ICS 3101:Advanced Database Systems
Advanced SQL

Strathmore
UNIVERSITY

Strathmore UNIVERSITY

Learning Outcomes

Views, Stored Procedures, Functions, and Triggers

References

1) Database Systems Concepts 6th Edition-Silberchartz,Korth and Surdarshan

2) https://www.mysqltutorial.org/advanced-mysql/

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

Views in SQL

- A view is a “virtual” table that is derived from other tables
- Allows for limited update operations
  - Since the table may not physically be stored
- Allows full query operations

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v*.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

SQL Views: An Example

- Create a view for Department Managers:

```
CREATE VIEW MANAGER AS
        SELECT FNAME, LNAME, DName, Dnumber, SALARY
        FROM EMPLOYEE, DEPARTMENT
        WHERE SSN=MGRSSN AND DNO=DNUMBER;
```

- Find employees who earn more than their managers

```
SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E, MANAGER M
WHERE E.DNO=M.DNUMBER AND E.SALARY > M.SALARY;
```

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

```
DROP VIEW MANAGER;
```

Example Views

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = ’Physics’
  and *section.semester* = ’Fall’
  and *section.year* = ’2009’;

- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building*= ’Watson’;

View Implementation

- There are two ways to implement a view:
- Approach 1: Query modification
  - Modify the view query into a query on the underlying base tables
  - Example:

    SELECT * FROM Manager WHERE Salary > 100000

    becomes

    ```
    SELECT Fname, Lname, Dname, Dnumber, Salary
    FROM EMPLOYEE, DEPARTMENT
    WHERE SSN=MgrSSN AND Salary > 100000
    ```

  - Disadvantage:
    - Inefficient for views defined via complex queries

View Implementation

- Approach 2: View materialization
  - Involves physically creating and keeping a temporary table
  - Concerns:
    - Maintaining correspondence between the base table and the view when the base table is updated

- ORACLE

  CREATE MATERIALIZED VIEW or CREATE SNAPSHOT

Update Views

- Update on a view can be implemented by mapping it to an update on the underlying base table

```
UPDATE MANAGER
SET Salary = 1.1*Salary
WHERE Dname = ‘Research’;
```

  - Becomes:

```
UPDATE EMPLOYEE
SET Salary = 1.1*Salary
WHERE SSN in (SELECT MgrSSN
                FROM DEPARTMENT
                WHERE DName = ‘Research’);
```

- Updating views involving joins are not always possible
  - Views defined using groups and aggregate functions are not updateable
- For mySQL, the keyword **“WITH CHECK OPTION”** must be added to the view definition if the view is to be updated

Managing Views

- Create views with a WITH CHECK OPTION – ensure the consistency of views using the WITH CHECK OPTION clause.
- LOCAL & CASCADED and WITH CHECK OPTION – specify the scope of the check with LOCAL and CASCADED options.

Strathmore UNIVERSITY

Uses for SQL Views

- Security: hide columns and rows
- Display results of computations
- Hide complicated SQL syntax
- Provide a level of isolation between actual data and the user's view of data
  - three-tier architecture
- Assign different processing permissions to different views on same table
- Assign different triggers to different views on same table

Stored Procedures

- A stored procedure is a program that is stored within the database and is compiled when used
  - In Oracle, it can be written in PL/SQL or Java
  - In SQL Server, it can be written in TRANSACT-SQL
- Stored procedures can receive input parameters and they can return results
- Stored procedures can be called from:
  - Programs written in standard languages, e.g., Java, C#
  - Scripting languages, e.g., JavaScript, VBScript
  - SQL command prompt, e.g., SQL*Plus, Query Analyzer

Stored Procedures in MySQL

- A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program

- Stored procedures are declared using the following syntax:

```
Create Procedure <proc-name>
        (param_spec1, param_spec2, …, param_specn )
begin
    -- execution code
end;
```

where each param_spec is of the form:

[in | out | inout] <param_name> <param_type>

- in mode: allows you to pass values into the procedure,
- out mode: allows you to pass value back from procedure to the calling program

Advantages of using Stored Procedures

- Reduce network traffic

Stored procedures help reduce the network traffic between applications and MySQL Server. Because instead of sending multiple lengthy SQL statements, applications have to send only the name and parameters of stored procedures.

- Centralize business logic in the database

You can use the stored procedures to implement business logic that is reusable by multiple applications. The stored procedures help reduce the efforts of duplicating the same logic in many applications and make your database more consistent.

Advantages of Stored procedure

- Make database more secure

The database administrator can grant appropriate privileges to applications that only access specific stored procedures without giving any privileges on the underlying tables.

