Views, Access Control, SQL Procedures

CSC 321/621 – 3/22/2012

Views

So far, database = collection of relations (tables)

- Relations explicitly defined with CREATE TABLE are called “base relations”

Sometimes useful to present “virtual tables” – tables that are provide an alternative *view* of the data

- Such relations may not actually exist in storage but can be dynamically generated as need be
- Technical definition from book: *The dynamic result of one or more relational operations operating on the base relations to produce another relation.*

Section 4.4 of text

Views: DBMS Implementation

- Remember, views are dynamic tables – they are never “permanently stored” but instead are managed at runtime
  - The definition of the view is stored itself, but not the data
- Two common techniques used by DBMS to support dynamic generation
  - View Resolution
  - View Materialization

Views: Resolution vs. Materialization

A simple way to think about the differences:

Let V be a view defined by a query X on database D.

$V = X(D)$

A query Y on view V is then:

$Y(V) = Y(X(D))$

Materialization is actually generating X(D) and then applying the query Y to the result

Resolution is determining the composition function ZY and then applying that composition function to D

Views: Resolution vs. Materialization

- At query time, materialization is much faster
  - No translation
- View maintenance, however, can be expensive
  - Have to check and see
    - A) whether a given statement can affect the view
    - B) if so, how the view should be updated

Views: View Maintenance

- An example of *view maintenance*:

Assume the following view is defined:

CREATE VIEW HighGPAMajors AS SELECT DISTINCT major FROM Student WHERE GPA >= 3.5;

Student

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

HighGPAMajors

major
CSC

Views: View Maintenance

- What types of inserts/update changes to the Student table will actually affect the HIGHGPAMajors view?
  - Adding a CSC major with a GPA < 3.5?
  - Adding a CSC major with GPA >= 3.5?
  - Adding a MTH major with GPA < 3.5?
  - Adding a MTH major with GPA >= 3.5?

Student

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

HighGPAMajors

major
CSC

Views: View Maintenance

- What types of inserts/update changes to the Student table will actually affect the HIGHGPAMajors view?
  - Adding a CSC major with a GPA < 3.5? NO (doesn't pass constraint)
  - Adding a CSC major with GPA >= 3.5? NO (already in view)
  - Adding a MTH major with GPA < 3.5? NO (doesn't pass constraint)
  - Adding a MTH major with GPA >= 3.5? YES

Student

Make our decision based only on insert values and HighGPAMajors view

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

HighGPAMajors

major
CSC

Views: View Maintenance

- What types of deletes/update changes will actually affect the HIGHGPAMajors view?

Student

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

HighGPAMajors

major
CSC

Views: View Maintenance

- What types of deletes/update changes to the Student table will actually affect the HIGHGPAMajors view?
  - Only deletes/changes to the last associated entry meeting the criteria
    - The lone CSC student with GPA >= 3.5 withdraws (out of frustration!) or has GPA drop below 3.5

This decision based only on insert/update/delete values and Student table

Student

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

HighGPAMajors

major
CSC

Views: Reading vs. Writing

- Views are “temporary tables” resulting from queries applied to another set of tables
- “Reading from” (querying) a view is essentially querying the underlying tables
  - Changes to the underlying tables are reflected in queries on the view

- Should we be able to “write into” (update a row in/insert into/delete from) a view?
  - Should changes to the view update the underlying tables?

Views: Updating Views

- Again assume V = X(D) (view V is a result of query X on D)
- Let U be an update operation on V, U(V)

  Then U(V) = U(X(D))

X(D) does not really exist, however, so it can't be physically updated, so we need to find update U' that allows U(X(D)) = X(U'(D))

Views: Updating Views

- DBMS's do allow updates through views, but support is limited in SQL

- An example of a potential problem:

MajorCount

major	cnt
CSC	2
MTH	1

```
CREATE VIEW MajorCount (major,cnt) AS
SELECT major, COUNT(*) FROM Student
GROUP BY major;
```

I should not be able to add a major, or update the count for a given major directly into this view. There has to be information added back to the Student table which I just don't know.

