Roles

- **create role** instructor;

- **grant** *instructor* **to Amit;**

- Privileges can be granted to roles:

  - **grant select on** *takes* **to** *instructor*;

- Roles can be granted to users, as well as to other roles

  - **create role** *teaching_assistant*

  - **grant** *teaching_assistant* **to** *instructor*;

    ‣ *Instructor* inherits all privileges of *teaching_assistant*

- Chain of roles

  - **create role** *dean*;

  - **grant** *instructor* **to** *dean*;

  - **grant** *dean* **to** Satoshi;

Authorization on Views

■ **create view** *geo_instructor* **as**
(**select** *
from *instructor*
where *dept_name* = 'Geology');

■ **grant select on** *geo_instructor* **to** *geo_staff*

■ Suppose that a *geo_staff* member issues

● **select** *
from *geo_instructor*;

■ What if

● *geo_staff* does not have permissions on *instructor?*

● creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
- Etc. read Section 4.6 for more details we have omitted here.

Figure 4.01

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.02

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	null

Figure 4.03

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	History	32	CS-315	1	Spring	2010	A
12345	Shankar	Finance	32	CS-347	1	Fall	2009	A
19991	Brandt	Music	80	HIS-351	1	Spring	2010	B
23121	Chavez	Physics	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.04

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	History	32	CS-315	1	Spring	2010	A
12345	Shankar	Finance	32	CS-347	1	Fall	2009	A
19991	Brandt	Music	80	HIS-351	1	Spring	2010	B
23121	Chavez	Physics	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.05

ID	course_id	sec_id	semester	year	grade	name	dept_name	tot_cred
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	History	32
12345	CS-347	1	Fall	2009	A	Shankar	Finance	32
19991	HIS-351	1	Spring	2010	B	Brandt	Music	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Physics	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	null	null	null	null	null	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	null	Tanaka	Biology	120

Figure 4.07

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

instructor

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

Figure 4.06

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Figure 4.03
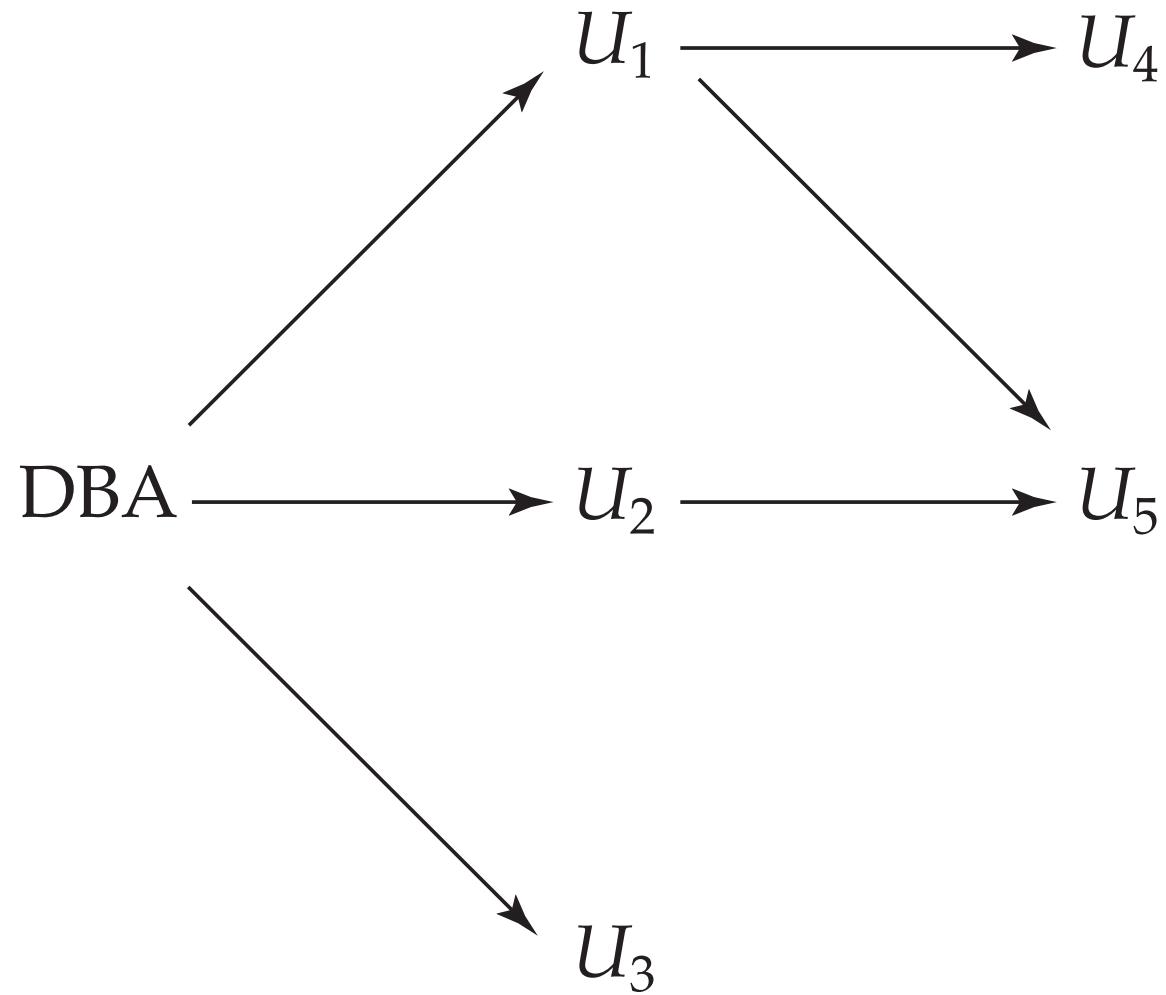
Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server