Disadvantages of using Stored Procedures

- Resource usages

If you use many stored procedures, the memory usage of every connection will increase substantially.

Besides, overusing a large number of logical operations in the stored procedures will increase the CPU usage because the MySQL is not well-designed for logical operations.

- Troubleshooting

It’s difficult to debug stored procedures. Unfortunately, MySQL does not provide any facilities to debug stored procedures like other enterprise database products such as Oracle and SQL Server.

Example

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
```

```
mysql> select * from department;
+---------+------------+
| dnumber | dname      |
+---------+------------+
|       1 | Payroll    |
|       2 | TechSupport |
|       3 | Research   |
+---------+------------+
```

- Suppose we want to keep track of the total salaries of employees working for each department

```
mysql> create table deptsal as
    -> select dnumber, 0 as totalsalary from department;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
```

We need to write a procedure to update the salaries in the deptsal table

Example

```
mysql> delimiter //
```

Step 1: Change the delimiter (i.e., terminating character) of SQL statement from semicolon (;) to something else (e.g., //)

So that you can distinguish between the semicolon of the SQL statements in the procedure and the terminating character of the procedure definition

Step 2:

1.Define a procedure called updateSalary which takes as input a department number.

2.The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.

3.Terminate the procedure definition using the delimiter you had defined in step 1 (//)

Example

```
mysql> delimiter //
mysql> create procedure updateSalary (IN param1 int)
    -> begin
    ->     update deptsal
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
    ->     where dnumber = param1;
    -> end; //
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;
```

Step 3: Change the delimiter back to semicolon (;)

Example

```
mysql> call updateSalary(1);
Query OK, 0 rows affected (0.00 sec)

mysql> call updateSalary(2);
Query OK, 1 row affected (0.00 sec)

mysql> call updateSalary(3);
Query OK, 1 row affected (0.00 sec)
```

Step 4: Call the procedure to update the totalsalary for each department

Example

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Step 5: Show the updated total salary in the deptsal table

Stored Procedures in MySQL

- Use show procedure status to display the list of stored procedures you have created

```
mysql> show procedure status;
+------+---------------+-----------+---------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
| Db   | Name          | Type      | Definer | Modified            | Created             | Security_
type | Comment | character_set_client | collation_connection | Database Collation |
+------+---------------+-----------+---------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
| ptan | updateSalary0 | PROCEDURE | ptan@%  | 2010-03-16 12:21:55 | 2010-03-16 12:21:55 | DEFINER
     |         | latin1               | latin1_swedish_ci    | latin1_swedish_ci  |
+------+---------------+-----------+---------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
1 row in set (0.02 sec)
```

- Use drop procedure to remove a stored procedure

```
mysql> drop procedure updateSalary;
Query OK, 0 rows affected (0.00 sec)
```

Strathmore UNIVERSITY

Stored Procedures in MySQL

- You can declare variables in stored procedures
- You can use flow control statements (conditional IF-THEN-ELSE or loops such as WHILE and REPEAT)
- MySQL also supports cursors in stored procedures.
  - A cursor is used to iterate through a set of rows returned by a query so that we can process each individual row.
- To learn more about stored procedures, go to:
  http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx

Example using Cursors

- The previous procedure updates one row in deptsal table based on input parameter
- Suppose we want to update all the rows in deptsal simultaneously
  - First, let's reset the totalsalary in deptsal to zero

```
mysql> update deptsal set totalsalary = 0;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 3  Changed: 0  Warnings: 0

mysql> select * from deptsal;
+----------+---------------+
| dnumber  | totalsalary   |
+----------+---------------+
|        1 |             0 |
|        2 |             0 |
|        3 |             0 |
+----------+---------------+
3 rows in set (0.00 sec)
```

Example using Cursors

```
mysql> delimiter $$
mysql> drop procedure if exists updateSalary$$
Query OK, 0 rows affected (0.00 sec)

mysql> create procedure updateSalary()
    -> begin
    ->          declare done int default 0;
    ->          declare current_dnum int;
    ->          declare dnumcur cursor for select dnumber from deptsal;
    ->          declare continue handler for not found set done = 1;
    ->
    ->          open dnumcur;
    ->
    ->          repeat
    ->                fetch dnumcur into current_dnum;
    ->                update deptsal
    ->                set totalsalary = (select sum(salary) from employee
    ->                                   where dno = current_dnum)
    ->                where dnumber = current_dnum;
    ->          until done
    ->          end repeat;
    ->
    ->          close dnumcur;
    -> end$$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Drop the old procedure

Use cursor to iterate the rows

Example using Cursors

- Call procedure

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
3 rows in set (0.01 sec)

mysql> call updateSalary;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Another Example

