SECTION 6

CSE 444 section, February 11, 2011

Today’s agenda

- Project 3 Introduction
- PostgreSQL Triggers
- Database access control
- Optimistic Concurrency Control (if there’s time)

- Reminder: HW 2 due Friday at 11pm

Project 3

You’re a database administrator at the UW Medical Center.

You want to:

1. Improve efficiency of popular queries (tune db)
2. Maintain integrity with a trigger
   - If a nurse deletes a product from the supplies table
     - Keep it in the Product and Stock table until the stock goes to zero and then remove it.
3. Make sure doctors, nurses, administrators, public see only information that they are allowed to
   - Create users, roles, views, grant access
4. Keep system secure

Files

- Import-database.sql
  - Import patient/doctor/supplies/etc data
- Import-voters.sql
  - Import voter data (for part 3, security)
- DatabaseGenerator.java
  - Generates the data for the database.
- TestQueries.java
  - Runs queries (use this to measure the efficiency of your database tuning in part 1)
- Sample-trigger.sql
  - Look here when you start writing your trigger!

References

Readings from textbook:

- Triggers: 7.5, 8.2.3
- Access control: 10.1

Also: Postgres references linked from project 3 instructions

What is a trigger?

Trigger: a procedure run automatically by the DBMS in response to an update to the database

Trigger = Event + Condition + Action

A trigger in English

Whenever we update a row in table Product…	EVENT
If the row’s *price* attribute has been reduced…	CONDITION
Then record the product’s *name* and *discount* in table Promotions	ACTION

EXAMPLE: ROW-LEVEL TRIGGER

```
CREATE TRIGGER InsertPromotions AFTER UPDATE OF price ON
Product

REFERENCING
OLD AS x
NEW AS y

FOR EACH ROW
WHEN (x.price > y.price)
INSERT INTO Promotions(name, discount)
VALUES x.name,
(x.price-y.price)*100/x.price
```

Event

Condition

Action

Events

INSERT, DELETE, UPDATE

Trigger can run:

- BEFORE the event
- AFTER the event
- INSTEAD OF the event

Scope

FOR EACH ROW = trigger executed for every row affected by the update

REFERENCING OLD ROW AS *old_name*,

NEW ROW AS *new_name*

FOR EACH STATEMENT = trigger executed once for the entire statement

REFERENCING OLD TABLE AS *old_name*,

NEW TABLE AS *new_name*

Statement-level trigger

```
CREATE TRIGGER max_avg_price
AFTER UPDATE OF price ON Product

REFERENCING
OLD TABLE AS OldStuff,
NEW TABLE AS NewStuff

FOR EACH STATEMENT
WHEN (1000 < (SELECT AVG (price) FROM Product))
BEGIN
    DELETE FROM Product
    WHERE (name, price, company) IN NewStuff;
    INSERT INTO Product
    (SELECT * FROM OldStuff);
END;
```

Triggers in standard SQL

Event = INSERT, DELETE, UPDATE

Condition = any WHERE condition

- Can refer to the old and new values

Action = more inserts, deletes, updates

- May result in cascading effects!

Trigger pros and cons

- Triggers are very powerful!
    - Enforce data correctness (integrity constraints)
    - Alert users/admins of strange patterns
    - Log events
- But hard to understand (ex. recursive triggers)
- Syntax is vendor specific, varies significantly
    - As we will see next...

Triggers in PostgreSQL

- No conditions
  - Instead, use IF/ELSE in action
- Use Postgres' procedural SQL – PL/pgSQL
  - Different syntax from the standard
- 2-part definition
  1. Define action as a PL/pgSQL function
  2. Create trigger that calls the action function

Postgres trigger example

Example table: employee salaries

```
CREATE TABLE emp (
   empname varchar(100),
   salary integer,
   last_date timestamp,
   last_user varchar(100));
```

Want to:

- Reject negative salaries
- Record user, date of each update

Based on PL/pgSQL example in Postgres docs: http://www.postgresql.org/docs/8.4/static/plpgsql-trigger.html

Defining the triggered action

```
-- Register PL/pgSQL with the database; do this only once
CREATE LANGUAGE plpgsql;

CREATE FUNCTION emp_stamp() RETURNS trigger AS
$$
   BEGIN
      IF NEW.salary < 0 THEN
         RAISE EXCEPTION 'Salary must be non-negative';
      END IF;

      NEW.last_date := current_timestamp;
      NEW.last_user := current_user;
      RETURN NEW;
   END;
$$ LANGUAGE plpgsql;
```

Creating the trigger

```
CREATE TRIGGER emp_stamp
BEFORE INSERT OR UPDATE ON emp

FOR EACH ROW
EXECUTE PROCEDURE emp_stamp();
```

SQL authentication

Many SQL DBs have 2 access control concepts:

- Role
  - A group with specific privileges (ex. data_entry, customer_support)
- User
  - An individual (ex. John, Fred, my_program)

PostgreSQL: a “user” is just a role that can log in

Access control example

Customers

name	**address**	**balance**
Mary	Houston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

Fred is not allowed to see this

Fred is allowed to see this

```
CREATE VIEW PublicCustomers
SELECT name, address
FROM Customers
```

Postgres access control example

```
-- Set up Fred's account - you need CREATEROLE privilege for this!
CREATE USER fred WITH PASSWORD 'fredpass';

-- Prevent Fred from reading the base table
REVOKE ALL PRIVILEGES ON Customers FROM fred;

-- Create the view that contains what Fred can access
CREATE VIEW PublicCustomers AS
    SELECT name, address
    FROM Customers;

-- Allow Fred to read the view
GRANT SELECT ON PublicCustomers TO fred;
```

Alternate approach without views

```
-- Set up Fred's account - you need CREATEROLE privilege for this!
CREATE USER fred WITH PASSWORD 'fredpass';

-- Prevent Fred from reading the base table
REVOKE ALL PRIVILEGES ON Customers FROM fred;

-- Allow Fred to read only the name and address columns
GRANT SELECT (name, address) ON Customers TO fred;
-- Now SELECT * FROM Customers fails,
-- but SELECT name or SELECT address works
```

Restricting access to rows

Customers

name	address	balance
Mary	Houston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

Repo men are not allowed to see balances > 0

```
CREATE VIEW BadCreditCustomers
SELECT *
FROM Customers
WHERE balance < 0
```

Row-level restrictions need views!

```
-- Create the repo men's group, and make Fred and John repo men
CREATE ROLE repo_men;
GRANT repo_men TO fred;
CREATE USER john WITH PASSWORD 'johnpass'
   IN ROLE repo_men;

-- Create the view that repo men can access
CREATE VIEW BadCreditCustomers AS
   SELECT *
   FROM Customers
   WHERE balance < 0;
GRANT SELECT ON BadCreditCustomers TO repo_men;
-- Must use view, because GRANT doesn't support WHERE clause
```

Optimistic Concurrency Control

- Timestamps
  - Key Idea: The timestamp order defines the serialization order
  - Scheduler maintains:
    - TS(T) for all transactions T
    - RT(X), WT(X), C(T)
- Multiversion Timestamps
  - Keep multiple version of each data element along with the write timestamp
  - Will reduce number of aborts due to read-too-late problem
- Validation
  - Transaction informs schedule of its read and write sets before it validates