ADVANCED SQL DDL

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 10

Advanced SQL DDL

- Last time, covered stored procedures and user-defined functions (UDFs)
  - Relatively simple but powerful mechanism for extending capabilities of a database
  - Most databases support these features (in different ways, of course…)
- Today, will cover three more advanced features of SQL data definition
  - Triggers
  - Materialized views (briefly)
  - Security constraints in databases

Triggers

- Triggers are procedural statements executed automatically when a database is modified
  - Usually specified in procedural SQL language, but other languages are frequently supported
- Example: an audit log for bank accounts
  - Every time a balance is changed, a trigger can update an "audit log" table, storing details of the change
    - e.g. old value, new value, who changed the balance, and why
- Why not have applications update the log directly?
  - Could easily forget to update audit log for some updates!
  - Or, a malicious developer might leave a back-door in an application, allowing them to perform unaudited operations

Triggers (2)

- If the database handles audit-log updates automatically and independently:
  - Application code doesn't become more complex by introducing audit functionality
  - Audit log will be a more trustworthy record of modifications to bank account records
- Triggers are used for many other purposes, such as:
  - Preventing invalid changes to table data
  - Automatically updating timestamp values, derived attributes, etc.
  - Executing business rules when data changes in specific ways
    - e.g. place an order for more parts when current inventory dips below a specific value
  - Replicating changes to another table, or even another database

Trigger Mechanism

- DB trigger mechanism must keep track of two things:
- When is the trigger actually executed?
  - The event that causes the trigger to be considered
  - The condition that must be satisfied before the trigger will execute
    - (Not every database requires a condition on triggers…)
- What does the trigger do when it's executed?
  - The actions performed when the trigger executes
- Called the event-condition-action model for triggers

When Triggers Execute

- Databases usually support triggering on inserts, updates, and deletes
- Can't trigger on selects
  - Implication: Can't use triggers to audit or prevent read-accesses to a database (bummer)
- Commercial databases also support triggering on many other operations
  - Data-definition operations (create/alter/drop table, etc.)
  - Login/logout of specific users
  - Database startup, shutdown, errors, etc.
- For simplicity, will limit discussion to DML triggers only

When Triggers Execute

- Can typically execute the trigger before or after the triggering DML event
  - Usually, DDL/user/database triggering events only run the trigger *after* the event (pretty obvious)
  - "Before" triggers can abort the DML operation, if necessary
- Some DBs also support "instead of" triggers
  - Execute trigger instead of performing the triggering operation
- Triggers are row-level triggers or statement-level triggers
  - A row-level trigger is executed for every single row that is modified by the statement
    - *(…as long as the row satisfies the trigger condition, if specified…)*
  - A statement-level trigger is executed once for the entire statement

Trigger Data

- Row-level triggers can access the old and new version of the row data, when available:
  - Insert triggers only get the new row data
  - Update triggers get both the old and new row data
  - Delete triggers only get the old row data
- Triggers can also access and modify other tables
  - e.g. to look up or record values during execution

Trigger Syntax

- SQL:1999 specifies a syntax for triggers
  - Discussed in the textbook, section 5.3
- Again, wide variation from vendor to vendor
  - Oracle and DB2 are similar to SQL99, but not identical
    - (triggers always seem to involve vendor-specific features)
  - SQLServer, Postgres, MySQL all have different features
  - Constraints on what triggers can do also vary widely from vendor to vendor
- Will focus on MySQL trigger syntax, functionality

Trigger Example: Bank Overdrafts

- Want to handle overdrafts on bank accounts
- If an update causes a balance to go negative:
  - Create a new loan with same ID as the account number
  - Set the loan balance to the negative account balance
    - *(…the account balance went negative…)*
  - Need to update `borrower` table as well!
- Needs to be a row-level trigger, executed before updates to the `account` table
  - If database supports trigger conditions, only trigger on updates when account balance < 0

SQL99/Oracle Trigger Syntax

□ Book uses SQL:1999 syntax, similar to Oracle/DB2

```
CREATE TRIGGER trg_overdraft AFTER UPDATE ON account
REFERENCING NEW ROW AS nrow
FOR EACH ROW WHEN nrow.balance < 0
BEGIN ATOMIC
    INSERT INTO loan VALUES (nrow.account_number,
                             nrow.branch_name,
                             -nrow.balance);

    INSERT INTO borrower
        (SELECT customer_name, account_number
         FROM depositor AS d
         WHERE nrow.account_number = d.account_number);

    UPDATE account AS a SET balance = 0
        WHERE a.account_number = nrow.account_number;
END
```