Views: Updating Views

- Essentially, to allow updating through a view, SQL requires ability to exactly map from a tuple in our view relation to a tuple in a underlying relation to be able to apply an update:
- Common problematic cases that prevent this:
  - Can't have used DISTINCT in the defining queries
    - Using DISTINCT discards rows
  - There is no GROUP BY or HAVING clause in the defining query
    - These cause rows to collapse together
  - Every element (column) in the SELECT list of the query defining the view is a real column from the underlying relation (not an aggregate, constant, or expression)
    - Using aggregate, constant, expression modify values so can't map back to an original row
  - The FROM clause in the defining query specifies exactly one table, so not a result of JOIN, INTERSECTION, UNION, DIFFERENCE (recursively holds for views based off views)
    - Combining tables prevents determining singular source

Views: Updating Views

- Assume GPA values can't be NULL. We should not be able to add students into CSCStudentForDirectory table (we can't fill in GPA!)
  - This was not one of our previous conditions, but is also problematic.

```
CREATE VIEW CSCStudentForDirectory
(StudentID,LastName,FirstName,Class) AS
SELECT studentId,lastName,firstName,year
FROM CSCStudent;
```

Student

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1129	Jones	Douglas	3	MTH	2.9
1145	Brady	Susan	4	CSC	3.8

CSCStudentForDirectory

Student ID	Last Name	First Name	Class
1123	Smith	Robert	4
1145	Brady	Susan	4

Views: Migrating Rows

- Rows exist in a view because they satisfied the defining query
  - Changes made to the view can cause rows to be added to or removed from view
    - “Migrating Rows”
  - Note we aren’t talking about changes made directly to the underlying tables, just changes made directly on the view
    - So, this discussion only applies to updateable views – views that don’t violate the constraints on the previous slides and support one-to-one mapping between view rows and underlying table rows
- Remember the CREATE view syntax?
  - CREATE VIEW ViewName [(newColumnName [,…])] AS subselect [WITH[CASCADED | LOCAL] CHECK OPTION]
  - The WITH CHECK OPTION can affect how migrating rows caused by changes to the view are handled

Views: Migrating Rows

- WITH CHECK OPTION prohibits a row from migrating OUT of a view by rejecting the update.
  - An update to Susan Brady's major can not be performed through this View, even though this table supports updates (1..1 mapping with Student); it must be done on Student proper.

CSCStudent

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1145	Brady	Susan	4	CSC	3.8

Views: Migrating Rows

- WITH CHECK OPTION prohibits a row from migrating OUT of a view by rejecting the update.
  - Insertion of an ENGLISH major through this view is also prevented (it would be "added to" the view, then to the underlying table, then immediately pulled back out of the view)

CSCStudent

studentID	lastName	firstName	year	major	GPA
1123	Smith	Robert	4	CSC	3.5
1145	Brady	Susan	4	CSC	3.8

Views: Migrating Rows

- WITH *CASCADED* CHECK OPTION applied to view V
  - Prevents updates on V and any views based off of V which would cause the row to disappear from V
    - Dealing with Views based off of V ➔ cascading
- WITH *LOCAL* CHECK OPTION applied to view V
  - Prevents updates on V and any views based off V which would cause the row to disappear from V, *unless the row would also be caused to be removed from the (view or table) underlying the view V*
    - Dealing with Views based off of V ➔ cascading
    - Linking what happens with V to what directly underlies V ➔ local check

Views: Review of Advantages/Disadvantages

- Advantages:
  - Data independence – can present an unchanging virtual view of the system, even if underlying sources change
  - Security/convenience/customization – can use access control to limit users to seeing particular views instead of whole database; Users aren't confused by totality of data; Can provide tailor-made versions of data to users (custom column names, aggregated information, etc)
  - Automatic currency – changes to bases tables propagate to views
  - Integrity preservation – techniques in place to ensure that updates directly view are consistent with view definition

Views: Review of Advantages/Disadvantages

- Disadvantages:
  - Not all views are update-able with SQL (so not really equivalent to relations)
  - Changes to underlying tables (adding columns, changing attribute types) do not propagate to the view – as the STRUCTURE of the view is only defined once
  - Cost overhead of doing view resolution on-the-fly (suffer when doing query) or view maintenance (suffer when making updates to database)

