Database Management Systems - I, CS 157A

Transactions in SQL & Constraints and Triggers & Views and Indexes

Ch. 6.6, Ch. 7, Ch. 8

Agenda

- Transactions Overview
- Constraints Overview
- Triggers Overview
- Views Overview
- Index Overview

Transactions Overview

Ch. 6.6

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*** = set of operations as a unit of work involving database queries and/or modification
- Normally with some strong properties regarding concurrency
- Formed in SQL from single statements or explicit programmer control (transaction demarcation)

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
- Joe decides to stop selling Bud and Miller, but 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}
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 will not 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 because of the rollback

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 will *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, i.e. after delete but before insert

Read-Committed 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:
  - 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 (Max, Min) 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

Constraints

Ch. 7

Constraints and Triggers

- A ***constraint*** is a relationship among data elements that the DBMS is required to enforce:
  - **Example:** key constraints
- ***Triggers*** are only executed when a specified condition occurs, e.g., insertion of a tuple:
  - Easier to implement than complex constraints

Kinds of Constraints

- **Keys**
- **Foreign-key**, or **referential-integrity**
- **Value-based** constraints
  - Constrain values of a particular attribute
- **Tuple-based** constraints
  - Relationship among components
- **Assertions:** any SQL boolean expression

Review: Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute

- **Example:**

```
CREATE TABLE Beers (
    name CHAR(20) UNIQUE,
    manf CHAR(20)
);
```

Review: Multiattribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
    bar         CHAR(20),
    beer        VARCHAR(20),
    price       REAL,
    PRIMARY KEY (bar, beer)
);
```

Foreign Keys

- Values appearing in attributes of one relation must appear together in certain attributes of another relation

- **Example:** in Sells(bar, beer, price), we might expect that a Sells.beer value also appears in Beers.name

Expressing Foreign Keys

- Use keyword REFERENCES, either:
  1. After an attribute (for one-attribute keys).
  2. As an element of the schema:

     FOREIGN KEY (<list of attributes>)

     REFERENCES <relation> (<attributes>)

- Referenced attributes must be declared PRIMARY KEY or UNIQUE

Example: With Attribute

```
CREATE TABLE Beers (
  name    CHAR(20) PRIMARY KEY,
  manf    CHAR(20) );

CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20) REFERENCES Beers(name),
  price   REAL );
```

Example: As Schema Element

```
CREATE TABLE Beers (
   name       CHAR(20) PRIMARY KEY,
   manf       CHAR(20) );


CREATE TABLE Sells (
   bar        CHAR(20),
   beer       CHAR(20),
   price      REAL,
   FOREIGN KEY(beer) REFERENCES Beers(name));
```

Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from relation *R* (*Foreign/references*), to relation *S* (primary/referenced) two violations are possible:

1. An insert or update to *R* introduces values not found in *S*
2. A deletion or update to S causes some tuples of *R* to “dangle”

Actions Taken --- (1)

- **Example:** suppose *R* = **Sells** <bar, beer, price>, *S* = Beers.
- An insert or update to Sells that introduces a nonexistent beer must be rejected!
- A deletion or update to **Beers** <<u>name</u>, manf> that removes a beer value found in some tuples of Sells <bar, beer, price> can be handled in three ways (next slide):

Actions Taken --- (2)

1. ***Default*****:** Reject the modification
2. ***Cascade*****:** Make the same changes in Sells:
   - Deleted beer tuple: delete Sells tuple
   - Updated beer tuple: change value in Sells
3. ***Set NULL*****:** Change the beer attribute in R/Sells tuple to NULL

Example: Cascade

- Delete the Bud tuple from Beers:
  - Then delete all tuples from **Sells** that have beer = ’Bud’

- Update the Bud tuple in Beers by changing ’Bud’ to ’Budweiser’:
  - Then change all **Sells** tuples with beer = ’Bud’ to beer = ’Budweiser’

Example: Set NULL

- Delete the Bud tuple from Beers:
  - Change all tuples of Sells that have beer = ’Bud’ to have beer = NULL

- Update the Bud tuple in Beers by changing ’Bud’ to ’Budweiser’:
  - Same change in Sells as we did for deletion

Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates

- Follow the foreign-key declaration by:

  ON [UPDATE, DELETE][SET NULL or CASCADE]

- Two such clauses may be used

- Otherwise, the default (reject) is used

Example: Setting Policy

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20),
  price  REAL,
  FOREIGN KEY(beer)
    REFERENCES Beers(name)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```

Foreign key

Primary key

Attribute-Based Checks

- Constraints on the value of a particular attribute
- Add CHECK(<condition>) to the declaration for the attribute
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery

Example: Attribute-Based Check

```
CREATE TABLE Sells (
  bar      CHAR(20),
  beer     CHAR(20)  CHECK ( beer IN
                     (SELECT name FROM Beers)),
  price    REAL      CHECK ( price <= 5.00 )
);
```

Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated
  - **Example:** `CHECK(price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5
  - **Example:** `CHECK(beer IN(SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys)

Tuple-Based Checks

- **CHECK** (<condition>) may be added as a relation-schema element
- The condition may refer to any attribute of the relation:
  - But other attributes or relations require a subquery
- Checked on **insert** or **update** only

Example: Tuple-Based Check

- Only Joe's Bar can sell beer and for more than \$5 (i.e., check that either [bar is "Joe's Bar"] or [price <= \$5] is TRUE):

```
CREATE TABLE Sells (
    bar       CHAR(20),
    beer      CHAR(20),
    price     REAL,
    CHECK     (bar = 'Joe''s Bar' OR
                     price <= 5.00)
);
```

Assertions

- These are database-schema elements, like relations or views
- **Defined by:**

  CREATE ASSERTION <name>

  CHECK (<condition>);

- Condition may refer to any relation or attribute in the database schema

Example: Assertion

- In **Sells**(bar, beer, price), no bar may charge an average price of more than $5