MySQL Trigger Syntax

- MySQL has more limited trigger capabilities
  - Trigger execution is only governed by events, not conditions
    - Workaround: Enforce the condition within the trigger body
  - Old and new rows have fixed names: **`OLD`**, **`NEW`**
- Change the overdraft example slightly:
  - Also apply an overdraft fee! *"Kick 'em while they're down!"*
  - What if the account is already overdrawn?
    - Loan table would already contain a record for the overdrawn account…
    - Borrower table would already contain records for the loan, too!
    - Previous version of trigger would cause a duplicate key error!

MySQL **INSERT** Enhancements

- MySQL has several enhancement to the **INSERT** command
  - (Most databases provide similar capabilities)
- Try to insert a row, but if key attributes are same as another row, simply don't perform the insert:

```
INSERT IGNORE INTO tbl ...;
```

- Try to insert a row, but if key attributes are same as another row, update the existing row:

```
INSERT INTO tbl ... ON DUPLICATE KEY
    UPDATE attr1 = value1, ...;
```

- Try to insert a row, but if key attributes are same as another row, replace the old row with the new row
  - If key is not same as another row, perform a normal **INSERT**

```
REPLACE INTO tbl ...;
```

MySQL Trigger Syntax (2)

```
CREATE TRIGGER trg_overdraft BEFORE UPDATE ON account FOR EACH ROW
BEGIN
    DECLARE overdraft_fee NUMERIC(12, 2) DEFAULT 30;
    DECLARE overdraft_amt NUMERIC(12, 2);

    -- If an overdraft occurred then handle by creating/updating a loan.
    IF NEW.balance < 0 THEN
        -- Remember that NEW.balance is negative.
        SET overdraft_amt = overdraft_fee - NEW.balance;

        INSERT INTO loan (loan_number, branch_name, amount)
            VALUES (NEW.account_number, NEW.branch_name, overdraft_amt)
        ON DUPLICATE KEY UPDATE amount = amount + overdraft_amt;

        INSERT IGNORE INTO borrower (customer_name, loan_number)
            SELECT customer_name, account_number FROM depositor
            WHERE depositor.account_number = NEW.account_number;

        SET NEW.balance = 0;
    END IF;
END;
```

Trigger Pitfalls

- Triggers may *or may not* execute when you expect…
  - e.g. MySQL insert-triggers fire when data is bulk-loaded into the DB from a backup file
    - Databases usually allow you to temporarily disable triggers
  - e.g. truncating a table usually does not fire delete-triggers
- If a trigger for a commonly performed task runs slowly, it will kill DB performance
- If a trigger has a bug in it, it may abort changes to tables at unexpected times
  - The *actual* cause of the issue may be difficult to discern
- Triggers can write to other tables, which may also have triggers on them…
  - Not hard to create an infinite chain of triggering events

Alternatives to Triggers

- Triggers can be used to implement *many* complex tasks
- Example: Can implement referential integrity with triggers!
    - On all inserts and updates to referencing table, ensure that foreign-key column value appears in referenced table
        - If not, abort the operation!
    - On all updates and deletes to referenced table, ensure that value doesn't appear in referencing table
        - If it does, can abort the operation, or cascade changes to the referencing relation, etc.
- This is definitely slower than the standard mechanism ☺

Alternatives to Triggers (2)

- Can you use stored procedures instead?
  - Stored procedures usually have fewer limitations than triggers
    - Stored procs can take more detailed arguments, return values to indicate success/failure, have out-params, etc.
    - Can perform more sophisticated transaction processing
  - Trigger support is also very vendor-specific, so either implementation choice will have this limitation
- Typically, triggers are used in very limited ways
  - Update "row version" or "last modified timestamp" values in modified rows
  - Simple operations that don't require a great deal of logic
  - Database replication (sometimes)