- Create a procedure to give a raise to all employees

```
mysql> select * from emp;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL |  90000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)
```

Another Example

```
mysql> delimiter |
mysql> create procedure giveRaise (in amount double)
    -> begin
    ->         declare done int default 0;
    ->         declare eid int;
    ->         declare sal int;
    ->         declare emprec cursor for select id, salary from employee;
    ->         declare continue handler for not found set done = 1;
    ->
    ->         open emprec;
    ->         repeat
    ->                 fetch emprec into eid, sal;
    ->                 update employee
    ->                 set salary = sal + round(sal * amount)
    ->                 where id = eid;
    ->         until done
    ->         end repeat;
    -> end |
Query OK, 0 rows affected (0.00 sec)
```

Another Example

```
mysql> delimiter ;
mysql> call giveRaise(0.1);
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+----+-------+---------+--------+------------+------+
| id | name  | superid | salary | bdate      | dno  |
+----+-------+---------+--------+------------+------+
|  1 | john  |       3 | 110000 | 1960-01-01 |    1 |
|  2 | mary  |       3 |  55000 | 1964-12-01 |    3 |
|  3 | bob   |    NULL |  88000 | 1974-02-07 |    3 |
|  4 | tom   |       1 |  55000 | 1978-01-17 |    2 |
|  5 | bill  |    NULL |   NULL | 1985-01-20 |    1 |
+----+-------+---------+--------+------------+------+
5 rows in set (0.00 sec)
```

Functions

- Functions are declared using the following syntax:

function <function-name> (param_spec$_1$, …, param_spec$_k$)
    returns <return_type>
    [not] deterministic allow optimization if same output
                for the same input (use RAND not deterministic)
Begin
  -- execution code
end;

where param_spec is:
    [in | out | in out] <param_name> <param_type>

– You need ADMIN privilege to create functions on mysql-user server

SQL Functions Example

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
        returns integer
         begin
         declare d_count  integer;
               select count (* ) into d_count
               from instructor
               where instructor.dept_name = dept_name
         return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL functions (Cont.)

Strathmore UNIVERSITY

- Compound statement: **begin … end**
  - May contain multiple SQL statements between **begin** and **end**.
- **returns** -- indicates the variable-type that is returned (e.g., integer)
- **return** **--** specifies the values that are to be returned as result of invoking the function
- SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters.

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
        returns table  (
                ID varchar(5),
                name varchar(20),
                dept_name varchar(20),
                salary numeric(8,2))
        return table
                (select ID, name, dept_name, salary
                 from instructor
                 where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table (instructor_of ( 'Music' ))
```

Example of Functions

```
mysql> select * from employee;
+----+------+----------+--------+-------------+------+
| id | name | superid  | salary | bdate       | dno  |
+----+------+----------+--------+-------------+------+
|  1 | john |        3 | 100000 | 1960-01-01  |    1 |
|  2 | mary |        3 |  50000 | 1964-12-01  |    3 |
|  3 | bob  |     NULL |  80000 | 1974-02-07  |    3 |
|  4 | tom  |        1 |  50000 | 1970-01-17  |    2 |
|  5 | bill |     NULL |   NULL | 1985-01-20  |    1 |
+----+------+----------+--------+-------------+------+
5 rows in set (0.00 sec)

mysql> deliniter |
mysql> create function giveRaise (oldval double, amount double
    -> returns double
    -> deterministic
    -> begin
    ->       declare newval double;
    ->       set newval = oldval * (1 + amount);
    ->       return newval;
    -> end |
Query OK, 0 rows affected (0.00 sec)

mysql> deliniter ;
```

Example of Functions

```
mysql> select name, salary, giveRaise(salary, 0.1) as newsal
    -> from employee;
+------+--------+--------+
| name | salary | newsal |
+------+--------+--------+
| john | 100000 | 110000 |
| mary |  50000 |  55000 |
| bob  |  80000 |  88000 |
| tom  |  50000 |  55000 |
| bill |   NULL |   NULL |
+------+--------+--------+
5 rows in set (0.00 sec)
```

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggers

- Trigger: stored program that is executed by the DBMS whenever a specified event occurs
- Associated with a table or view
- Three trigger types: BEFORE, INSTEAD OF, and AFTER
- Each type can be declared for INSERT, UPDATE, and/or DELETE
  - Resulting in a total of nine trigger types

SQL Triggers

- To monitor a database and take a corrective action when a condition occurs

  Examples:

  Charge $10 overdraft fee if the balance of an account after a withdrawal transaction is less than $500

  Limit the salary increase of an employee to no more than 5% raise