Database Access Control

- SQL supports access control to database objects
- Three parts to supporting access control:
  - Who is doing the acting (authorization identifier)
  - What objects can they act on (ownership)
  - What can they do to objects (privileges)

Database Access Control

- Privileges:
  - SELECT: Able to query
  - INSERT, DELETE, UPDATE: Able to modify tuples
  - REFERENCES: Ability reference columns in an integrity constraint
  - INSERT, UPDATE, REFERENCES can be assigned down to the column level
  - There are also CREATE, DROP (structural) privileges

Database Access Control

- Ownership:
  - The DBMS user that creates a table is the default OWNER of the table
  - Owners by default have the full set of privileges
  - That user can GRANT privileges to other users using the GRANT command as well as REVOKE privileges

Database Access Control

- Access Control For Views:
  - The issuer of a CREATE VIEW statement has ownership, but does not necessarily have the full set of privileges
    - To even create the view, the user must have SELECT privileges on all underlying tables and REFERENCES privileges for any columns named in the view
    - INSERT, UPDATE, and DELETE privileges for the VIEW are inherited iff the users holds those privileges for all underlying tables composing the view

Database Access Control: GRANT

- GRANT:
  - GRANT {PrivilegeList | ALL PRIVILEGES}
    ON ObjectName
    TO {AuthorizationList | PUBLIC}
    [WITH GRANT OPTION]

WITH GRANT OPTION allows the user to pass on privileges as well

Database Access Control: REVOKE

- REVOKE:
  - REVOKE [GRANT OPTION FOR] {PrivilegeList | ALL PRIVILEGES} ON ObjectName FROM {AuthorizationIDList | PUBLIC} [RESTRICT | CASCADE}
  - GRANT OPTION FOR removes the users ability to pass on access
  - REVOKE may take away the privileges that allowed an object (such as a View) to be created. That table is now considered abandoned.
    - RESTRICT will not revoke privileges if it would lead to abandoned tables
    - CASCADE will automatically drop any tables that become abandoned.

Database Access Control: MySQL Example

- MySQL access control incorporates both a user and the host they are connecting from:
  - I made a new user, droid, with password 321 and allowed him to connect from anywhere.
    - CREATE USER 'droid'@'%' IDENTIFIED BY 'csc321';
  - I allow that user to query and update all tables in the hotel database, but not to create or remove tables or change their structure
    - GRANT SELECT,INSERT,DELETE,UPDATE ON DBH.* TO 'droid'@'%' WITH GRANT OPTI ON;

SQL Syntax: Higher Level Instructions

- So far, our view of SQL has been
  - CREATE/DROPS (Structure)
  - INSERT/UPDATE/DELETE/SELECT (Data)

- SQL supports a number of higher level programming constructs which we can use to write and execute procedures on the DBMS server
  - Those procedures can access database data if desired

SQL: Stored Procedures

- Why Stored Procedures?
  - More complex management of data
  - Generic parameterizable interfaces to queries
  - Write & debug the code on the server side, make clients just call procedures
  - Support complex processing on inserts/updates/deletes (triggers)

SQL: MySQL Specific

- My examples are going to be MySQL specific
  - Other database implementations are slightly different
  - Chapters 12 and 18 of the MySQL documentation
- To cover:
  - Blocks, Variables, Assignments, Conditionals, Loops
- Lots of variations, will try to show high points

Stored Procedures/Functions

- Function: Has arbitrary number of input parameters, returns one value

- Procedure: Arbitrary number of input and output parameters, returns no values (except via setting of output parameters)

SQL Syntax: Simple Stored Function

Defining:

```
CREATE FUNCTION                    ** SQL
    answerToLife() RETURNS INT  ** funchead
        BEGIN                      ** start compound
            RETURN(42);   ** return(value)
        END;                       ** end compound
```