Triggers and Summary Tables

- Triggers are sometimes used to compute summary results when detail records are changed
- Example: a table of branch summary values
  - e.g. (*branch_name*, *total_balances*, *total_loans*)
- Motivation:
  - If these values are used frequently in queries, want to avoid overhead of recomputing them all the time
- Idea: update this summary table with triggers
  - Anytime changes are made to `account` or `loan`, update the summary table based on the changes

Materialized Views

- Some databases provide materialized views, which implement such functionality
- Simple views usually treated as named SQL queries
  - i.e. a derived relation with the specified definition
- When a query refers to a simple view, database substitutes view's definition directly into the query
  - Benefit: allows optimization of the entire query
  - Drawback: if many queries reference a simple view, the same values will be computed again and again…

Materialized Views (2)

- Materialized views actually create a new table, populated by the results of the view definition
  - Queries can use values in the materialized view over and over, without recomputing
  - Database can perform optimized lookups against the materialized view, e.g. by using indexes
- Just one little problem:
  - What if the tables referenced by the view change?
  - Need to recompute contents of the materialized view!
  - Called <u>view maintenance</u>

Materialized View Maintenance

- If a database doesn't support materialized views:
  - Can perform view maintenance with triggers on the referenced tables
  - A very manual approach, but definitely an option for databases that don't support materialized views
    - e.g. Postgres, MySQL
- Databases with materialized views will perform view maintenance automatically
  - …*much* simpler than creating a bunch of triggers!
  - Typically provide many options, such as:
    - Immediate view maintenance – update contents after any change
    - Deferred view maintenance – update view on a periodic schedule

Materialized View Maintenance (2)

- A simple approach for updating materialized views:
  - Recompute entire view from scratch after every change!
  - Very expensive approach, especially if backing tables are changed frequently
- A better approach: incremental view maintenance
  - Using the view definition and the specific data changes applied to the backing tables, only update those parts of the view that are actually affected
- Again, DBs with materialized views will do this for you
- Can also do incremental view maintenance manually with triggers, but it can be complicated…

Authentication and Authorization

- Security systems must provide two major features
- Authentication (aka “A1”, “AuthN”, “Au”):
  - “I am who I say I am.”
- Authorization (aka “A2”, “AuthZ”, “Az”):
  - “I am allowed to do what I want to do.”
- Each component is useless without the other

User Authorization

- SQL databases perform authentication of users
  - Must specify username and password when connecting
  - Most DBMSes provide secure connections (e.g. SSL), etc.
- SQL provides an authorization mechanism for various operations
  - Different operations require different privileges in the database
  - Users can be granted privileges to perform necessary operations
  - Privileges can also be revoked, to limit available user operations

Basic SQL Privileges

- Most fundamental set of privileges:
  - **`SELECT`, `INSERT`, `UPDATE`, `DELETE`**
  - Allows (or disallows) user to perform specified action
  - User is granted access to perform specified operations on particular relations
- Simple syntax:

```
GRANT SELECT ON account TO banker;
```

  - User "banker" is allowed to issue queries against the *account* relation

Granting Privileges

- Can grant multiple privileges to multiple users

```
GRANT SELECT, UPDATE ON account
  TO banker, manager;
GRANT INSERT, DELETE ON account
  TO manager;
```

  - Bankers can view and modify account balances
  - Only managers can create or remove accounts
  - Must specify each table individually

All Users, All Privileges

- Can specify **PUBLIC** to grant privileges to all users
  - Also includes users added to DBMS in future

```
GRANT SELECT ON promotions TO PUBLIC;
```

- Can specify **ALL PRIVILEGES** to grant all privileges to a user

```
GRANT ALL PRIVILEGES ON account
  TO admin_lackey;
```

Column-Level Privileges

- For **`INSERT`** and **`UPDATE`** privileges, can constrain to specific *columns* of relations
  - **`UPDATE`**: can only update specified columns
  - **`INSERT`**: can only insert into specified columns
- Example: *employee* relation
  - Employees can only modify their contact info
  - Allow HR to manipulate *all* aspects of employees

```
GRANT UPDATE (home_phone, email) ON employee
  TO emp_user;
GRANT INSERT, UPDATE ON employee TO hr_user;
```

Revoking Privileges

- Can revoke privileges just as easily:

```
REVOKE priv1, ... ON relation
  FROM user1, ...;
```

