SQL: Part III

CPS 196.3
Introduction to Database Systems

Announcement

- Homework #1 will be graded by next week
  - Sample solution will be handed out this Wednesday
  - Grades will be posted on Blackboard
- Homework #2 will be assigned this Wednesday (September 17)
- Presentation assignments will be finalized this week

“Active” data

- Constraint enforcement: When a transaction violates a constraint, abort the transaction or try to “fix” the data
  - Example: enforcing referential integrity constraints
  - Generalize to arbitrary constraints?
- Data monitoring: When something happens to the data, automatically execute some action
  - Example: When price rises above $20 per share, sell
  - Example: When enrollment is at the limit and more students try to register, email the instructor

Triggers

- A trigger is an event-condition-action rule
  - When event occurs, test condition; if condition is satisfied, execute action
- Example:
  - Event: whenever there comes a new student…
  - Condition: with GPA higher than 3.0…
  - Action: then make him/her take CPS196!

Trigger example

```
CREATE TRIGGER CPS196AutoRecruit
AFTER INSERT ON Student
REFERENCING NEW ROW AS newStudent
FOR EACH ROW
WHEN (newStudent.GPA > 3.0)
INSERT INTO Enroll
VALUES(newStudent.SID, 'CPS196');
```

Trigger options

- Possible events include:
  - INSERT ON table
  - DELETE ON table
  - UPDATE (OF column) ON table
- Trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification
- Action can be executed:
  - AFTER or BEFORE the triggering event
Statement-level trigger example

CREATE TRIGGER CPS196AutoRecruit
AFTER INSERT ON Student
REFERENCING NEW TABLE AS newStudents
FOR EACH STATEMENT
INSERT INTO Enroll
(SELECT SID, 'CPS196'
FROM newStudents
WHERE GPA > 3.0);

BEFORE trigger example

Never give faculty more than 50% raise in one update
CREATE TRIGGER NotTooGreedy
BEFORE UPDATE OF salary ON Faculty
REFERENCING OLD ROW AS o, NEW ROW AS n
FOR EACH ROW
WHEN (n.salary > 1.5 * o.salary)
SET n.salary = 1.5 * o.salary;

Statement- vs. row-level triggers

Why are both needed?
• Certain triggers are only possible at statement level
  • If the average GPA of students inserted by this
    statement exceeds 3.0, do …
• Simple row-level triggers are easier to implement
  and may be more efficient
  • Statement-level triggers require significant amount of
    state to be maintained in OLD TABLE and NEW TABLE
• However, a row-level trigger does get fired for each row,
  so complex row-level triggers may be inefficient for
  statements that generate lots of modifications

Another statement-level trigger

Give faculty a raise if GPA’s in one update statement are all increasing
CREATE TRIGGER AutoRaise
AFTER UPDATE OF GPA ON Student
REFERENCING OLD TABLE AS o, NEW TABLE AS n
FOR EACH STATEMENT
WHEN (NOT EXISTS(SELECT * FROM o, n
WHERE o.SID = n.SID
AND o.GPA >= n.GPA))
UPDATE Faculty SET salary = salary + 1000;
• A row-level trigger would be difficult to write in this case

System issues

• Recursive firing of triggers
  • Action of one trigger causes another trigger to fire
  • Can get into an infinite loop
    • Some DBMS restrict trigger actions
    • Most DBMS set a maximum level of recursion (16 in DB2)
• Interaction with constraints (very tricky to get right!)
  • When do we check if a triggering event violates constraints?
    • After a BEFORE trigger (so the trigger can fix a potential violation)
    • Before an AFTER trigger
  • AFTER triggers also see the effects of, say, cascaded deletes caused by
    referential integrity constraint violations
    (Based on DB2; other DBMS may implement a different policy!)
Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the view definition query instead of view contents
  - Can be used in queries just like a regular table