```
CREATE ASSERTION NoRipoffBars CHECK (
  NOT EXISTS (
      SELECT bar FROM Sells
      GROUP BY bar
      HAVING AVG(price) > 5.00
));
```

Bars with an average price above $5

- Subquery returns bar(s) with average price > $5. Check() evaluates to TRUE if there is no bar(s) returned from the subquery!

Example: Assertion

- In Drinkers(name, addr, phone) and Bars(name, addr, license), there cannot be more bars than drinkers

```
CREATE ASSERTION FewBar CHECK (
  (SELECT COUNT(*) FROM Bars) <=
  (SELECT COUNT(*) FROM Drinkers)
);
```

- Check evaluates to TRUE if # of Bars <= # of Drinkers

Timing of Assertion Checks

- In principle, we must check every assertion after every modification to any relation of the database

- A clever system can observe that only certain changes could cause a given assertion to be violated:
  - **Example:** No change to Beers can affect FewBar! Neither can an insertion to Drinkers can change the Assertion to be False

Triggers
Ch. 7

Triggers: Motivation

- Assertions are powerful, but the DBMS often can’t tell when they need to be checked
- Attribute- and tuple-based checks are checked at known times, but are not powerful
- Triggers let the user decide when to check for any condition

Event-Condition-Action Rules

- Another name for "trigger" is ***ECA*** *rule*, or ***Event-Condition-Action*** rule
- ***Event*****:** typically a type of database modification, e.g., "insert on Sells"
- ***Condition*****:** Any SQL boolean-valued expression
- ***Action*****:** Any SQL statements

Preliminary Example: A Trigger

- Instead of using a foreign-key constraint and rejecting insertions into Sells(bar, beer, price) ← (*Foreign/references)* with unknown beers (primary/referenced), a trigger can add that beer to Beers, with a NULL manufacturer ☺

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig
  AFTER INSERT ON Sells
  REFERENCING NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.beer NOT IN
        (SELECT name FROM Beers))
  INSERT INTO Beers(name)
              VALUES(NewTuple.beer);
```

The **E**vent

The **C**ondition

The **A**ction

< If inserting new Beer in the “Sells” table that is not known in the “Beers” table, then insert tuple with the new Beer in the “Beers” Table>

Options: CREATE TRIGGER (1)

CREATE TRIGGER <name>

- Or:

CREATE OR REPLACE TRIGGER <name>

  - Useful if there is a trigger with that name and you want to modify the existing trigger

Options: The Event (2)

- **AFTER** can be **BEFORE**
  - Also, **INSTEAD OF**, if the relation is a view
    - A clever way to execute view modifications: have triggers translate them to appropriate modifications on the base tables
- **INSERT** can be **DELETE** or **UPDATE**
  - And **UPDATE** can be **UPDATE . . . ON** a particular attribute

Options: REFERENCING (3)

- **INSERT** statements imply a new tuple (for row-level) or new table (for statement-level):
  - The “table” is the set of inserted tuples
- **DELETE** implies an old tuple or table
- **UPDATE** implies both, i.e., Delete and Insert
- Refer to these by

  [NEW OLD][TUPLE TABLE] **AS** <name>

Options: FOR EACH ROW (4)

- Triggers (E-C-A) are either “row-level” or “statement-level”:
  - **FOR EACH ROW** indicates row-level; its absence indicates statement-level
  - ***Row level triggers*:** execute once for each modified tuple
  - ***Statement-level triggers*:** execute once for a SQL statement, regardless of how many tuples are modified

Options: The Condition (5)

- Any boolean-valued condition
- Evaluated on the database as it would exist before or after the triggering event, depending on whether **BEFORE** or **AFTER** is used
  - But always before the changes take effect
- Access the new/old tuple/table through the names in the **REFERENCING** clause

Options: The Action (6)

- There can be more than one SQL statement in the action:
  - Surround by **BEGIN . . . END** if there is more than one
- But queries make no sense in an action, so we are really limited to modifications

Another Example

- Using Sells(bar, beer, price) and a unary relation RipoffBars(bar), maintain a list of bars that raise the price of any beer by more than $1

The Trigger

```
CREATE TRIGGER PriceTrig
  AFTER UPDATE OF price ON Sells
  REFERENCING
      OLD ROW AS ooo
      NEW ROW AS nnn
  FOR EACH ROW
  WHEN(nnn.price > ooo.price + 1.00)
  INSERT INTO RipoffBars
          VALUES(nnn.bar);
```

The event – only changes to prices

Updates let us talk about old and new tuples

We need to consider each price change

Condition: a raise in price > $1

When the price change is great enough, add the bar to RipoffBars

Views
Ch. 8

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:

```
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 drinker 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

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 (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**.

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 re-compute the view with each change
- **Solution:** Periodic reconstruction of the materialized view, which is otherwise "out of date"

Example: Axess/Class Mailing List

- The class mailing list cs510-aut0708-students is in effect a materialized view of the class enrollment in Axess

- **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
Ch. 8

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 always 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** (index on attribute (manf) in **Beers** table) and **SellInd** {index on attribute (bar, beer) in **Sells** table} to find the prices of beers manufactured by Pete's and sold by Joe's bar. ➔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 (i.e., <beer, price>), with bar = 'Joe''s Bar'
3. Join output of steps (1) and (2) to get the price of beers that are made by Pete's and are sold in Joe's Bar

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

- Then **SellInd** on **Sells**(bar, beer) → {READ} would be wonderful, but **BeerInd** on **Beers**(manf) → {INSERT} would be harmful

Tuning (Design) 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 "**design 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

END