALIZED VIEWS

SPEEDING ACCESSES TO DATA

# Why Transactions?

Database systems are normally being accessed by many users or processes at the same time.

- Both queries and modifications.

The same as operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

# Example: Bad Interaction

You and your 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.

# 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 (Query 1) and lowest (Query2) price Joe charges.

Joe decides to stop selling Bud and Miller (Query 3), but to sell only Heineken at $3.50 (Query 4).

# 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, <u>unless we group Sally's and/or Joe's statements into transactions</u>!

# Example: Strange Interleaving

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

What is the result of Query 1 and Query 2?

# 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

# 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 3.00.
- (min) can see 3.50, but must also see 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.

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

Frequents(drinker, bar)

Sells(bar, beer, price)

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

# Example: Accessing a View

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 drinker = 'Sally';
```

# Triggers on Views

Generally, it is impossible to modify a virtual view, because it doesn't exist.

But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.

Example: View Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

# Example: The View

Pick one copy of each attribute

CREATE VIEW Synergy AS

```
SELECT Likes.drinker, Likes.beer, Sells.bar
FROM Likes, Sells, Frequents
WHERE Likes.drinker = Frequents.drinker
      AND Likes.beer = Sells.beer
      AND Sells.bar = Frequents.bar;
```

Natural join of Likes, Sells, and Frequents

Frequents(<u>drinker</u>, <u>bar</u>)
Sells(<u>bar</u>, <u>beer</u>, price)
Likes(<u>drinker</u>, <u>beer</u>)

# Interpreting a View Insertion

We <u>cannot insert into Synergy</u> --- it is a virtual view.

But we can use an **INSTEAD OF trigger** to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.

- Sells.price will have to be NULL (as price is not an element of the view).

# The Trigger

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

# Materialized Views

Problem: each time a base table changes, the materialized view may change.

- Cannot afford to recompute the view with each change.

Solution: **Periodic reconstruction** of the **materialized view**, which is otherwise “out of date.”

# Example: Class Mailing List

Eg., the class mailing list cs-students can be a **materialized view** of the class enrollment.

Actually updated four times/day.

- You can enroll and miss an email sent out after you enroll.

# Example: A Data Warehouse

Wal-Mart 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 to where they are selling best.

# Indexes

***Index*** = **data structure** used to **speed access** to tuples of a relation, given values of one or more attributes.

Could be a **hash table**, but in a DBMS it is usually a **balanced search tree** with giant nodes (a full disk page) called a *B-tree*.

# 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)

# 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’

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

# Database Tuning

A major problem in **making a database run fast is deciding which indexes to create**.

Pro: An index **speeds up queries** that can use it.

Con: An index **slows down all modifications** on its relation because the index must be modified too.

# 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 at a given bar of a given beer (90%).

Then SellInd on Sells(bar, beer) would be wonderful, but BeerInd on Beers(manf) would be harmful.

# Tuning Advisors

A major research thrust.

- Because hand tuning is so hard.

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.

# 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.

# Actions

Review slides!

Read Chapter 6.6 Transactions in SQL and Chapter 8 Views and Indexes.