Creating and dropping views

- Example: CPS196 roster
  - CREATE VIEW CPS196Roster AS
    SELECT SID, name, age, GPA
    FROM Student
    WHERE SID IN (SELECT SID FROM Enroll
    WHERE CID = 'CPS196');
  - To drop a view
    - DROP VIEW view_name;

Using views in queries

- Example: find the average GPA of CPS196 students
  - SELECT AVG(GPA) FROM CPS196Roster;
  - To process the query, replace the reference to the view by its definition
    - SELECT AVG(GPA)
      FROM (SELECT SID, name, age, GPA
      FROM Student
      WHERE SID IN (SELECT SID
      FROM Enroll
      WHERE CID = 'CPS196'));

Why use views?

- To hide data from users
- To hide complexity from users
- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications
  - Recall physical data independence: change the physical organization of data without affecting applications
- Real database applications use tons of views

Modifying views

- Does not seem to make sense since views are virtual
- But does make sense if that is how users see the database
- Goal: modify the base tables such that the modification would appear to have been accomplished on the view

A simple case

CREATE VIEW StudentGPA AS
SELECT SID, GPA FROM Student;
DELETE FROM StudentGPA WHERE SID = 123;
translates to:
DELETE FROM Student WHERE SID = 123;
An impossible case

CREATE VIEW HighGPAStudent AS
  SELECT SID, GPA FROM Student
  WHERE GPA > 3.7;

INSERT INTO HighGPAStudent
  VALUES(987, 2.5);

- No matter what you do on Student, the inserted row will not be in HighGPAStudent

A case with too many possibilities

CREATE VIEW AverageGPA(GPA) AS
  SELECT AVG(GPA) FROM Student;

- Note that you can rename columns in view definition

UPDATE AverageGPA SET GPA = 2.5;

- Set everybody's GPA to 2.5?
- Adjust everybody's GPA by the same amount?
- Just lower Bart's GPA?

SQL92 updateable views

- Single-table SFW
  - No aggregation
  - No subqueries

- Overly restrictive
  - Still might get it wrong in some cases
    - See the slide titled "An impossible case"

Indexes

- An index is an auxiliary persistent data structure
  - Search tree (e.g., B+-tree), lookup table (e.g., hash table), etc.
  - More on indexes in the second half of this course!
- An index on \( R.A \) can speed up accesses of the form
  - \( R.A = value \)
  - \( R.A > value \) (sometimes; depending on the index type)
  - An index on \( \{ R.A_1, \ldots, R.A_n \} \) can speed up
    - \( R.A_1 = value_1 \land \ldots \land R.A_n = value_n \)
    - Is an index on \( \{ R.A, R.B \} \) equivalent to an index on \( R.A \) plus another index on \( R.B \)?

Examples of using indexes

- SELECT * FROM Student WHERE name = 'Bart'
  - Without an index on Student.name: must scan the entire table if we store Student as a flat file of unordered rows
  - With index: go "directly" to rows with name = 'Bart'

- SELECT * FROM Student, Enroll
  WHERE Student.SID = Enroll.SID;
  - Without any index: for each Student row, scan the entire Enroll table for matching SID
  - Sorting could help
  - With an index on Enroll.SID: for each Student row, directly look up Enroll rows with matching SID

Creating and dropping indexes in SQL

- CREATE (UNIQUE) INDEX index_name ON table_name(column_name_1, ..., column_name_n);
  - With UNIQUE, the DBMS will also enforce that \( \{ column_name_1, ..., column_name_n \} \) is a key of table_name

- DROP INDEX index_name;

- Typically, the DBMS will automatically create indexes for PRIMARY KEY and UNIQUE constraint declarations
Choosing indexes to create

- More indexes = better performance?
- Indexes take space
- Indexes have one more level of indirection
- Indexes need to be maintained when data is updated

- Optimal index selection depends on both query and update workload and the size of tables
  - Automatic index selection is still an area of active research

Summary of SQL features covered so far

- Query
- Modification
- Constraints
- Triggers
- Views
- Indexes

Next: transactions