```
CREATE TRIGGER trigger-name
    trigger-time trigger-event
    ON table-name
    FOR EACH ROW
        trigger-action;
```

  - trigger-time ∈ {BEFORE, AFTER}
  - trigger-event ∈ {INSERT,DELETE,UPDATE}

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as : ** for deletes and updates
  - **referencing new row as : ** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ‘ ‘)
begin atomic
        set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

Strathmore UNIVERSITY

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred= tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id= nrow.course_id*)
  where *student.id = nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

SQL Triggers: An Example

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
5 rows in set (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

- We want to create a trigger to update the total salary of a department when a new employee is hired

SQL Triggers: An Example

- Create a trigger to update the total salary of a department when a new employee is hired:

```
mysql> delimiter !
mysql> create trigger update_salary
    -> after insert on employee
    -> for each row
    -> begin
    ->       if new.dno is not null then
    ->           update deptsal
    ->           set totalsalary = totalsalary + new.salary
    ->           where dnumber = new.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.06 sec)

mysql> delimiter ;
```

- The keyword “new” refers to the new row inserted

SQL Triggers: An Example

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> insert into employee values (6,'lucy',null,90000,'1981-01-01',1);
Query OK, 1 row affected (0.08 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> insert into employee values (7,'george',null,45000,'1971-11-11',null);
Query OK, 1 row affected (0.02 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> drop trigger update_salary;
Query OK, 0 rows affected (0.00 sec)
```

totalsalary increases by 90K

totalsalary did not change

SQL Triggers: An Example

- A trigger to update the total salary of a department when an employee tuple is modified:

```
mysql> delimiter !
mysql> create trigger update_salary2
    -> after update on employee
    -> for each row
    -> begin
    ->       if old.dno is not null then
    ->          update deptsal
    ->          set totalsalary = totalsalary - old.salary
    ->          where dnumber = old.dno;
    ->       end if;
    ->       if new.dno is not null then
    ->          update deptsal
    ->          set totalsalary = totalsalary + new.salary
    ->          where dnumber = new.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.06 sec)
```

SQL Triggers: An Example

```
mysql> delimiter ;
mysql> select * from employee;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL |  90000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> update employee set salary = 100000 where id = 6;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      200000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

SQL Triggers: An Example

- A trigger to update the total salary of a department when an employee tuple is deleted:

```
mysql> delimiter !
mysql> create trigger update_salary3
    -> before delete on employee
    -> for each row
    -> begin
    ->       if (old.dno is not null) then
    ->           update deptsal
    ->           set totalsalary = totalsalary - old.salary
    ->           where dnumber = old.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.08 sec)

mysql> delimiter ;
```

SQL Triggers: An Example

```
mysql> select * from employee;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL | 100000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)
```

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      200000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

```
mysql> delete from employee where id = 6;
Query OK, 1 row affected (0.02 sec)

mysql> delete from employee where id = 7;
Query OK, 1 row affected (0.03 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

SQL Triggers

- To list all the triggers you have created:

```
mysql> show triggers;
```

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

Strathmore UNIVERSITY

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation
  student_grades(ID, GPA)
  giving the grade-point average of each student
- Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc**) **as** *s_rank*
  from *student_grades*
- An extra **order by** clause is needed to get them in sorted order

  select *ID*, **rank**() **over** (**order by** *GPA* **desc**) **as** *s_rank*
  from *student_grades*
  order by *s_rank*
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - **dense_rank** does not leave gaps, so next dense rank would be 2

Ranking

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

Ranking (Cont.)

- Ranking can be done within partition of the data.
- "Find the rank of students within each department."

```
select ID, dept_name,
        rank () over (partition by dept_name order by GPA desc)
                as dept_rank
    from dept_grades
    order by dept_name, dept_rank;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
  - More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify **nulls first** or **nulls last**

```
select ID,
        rank ( ) over (order by GPA desc nulls last) as s_rank
from student_grades
```

Ranking (Cont.)

- For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.
- E.g.,

  select *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile*
  from *student_grades*;

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

```
select date, sum(value) over
    (order by date between rows 1 preceding and 1 following)
from sales
```

Windowing

- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value -10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number*, *date_time*, *value*), where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

```
select account_number, date_time,
    sum (value) over
            (partition by account_number
            order by date_time
            rows unbounded preceding)
   as balance
from transaction
order by account_number, date_time
```

Lab Exercise

Use the university database to

1. Create a procedure to give a raise of 20,000 to all Instructors
2. Create a procedure to keep track of the total salaries of Instructors working for each department
3. Write a function to return all instructors in a given department
4. Create a trigger to update the total salary of a department when a new Instructor is hired
5. Create a trigger to update the total salary of a department when an Instructor tuple is modified

PL/SQL

- PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.