SQL:
Triggers, Views, Indexes

Introduction to Database Management
CS348 Spring 2021
Announcements (Tue., May 25)

• **Milestone 0** - Project groups are formed by tonight!
  • Questions are welcome on Piazza
  • Watch out for updates

• **Assignment #1** due next Mon (May 31, 11pm)
  • Servers:
    • ubuntu2004-002.student.cs.uwaterloo.ca
    • ubuntu2004-004.student.cs.uwaterloo.ca
  • Setup db2 environment:
    • $ source ~cs348/public/db2profile
SQL

• Basic SQL (queries, modifications, and constraints)

• Intermediate SQL
  • Triggers
  • Views
  • Indexes

• Advanced SQL
  • Programming
  • Recursive queries

this week
Still remember “referential integrity”?

Example: Member.uid references User.uid

• Delete or update a User row whose uid is referenced by some Member row
  • Multiple Options (in SQL)

CREATE TABLE Member
    (uid DECIMAL(3,0) NOT NULL
    REFERENCES User(uid)
    ON DELETE CASCADE,
    .....
    )

<table>
<thead>
<tr>
<th>User</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>name</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
</tr>
<tr>
<td>857</td>
<td>...</td>
</tr>
<tr>
<td>857</td>
<td>...</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
</tr>
<tr>
<td>789</td>
<td>Nelson</td>
</tr>
</tbody>
</table>

Option 1: Reject

Option 2: Cascade (ripple changes to all referring rows)
Can we generalize it?

Delete/update a User row

Whether its uid is referenced by some Member row

Yes: reject/delete cascade/null

Referential constraints

Event

Condition

Action

Some user’s popularity is updated

Whether the user is a member of “S group” and pop drops below 0.5

Yes: kick that user out of S group

Data Monitoring
Triggers

- A **trigger** is an event-condition-action (ECA) rule
  - When **event** occurs, test **condition**; if condition is satisfied, execute **action**

CREATE TRIGGER PickySGroup
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 = 'sgroup'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'sgroup';
Trigger option 1 – possible events

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

CREATE TRIGGER PickySGroup
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 = 'sgroup'))
  DELETE FROM Member
  WHERE uid = newUser.uid AND gid = 'sgroup';
Trigger option 2 – timing

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

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;
Trigger option 3 – granularity

- Granularity—trigger can be activated:
  - FOR EACH ROW modified

```sql
CREATE TRIGGER PickySGroup
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 = 'sgroup'))
  DELETE FROM Member
  WHERE uid = newUser.uid AND gid = 'sgroup';
```
Trigger option 3 – granularity

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

CREATE TRIGGER PickySGroup2
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
DELETE FROM Member
  WHERE gid = 'sgroup'
  AND uid IN (SELECT uid
                FROM newUsers
                WHERE pop < 0.5);
Trigger option 3 – granularity

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

```
CREATE TRIGGER PickySGroup2
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
  DELETE FROM Member
  WHERE gid = 'sgroup'
  AND uid IN (SELECT uid
               FROM newUsers
               WHERE pop < 0.5);
```
## Transition variables/tables

- **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

<table>
<thead>
<tr>
<th>Event</th>
<th>Row</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>old r; old t</td>
<td>old t</td>
</tr>
<tr>
<td>Insert</td>
<td>new r; new t</td>
<td>new t</td>
</tr>
<tr>
<td>Update</td>
<td>old/new r; old/new t</td>
<td>old/new t</td>
</tr>
</tbody>
</table>

**AFTER Trigger**

<table>
<thead>
<tr>
<th>Event</th>
<th>Row</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>old/new r</td>
<td>-</td>
</tr>
<tr>
<td>Insert</td>
<td>new r</td>
<td>-</td>
</tr>
<tr>
<td>Delete</td>
<td>old r</td>
<td>-</td>
</tr>
</tbody>
</table>

**BEFORE Trigger**
Statement- vs. row-level triggers

• 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

• Exercise 1: However, can you think of a case when a row-level trigger may be less efficient?

• Exercise 2: Certain triggers are only possible at statement level. Can you think of an example?
System issues

• Recursive firing of triggers
  • Action of one trigger causes another trigger to fire
  • Can get into an infinite loop

• Interaction with constraints (tricky to get right!)
  • When to check if a triggering event violates constraints?
    • After a BEFORE trigger
    • Before an AFTER trigger
    • (based on db2, other DBMS may differ)

• Be best avoided when alternatives exist
SQL features covered so far

• Basic SQL

• Intermediate SQL
  • Triggers
  • Views
Views

• A **view** is like a “virtual” table
  • Defined by a query, which describes how to compute the view contents on the fly
  • Stored by DBMS instead of view contents
  • Can be used in queries just like a regular table

```
CREATE VIEW SGroup AS
  SELECT * FROM User
  WHERE uid IN (SELECT uid
                FROM Member
                WHERE gid = 'sgroup');

SELECT AVG(pop) FROM SGroup;
SELECT AVG(pop)
FROM (SELECT * FROM User
      WHERE uid IN (SELECT uid
                    FROM Member
                    WHERE gid = 'jes'))
     AS SGroup;
SELECT MIN(pop) FROM SGroup;
SELECT ... FROM SGroup;
DROP VIEW SGroup;
```
Why use views?

• To hide complexity from users

• To hide data from users

• Logical data independence

• To provide a uniform interface for different implementations or sources
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

• Set everybody’s `pop` to 0.5?
• Adjust everybody’s `pop` by the same amount?
• Just lower one user’s `pop`?

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

UPDATE AveragePop SET pop = 0.5;
```
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” (slide 20)
  • Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications
INSTEAD OF triggers for views

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

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?

  UPDATE AveragePop SET pop = 0.5;
INSTEAD OF triggers for views

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

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?

UPDATE AveragePop SET pop = 0.5;
SQL features covered so far

• Basic SQL

• Intermediate SQL
  • Triggers
  • Views
  • Indexes
Motivating examples of using indexes

SELECT * FROM User WHERE name = 'Bart';

• Can we go “directly” to rows with name='Bart' instead of scanning the entire table?
  → index on User.name

SELECT * FROM User, Member
WHERE User.uid = Member.uid AND Member.gid = 'sgroup';

• Can we find relevant Member rows “directly”?
  → index on Member.gid or (gid, uid)
• For each relevant Member row, can we “directly” look up User rows with matching uid
  → index on User.uid
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!

• CREATE [UNIQUE] INDEX indexname ON tablename(columnname₁,...,columnnameₙ);
  • With UNIQUE, the DBMS will also enforce that \{columnname₁, ..., columnnameₙ\} is a key of tablename

• DROP INDEX indexname;

• Typically, the DBMS will automatically create indexes for PRIMARY KEY and UNIQUE constraint declarations
Indexes

• 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 $(R.A_1, ..., R.A_n)$ can speed up
  • $R.A_1 = value_1 \land \ldots \land R.A_n = value_n$
  • $(R.A_1, ..., R.A_n) > (value_1, ..., value_n)$ (again depends)

Questions (lecture 12):
  ✐ Ordering of 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 $R.B$?
  ✐ More indexes = better performance?
SQL features covered so far

Basic & Intermediate SQL
• Query
• Modification
• Constraints
• Triggers
• Views
• Indexes

Next: Recursion, programming