Advanced Database Management System

Lecture 10 – Sections 8.7, 8.8, 24.1
Views, Semantic Constraints and Triggers

REVIEW: SQL Queries

- Count the number of men and women
  in each department.
  Report: (department name, # of women, # of men)

EMPLOYEE(…, SSN, …, SEX, …, DNO)
domain(SEX) = { ‘M’, ‘F’ }

DEPARTMENT(DNAME, DNUMBER, …)

1: Count women/men per department

R_f = select dno as dno1, count(*) as cnt_f
from employee where sex='F'
group by dno

dno1	cnt_f
4	2
5	1

dno2	cnt_m
1	1
4	1
5	3

R_m = select dno as dno2, count(*) as cnt_m
from employee where sex='M'
group by dno

2a: combine the results

dno1	cnt_f
4	2
5	1

⋈ dno1=dno2

dno2	cnt_m
1	1
4	1
5	3

Result:

dno1	cnt_f	dno2	cnt_m
4	2	4	1
5	1	5	3

close … but we lost department 1!

2b: combine the results

dno1	cnt_f
4	2
5	1

$\bowtie_{dno1=dno2}$

dno2	cnt_m
1	1
4	1
5	3

Result:

dno1	cnt_f	dno2	cnt_m
4	2	4	1
5	1	5	3
NULL	NULL	1	1

full outer join gives everything we need …
unfortunately, MySql doesn't support full outer join.

2c: alternate approach

Since we'll need department names, left join the initial results with DEPARTMENT,
then join those results.

$$(\text{Department} \rightthreetimes_{\text{dnumber=dno1}} R_f) \bowtie_{\text{dnumber=dnumber}} (\text{Department} \rightthreetimes_{\text{dnumber=dno1}} R_m)$$

The left joins force all department numbers to be represented in the subqueries, and they will all find a match in the third join.

3: subqueries

```
(select department.dnumber as department1, cnt_f as women
from
department left join
(select dno as dno1, count(*) as cnt_f
from employee where sex='F'
group by dno) as R1 on dno1=department.dnumber) as R3
```

Result will look like (department1, cnt_f) and will include all departments. Departments with no women will have NULL in second column.

Subquery for counting men will be similar.

4: solution in SQL

```
select department1 as department, women, men
from

(select department.dnumber as department1, cnt_f as women
from
department left join
(select dno as dno1, count(*) as cnt_f
from employee where sex='F'
group by dno) as R1 on dno1=department.dnumber) as R3,

(select department.dnumber as department2, cnt_m as men
from
department left join
(select dno as dno1, count(*) as cnt_m
from employee where sex='M'
group by dno) as R2 on dno1=department.dnumber) as R4

where department1=department2;
```

Views

Views in SQL

- A ***view*** is a ***virtual table*** that is derived from other tables
  - Virtual → computed, not stored
- Allows full query operations
  - Behaves as a table when queried
- Allows for limited update operations
  - Updates must be mapped down to tables that contribute to the view

Specification of Views

- SQL command: **`CREATE VIEW`**
  - a table (view) name
  - a possible list of attribute names
    (for example, when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations)
  - a query to specify the table contents

SQL Views: An Example

- Specify a different WORKS_ON table

```
CREATE VIEW WORKS_ON_BYNAME AS
SELECT FNAME, LNAME, PNAME, HOURS
      FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER
      GROUP BY PNAME;
```

Using a Virtual Table

- We can specify SQL queries
  on a newly created table (view):

```
SELECT FNAME, LNAME
      FROM WORKS_ON_BYNAME
      WHERE PNAME='Seena';
```

- When no longer needed, a view can be dropped:

```
DROP WORKS_ON_NEW;
```

Efficient View Implementation

- Query modification:
  - Represent the view query in terms of a query on the underlying base tables
- Disadvantage:
  - Inefficient for views defined via complex queries
    - Especially if additional queries are to be applied to the view within a short time period

Efficient View Implementation

- View materialization:
  - Involves physically creating and keeping a temporary table
- Assumption:
  - Other queries on the view will follow
- Concerns:
  - Maintaining correspondence between the base table and the view when the base table is updated
- Strategy:
  - Incremental update

Updating Views

- Update on a view of a single table without aggregate operations:
  - Update may map to an update on the underlying base table
- Views involving joins:
  - An update *may* map to an update on the underlying base relations
    - (This is not always possible)