Executing:

```
SELECT return42();
```

SQL Syntax: Simple Stored Function

```
mysql> DELIMITER //
mysql> CREATE FUNCTION
    -> answerToLife() RETURNS INT
    -> BEGIN
    -> RETURN(42);
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> select answerToLife();
    -> //
+----------------+
| answerToLife() |
+----------------+
|             42 |
+----------------+
1 row in set (0.00 sec)
```

Because I want to use ; as part of my input (not to be interpreted as the end of my input), I temporarily change the delimiter for MySQL to be //

SQL Syntax: Simple Stored Function

```
CREATE FUNCTION
   reservationsMadeAtHotel(hn INT)
   RETURNS INT
         BEGIN
               DECLARE x INT;
               SELECT COUNT(*) INTO x FROM
                 booking WHERE hotelNumber=hn;
               RETURN(x);
         END;
```

New ideas:
* Variable declarations
* SELECT INTO
* Parameters

Functions can only return a singular value, not a result set – can't directly return SELECT result
(use INTO *x* to push into a declared variable named *x)*

SQL Syntax: Simple Stored Function

```
CREATE FUNCTION
    reservationsMadeAtHotel(hn INT)
    RETURNS INT
        BEGIN
            DECLARE x INT;
            SELECT COUNT(*) INTO x FROM
          booking WHERE hotelNumber=hn;
            RETURN(x);
        END;
```

```
mysql> select * from booking//
+-------------+-------------+------------+------------+------------+
| hotelNumber | guestNumber | dateFrom   | dateTo     | roomNumber |
+-------------+-------------+------------+------------+------------+
|           1 |           1 | 2012-02-15 | 2012-02-18 |          2 |
|           1 |           5 | 2012-03-12 | 2012-03-15 |          2 |
|           3 |           2 | 2012-02-20 | 2012-02-23 |          1 |
|           3 |           3 | 2012-02-22 | 2012-02-25 |          2 |
|           4 |           1 | 2012-02-20 | 2012-02-22 |          3 |
|           4 |           2 | 2012-04-15 | 2012-04-15 |          1 |
|           4 |           5 | 2012-05-24 | 2012-05-30 |          3 |
+-------------+-------------+------------+------------+------------+
7 rows in set (0.00 sec)
```

```
mysql> select reservationsMadeAtHotel(1);
    -> //
+-----------------------------+
| reservationsMadeAtHotel(1)  |
+-----------------------------+
|                           2 |
+-----------------------------+
1 row in set (0.00 sec)
```

SQL Syntax: Stored Procedure

```
CREATE PROCEDURE
        reservationsMadeAtHotel
                (IN hn INT, OUT x INT)
        BEGIN
        SELECT COUNT(*) INTO x FROM
            booking WHERE hotelNumber=hn;
        END;

    CALL reservationsMadeAtHotel(1,@theCount);
    SELECT @theCount;
```

New ideas:

- IN and OUT parameters
- Variables at prompt

SQL Syntax: Stored Procedure

```
mysql> CREATE PROCEDURE
    -> reservationsMadeAtHotel(IN hn INT, OUT x INT)
    -> BEGIN
    -> SELECT COUNT(*) INTO x FROM booking WHERE hotelNumber=hn;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL reservationsMadeAtHotel(1,@theCount);
    -> //
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @theCount;
    -> //
+------------+
| @theCount  |
+------------+
|          2 |
+------------+
1 row in set (0.00 sec)
```

SQL Syntax: Stored Procedure

```
CREATE PROCEDURE      printHotelStatus()
  BEGIN
     DECLARE mh INT;
     DECLARE h INT;
     DECLARE x INT;
     DECLARE toprint TEXT;
     SELECT MAX(hotelNumber) into mh FROM hotel;
     SET h = 1;
     WHILE (h <= mh) DO
          SELECT COUNT(*) INTO x FROM  booking WHERE hotelNumber=h;
          IF x = 0 THEN SET toprint = CONCAT("HOTEL ", h, " HAS NO RESERVATIONS");
          ELSE SET toprint = CONCAT("HOTEL ", h, " HAS RESERVATIONS");
          END IF;
          SELECT toprint;
          SET h = h + 1;
     END WHILE;
   END;
```

New ideas:

- SET, = for assignment
- WHILE Loop
- IF Statement