  - Can specify a list of privileges, and a list of users
- With **INSERT** and **UPDATE**, can also revoke privileges on individual columns

Privileges and Views

- Users can be granted privileges on views
  - May differ from privileges on underlying tables
- When accessing a view:
  - Privileges on the *view* are checked, not the privileges on underlying tables
- Example: *employee* relation
  - Only HR can view all employee data
  - Employees can only view contact details

Example View Privileges

- SQL commands:

```
-- Start by disallowing all access to employee
REVOKE ALL PRIVILEGES ON employee TO PUBLIC;

-- Only allow hr_user to access employee relation
GRANT ALL PRIVILEGES ON employee TO hr_user;

-- View for "normal" employees to access
CREATE VIEW directory AS
    SELECT emp_name, email, office_phone
    FROM employee;
GRANT SELECT ON directory TO emp_user;
```

- When employees issue queries against *directory*, DB only checks *directory* privileges

View Processing

- As stated before, databases usually treat views as named SQL queries
  - Database substitutes view's definition directly into queries that reference the view
- SQL engine performs authorization *before* this process occurs
  - DB verifies access permissions on referenced views, and then substitutes view definitions into the query plan
  - Allows DB to support different access constraints on views, vs. their underlying tables

Other Privileges

- Many other privileges in SQL
  - **EXECUTE** grants privilege to execute a function or stored procedure
  - **CREATE** grants privilege to create tables, views, other schema objects
  - **REFERENCES** grants privilege to create foreign key or **CHECK** constraints
  - Most DBMSes provide several others, too
    - PostgreSQL has 11 permissions; MySQL has 27
    - Oracle has nearly 200 different permissions!

REFERENCES Privilege

- Foreign key constraints limit what users can do
  - Rows in referencing relation limit update and delete operations in referenced relation
  - A user adding a foreign key constraint can disallow these operations for all users!
- Must have the **REFERENCES** privilege to create foreign keys
- **REFERENCES** requires both a relation and some attributes to be specified
  - May create foreign keys involving those attributes

Passing On Privileges

- Users can't automatically grant their own privileges to other users
- Must explicitly allow this:

```
GRANT SELECT ON directory TO emp_user
    WITH GRANT OPTION;
```

  - **`WITH GRANT OPTION`** clause allows privileges to be passed on
- Can lead to confusing situations:
  - If **`alex`** grants a privilege to **`bob`**, then **`alex`** has that privilege revoked, should it affect **`bob`**?
  - If **`alex`** and **`bob`** both grant a privilege to **`carl`**, then **`alex`** revokes that privilege, does **`carl`** still have the privilege?
- Typically, databases implement simple solutions to these kinds of problems

Authorization Notes

- SQL authorization mechanism is very rich
- Still has a number of shortcomings
  - Can’t grant/revoke privileges on per-tuple basis
    - e.g. “I can see only the rows in the *account* relation corresponding only to my bank accounts.”
    - (If there were **`SELECT`** triggers, we could implement this…)
    - (Or, you could emulate this with table-returning functions…)
  - Significant variations in security models implemented by various databases

Authorization Notes (2)

- Most applications don't rely heavily on DB authorization
  - Application can implement a broad range of authorization schemes, but implementation complexity increases
  - Web applications are primary example of this
  - Database access layer typically has only one user, with full access and modification privileges
- Application performs authentication/authorization itself
  - Access-checks are sprinkled throughout application code; easy to introduce security holes! (e.g. PHP applications)
  - App-servers with declarative security specifications greatly mitigate this problem (e.g. JavaEE platform security)

Authorization Notes (3)

- Best to employ SQL auth mechanism in *some* way…
  - Declarative security specifications
  - Database simply won't allow access to privileged data, or unauthorized changes to schema
- For large, important database apps, definitely want to explore using SQL authorization features
  - At the least, create a DBMS user for each user-role that application supports
  - An "admin" user for administrators in the application, with fewer restrictions
  - A very restricted "common user" for end-users
  - Greatly reduces the dangers of SQL-based attacks

Next Time

- Last major topic for SQL data definition: indexes
  - Used to facilitate *much* faster database lookups
- Will also briefly discuss DB storage mechanisms, and how this affects query performance