Non-updatable Views

- Views defined using GROUP BY and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable
- View on tables that have CHECK constraints: the CHECK must also be added to the definition of a view if the view is to be updated
  - To allow check for updatability and to plan for an execution strategy

Constraints

Previous Constraints

```
CREATE TABLE Teaching (
   ProfId    INTEGER,
   CrsCode CHAR (6),
   Semester CHAR (6),
   PRIMARY KEY (CrsCode, Semester),

   FOREIGN KEY (ProfId) REFERENCES Professor (Id)
      ON DELETE NO ACTION
      ON UPDATE CASCADE,

    FOREIGN KEY (CrsCode) REFERENCES Course (CrsCode)
      ON DELETE SET NULL
      ON UPDATE CASCADE )
```

Table Semantic Constraints

- Used for application dependent conditions
- *Example*: limit attribute values

```
CREATE TABLE Transcript (
   StudId  INTEGER,
   CrsCode  CHAR(6),
   Semester  CHAR(6),
   Grade  CHAR(1),
   CHECK (Grade IN ('A', 'B', 'C', 'D', 'F')),
   CHECK (StudId > 0 AND StudId < 1000000000) )
```

- Each row in table must satisfy condition

User-Defined Domains

- Possible attribute values can be specified
  - Using a CHECK constraint, or
  - Creating a new domain
- Domain can be used in several declarations
- Domain is a schema element

```
CREATE DOMAIN Grades CHAR (1)
   CHECK  (VALUE IN ('A', 'B', 'C', 'D', 'F'))
CREATE TABLE Transcript (
    …..,
    Grade: Grades,
    …  )
```

Table Constraint Example

- Ensure that managers are paid more than their employees.

```
CREATE TABLE Employee (
   Id   INTEGER,
   Name   CHAR(20),
   Salary   INTEGER,
   MngrSalary   INTEGER,
   CHECK ( MngrSalary > Salary) )
```

Constraints – Problems

- **Problem 1:**

An empty table always satisfies all CHECK constraints (an idiosyncrasy of the SQL standard)

```
CREATE TABLE Employee (
   Id  INTEGER,
   Name  CHAR(20),
   Salary  INTEGER,
   MngrSalary  INTEGER,
   CHECK ( 0 < (SELECT COUNT (*) FROM Employee)) )
```

- If Employee is empty, there are no rows on which to evaluate the CHECK condition.

Constraints – Problems

- **Problem 2:**
  Inter-relational constraints should be symmetric

```
CREATE TABLE Employee (
   Id  INTEGER,
   Name  CHAR(20),
   Salary  INTEGER,
   MngrSalary  INTEGER,
    CHECK ((SELECT  COUNT (*) FROM Manager) <
                  (SELECT  COUNT (*) FROM Employee)) )
```

  - Why should constraint be in Employee, rather than Manager?
  - What if Employee is empty?

Assertions

- Assertions are schema elements
- Symmetrically specifies
  an inter-relational constraint
- Applies to entire database
  (not just the individual rows of a single table)
  - hence it works even if Employee is empty

```
CREATE ASSERTION DontFireEveryone
  CHECK (0 < SELECT  COUNT (*)  FROM Employee)
```

Designing Assertions

- Specify a query that violates the condition; include that inside a `NOT EXISTS` clause

- Query result must be empty
  - if the query result is not empty, the assertion has been violated

```
CREATE ASSERTION KeepEmployeeSalariesDown
    CHECK (NOT EXISTS(
                SELECT * FROM Employee E
                WHERE E.Salary > E.MngrSalary))
```

Assertion Example

- The salary of an employee must not be greater than the salary of the manager of the department that the employee works for.

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS
            (SELECT *
             FROM EMPLOYEE E, EMPLOYEE M,
                  DEPARTMENT D
             WHERE E.SALARY > M.SALARY AND
                   E.DNO=D.NUMBER AND
                   D.MGRSSN=M.SSN))
```

Assertion Example

```
CREATE ASSERTION  NoEmptyCourses
   CHECK  (NOT  EXISTS (
                  SELECT *  FROM Teaching T
                  WHERE NOT  EXISTS (
                        SELECT * FROM Transcript  R
                        WHERE T.CrsCode = R.CrsCode
                        AND T.Semester = R.Semester)
               ) )
```

not the double negative logic:
It is ***not true*** that there are courses taught that ***do not*** have students.

Triggers

Triggers

- Triggers are active statements
  that specify responses to specific conditions.

- A trigger is a schema element

