SQL: Triggers, Views, Indexes
Introduction to Databases
CompSci 316 Fall 2015

Announcements (Tue., Sep. 22)

• Homework #1 sample solution posted on Sakai
• Homework #2 due in two weeks (Tuesday)
  • Midterm on Thursday of the same week
• Project Milestone #1 due Thursday, Oct. 15
  • See project description on what to accomplish by then

Announcements (Thu., Sep. 24)

• Homework #2 due in 1½ weeks
• Midterm in class in two weeks
  • Open-book, open-notes
  • Same format as sample midterm (from last year), to be posted on Sakai by Tuesday

“Active” data

• Constraint enforcement: When an operation violates a constraint, abort the operation or try to “fix” 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 (ECA) rule
  • When event occurs, test condition; if condition is satisfied, execute action

• Example:
  • Event: some user’s popularity is updated
  • Condition: the user is a member of “Jessica’s Circle,” and pop drops below 0.5
  • Action: kick that user out of Jessica’s Circle

Trigger example

```
CREATE TRIGGER PickyJessica
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
WHEN (newUser.pop < 0.5)
AND (newUser.uid IN (SELECT uid FROM Member WHERE gid = 'jes'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'jes';
```
Trigger options

- Possible events include:
  - INSERT ON table
  - DELETE ON table
  - UPDATE [OF column] ON table

- Granularity—trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification

- Timing—action can be executed:
  - AFTER or BEFORE the triggering event
  - INSTEAD OF the triggering event on views (more later)

Transition variables

- OLD ROW: the modified row before the triggering event
- NEW ROW: the modified row after the triggering event
- OLD TABLE: a hypothetical read-only table containing all rows to be modified before the triggering event
- NEW TABLE: a hypothetical table containing all modified rows after the triggering event

Statement-level trigger example

CREATE TRIGGER PickyJessica
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
DELETE FROM Member
WHERE gid = 'jes'
AND uid IN (SELECT uid
FROM newUsers
WHERE pop < 0.5);

BEFORE trigger example

CREATE TRIGGER NoFountainOfYouth
BEFORE UPDATE OF age ON User
REFERENCING OLD ROW AS o,
NEW ROW AS n
FOR EACH ROW
WHEN (n.age < o.age)
SET n.age = o.age;

Statement- vs. row-level triggers

Why are both needed?
- Certain triggers are only possible at statement level
  - if the number of users inserted by this statement exceeds 100 and their average age is below 13, then ...
- Simple row-level triggers are easier to implement
  - Statement-level triggers require significant amount of state to be maintained in OLD TABLE and NEW TABLE
  - However, a row-level trigger gets fired for each row, so complex row-level triggers may be less efficient for statements that modify many rows

System issues

- Recursive firing of triggers
  - Action of one trigger causes another trigger to fire
  - Can get into an infinite loop
    - Some DBMS leave it to programmers/database administrators (e.g., PostgreSQL)
    - Some restrict trigger actions (e.g., Oracle)
    - Many set a maximum level of recursion (e.g., 16 in DB2)

- Interaction with constraints (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 differ)
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: members of Jessica's Circle
  - CREATE VIEW JessicaCircle AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
    WHERE gid = 'jes');
  - Tables used in defining a view are called "base tables"
    - User and Member above
  - To drop a view
    - DROP VIEW JessicaCircle;

Using views in queries

- Example: find the average popularity of members in Jessica's Circle
  - SELECT AVG(pop) FROM JessicaCircle;
  - To process the query, replace the reference to the view by its definition
  - SELECT AVG(pop)
    FROM (SELECT * FROM User
    WHERE uid IN
    (SELECT uid FROM Member
    WHERE gid = 'jes'))
    AS JessicaCircle;

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
- To provide a uniform interface for different implementations or sources
  - Real database applications use tons of views

Modifying views

- Does it even make sense, since views are virtual?
- It does make sense if we want users to really see views as tables
- Goal: modify the base tables such that the modification would appear to have been accomplished on the view

A simple case

CREATE VIEW UserPop AS
SELECT uid, pop FROM User;

DELETE FROM UserPop WHERE uid = 123;
translates to:
DELETE FROM User WHERE uid = 123;
An impossible case

CREATE VIEW PopularUser AS
SELECT uid, pop FROM User
WHERE pop >= 0.8;

INSERT INTO PopularUser
VALUES (987, 0.3);

No matter what we do on User, the inserted row will not be in PopularUser

A case with too many possibilities

CREATE VIEW AveragePop(pop) AS
SELECT AVG(pop) FROM User;

UPDATE AveragePop SET pop = 0.5;

Set everybody’s pop to 0.5?

Adjust everybody’s pop by the same amount?

Just lower Jessica’s pop?

SQL92 updateable views

• More or less just single-table selection queries
  • No join
  • No aggregation
  • No subqueries

• Arguably somewhat restrictive
• Still might get it wrong in some cases
  • See the slide titled “An impossible case”
  • Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications

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 later in this course!
• An index on $R.A$ can speed up accesses of the form
  • $R.A = \text{value}$
  • $R.A > \text{value}$ (sometimes; depending on the index type)
• An index on $(R.A_1, ..., R.A_n)$ can speed up
  • $R.A_1 = \text{value}_1$ AND $R.A_2 = \text{value}_2$
  • $(R.A_1, ..., R.A_n) > (\text{value}_1, ..., \text{value}_n)$ (again depends)
• Ordering or index columns is important—is an index on $(R.A.R.B)$ equivalent to one on $(R.B.R.A)$?
• How about an index on $R.A$ plus another on $R.B$?

Examples of using indexes

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

• SELECT * FROM User, Member
  WHERE User.uid = Member.uid AND Member.gid = 'jes';
  • With an index on Member.gid or (gid, uid): find relevant Member rows directly
  • With an index on User.uid: for each relevant Member row, directly look up User rows with matching uid
• Without it: for each Member row, scan the entire User table for matching uid
  • Sorting could help

INSTED OF triggers for views

CREATE TRIGGER AdjustAveragePop
INSTEAD OF UPDATE ON AveragePop
REFERENCING OLD ROW AS o,
NEW ROW AS n
FOR EACH ROW
UPDATE User
SET pop = pop + (n.pop - o.pop);

• What does this trigger do?
Creating and dropping indexes in SQL

CREATE [UNIQUE] INDEX indexname ON tablename (columnname1,...,columnname_n);

• With UNIQUE, the DBMS will also enforce that (columnname1,...,columnname_n) is a key of tablename

DROP INDEX indexname;

• 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 need to be maintained when data is updated
• Indexes have one more level of indirection

Optimal index selection depends on both query and update workload and the size of tables

• Automatic index selection is now featured in some commercial DBMS

SQL features covered so far

• Query
• Modification
• Constraints
• Triggers
• Views
• Indexes