```
CREATE TRIGGER CrsChange
    AFTER UPDATE OF CrsCode, Semester ON Transcript
    WHEN  (Grade IS NOT NULL)
        ROLLBACK
```

Trigger Overview

- Element of the database schema
- General form:

  ON <*event*> IF <*condition*> THEN <*action*>

  - *Event*- request to execute database operation
  - *Condition* - predicate evaluated on databaase state
  - *Action* – execution of procedure that might involve database updates

- Example:

  ON updating maximum course enrollment

  IF number registered > new max enrollment limit
  THEN deregister students using LIFO policy

Possible Trigger Semantics

- **Activation** - Occurrence of the *event*

- **Consideration** - The point, after activation, when the *condition* is evaluated
  - **Immediate:** evaluate condition as soon as the event occurs
  - **Deferred:** wait to evaluate the condition at the end of the transaction

  - The *condition* may refer to the database state both before and after the triggering event

Possible Trigger Semantics

- **Execution** – point at which the *action* occurs
  - With deferred consideration, execution is also deferred
  - With immediate consideration, execution can occur immediately after consideration or it can be deferred
    - If execution is immediate, execution can occur before, after, or instead of triggering event.
    - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Possible Trigger Semantics

- **Granularity**
  - *Row-level granularity*: change of a single row is an event (a single UPDATE statement might result in multiple events)
  - *Statement-level granularity*: events are statements (a single UPDATE statement that changes multiple rows is a single event).

Possible Trigger Semantics

- **Multiple Triggers**
  - How should multiple triggers activated by a single event be handled?
    - Evaluate one condition at a time and if true immediately execute action or
    - Evaluate all conditions, then execute actions
  - The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.

Triggers in SQL

- **Events:**
  INSERT, DELETE, or UPDATE statements or changes to individual rows caused by these statements

- **Condition:**
  Anything that is allowed in a WHERE clause

- **Action:**
  An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM)
  (which may contain embedded SQL statements)

Triggers in SQL

- **Consideration:** *Immediate*
  - Condition can refer to the state of the affected row or table before *and* after the event occurs
- **Execution:** *Immediate* – can be before or after the execution of the triggering event
  - Action of a before trigger cannot modify the database
- **Granularity:** Both *row-level* and *statement-level*

Trigger Syntax

CREATE TRIGGER *trigger-name*
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [OF *column-name-list*] }
ON *table-name*
[REFERENCING { OLD AS *old-tuple-name* |
NEW AS *new-tuple-name* |
OLD TABLE AS *old-table-name* |
NEW TABLE AS *new-table-name* }]
[FOR EACH { ROW | STATEMENT }]
[WHEN (*precondition*)]
statement-list

Before Trigger (row granularity)

Check that enrollment ≤ limit

```
CREATE TRIGGER  Max_EnrollCheck
  BEFORE INSERT ON Transcript
        REFERENCING NEW AS N   --row to be added
   FOR EACH ROW
   WHEN
   ((SELECT  COUNT (T.StudId) FROM Transcript T
     WHERE  T.CrsCode = N.CrsCode
                AND T.Semester = N.Semester)
   >=
   (SELECT C.MaxEnroll FROM Course C
     WHERE C.CrsCode = N.CrsCode ))
  ABORT TRANSACTION
```

After Trigger (row granularity)

```
CREATE TRIGGER LimitSalaryRaise
  AFTER UPDATE OF Salary ON Employee
  REFERENCING OLD AS O
                      NEW AS N
   FOR EACH ROW
   WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
      UPDATE Employee
      SET Salary = 1.05 *  O.Salary
      WHERE Id = O.Id
```

No salary raises greater than 5%

Note: The action itself is a triggering event (but in this case a chain reaction is not possible)

After Trigger (stmt granularity)

```
CREATE TRIGGER RecordNewAverage
  AFTER UPDATE OF Salary ON Employee
  FOR EACH STATEMENT
     INSERT INTO Log
     VALUES  (CURRENT_DATE,
                  SELECT AVG (Salary)
                  FROM  Employee)
```

Keep track of salary averages in the log

Trigger Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
     SALARY, SUPERVISOR_SSN ON EMPLOYEE
     FOR EACH ROW
     WHEN
     (NEW.SALARY > (SELECT SALARY FROM EMPLOYEE
                    WHERE SSN=NEW.SUPERVISOR_SSN))
     INFORM_SUPERVISOR(NEW.SUPERVISOR_SSN,NEW.SSN);
```

INFORM_SUPERVISOR is a ***stored procedure.***