- Application makes calls to
    - Connect with the database server
    - Send SQL commands to the database server
    - Fetch tuples of result one-by-one into program variables

- Various tools:
    - JDBC (Java Database Connectivity) works with Java
    - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic. Other API's such as ADO.NET sit on top of ODBC
    - Embedded SQL

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.

- JDBC supports a variety of features for querying and updating data, and for retrieving query results.

- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.

- Model for communicating with the database:

  - Open a connection

  - Create a "statement" object

  - Execute queries using the Statement object to send queries and fetch results

  - Exception mechanism to handle errors

ODBC

- Open DataBase Connectivity (ODBC) standard

  - standard for application program to communicate with a database server.

  - application program interface (API) to

    - open a connection with a database,

    - send queries and updates,

    - get back results.

- Applications such as GUI, spreadsheets, etc. can use ODBC

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,

- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.

- The basic form of these languages follows that of the System R embedding of SQL into PL/1.

- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

          EXEC SQL <embedded SQL statement >;

  Note: this varies by language:

  - In some languages, like COBOL, the semicolon is replaced with END-EXEC

  - In Java embedding uses   # SQL { …. };

Embedded SQL (Cont.)

■ Before executing any SQL statements, the program must first connect to the database. This is done using:

    EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

 Here, *server* identifies the server to which a connection is to be established.

■ Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount*)

■ Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

      EXEC-SQL BEGIN DECLARE SECTION}

         int *credit-amount* ;

      EXEC-SQL END DECLARE SECTION;

Embedded SQL (Cont.)

■ To write an embedded SQL query, we use the

declare *c* **cursor for <SQL query>**

statement. The variable *c* is used to identify the query

■ Example:

● From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue

● Specify the query in SQL as follows:

EXEC SQL

declare *c* **cursor for**
select *ID, name*
from *student*
where tot_cred *> :credit_amount*

END_EXEC

Embedded SQL (Cont.)

- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue

- Specify the query in SQL as follows:

  EXEC SQL

      declare *c* **cursor for**
      select *ID, name*
      from *student*
      where tot_cred *> :credit_amount*

    END_EXEC

- The variable *c* (used in the cursor declaration) is used to identify the query

Embedded SQL (Cont.)

■ The open statement for our example is as follows:

<p align="center">EXEC SQL open c ;</p>

This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

■ The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

<p>EXEC SQL fetch c into :si, :sn END_EXEC</p>

Repeated calls to fetch get successive tuples in the query result

Embedded SQL (Cont.)

■ A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

■ The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** c ;

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Embedded SQL

■ Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)

■ Can update tuples fetched by cursor by declaring that the cursor is for update

    EXEC SQL

        declare *c* **cursor for**
        select *
        from *instructor*
        where *dept_name* = 'Music'
        for update

■ We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

        update *instructor*
        set *salary* = *salary* + 1000
        where current of *c*

Extensions to SQL

Functions and Procedures

- SQL:1999 supports functions and procedures

    - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).

    - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.

        ‣ Example: functions to check if polygons overlap, or to compare images for similarity.

    - Some database systems support **table-valued functions**, which can return a relation as a result.

- SQL:1999 also supports a rich set of imperative constructs, including

    - Loops, if-then-else, assignment

- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

- 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.)

- 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'))

SQL Procedures

■ The *dept_count* function could instead be written as procedure:

create procedure *dept_count_proc* (**in** *dept_name* **varchar**(20),
out *d_count* **integer)**

  begin

    select count(***) **into** *d_count*
    from *instructor*
    where *instructor.dept_name = dept_count_proc.dept_name*

  end

■ Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

    declare *d_count* **integer**;
    call *dept_count_proc*('Physics', *d_count*);

Procedures and functions can be invoked also from dynamic SQL

■ SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Language Constructs for Procedures & Functions

■ SQL supports constructs that gives it almost all the power of a general-purpose programming language.

  ● Warning: most database systems implement their own variant of the standard syntax below.

■ Compound statement: **begin … end**,

  ● May contain multiple SQL statements between **begin** and **end**.

  ● Local variables can be declared within a compound statements

■ **While** and **repeat** statements:

  ● **while** *boolean expression* **do**

      sequence of statements ;
    end while

  ● **repeat**

      sequence of statements ;
    until *boolean expression*
    end repeat

Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

  declare n **integer default** 0;
   for r **as**
        select $budget$ **from** $department$
   do
            set $n = n + r.budget$
   end for

Language Constructs (Cont.)

■ Conditional statements (**if-then-else**)
SQL:1999 also supports a **case** statement similar to C case statement

■ Example procedure: registers student after ensuring classroom capacity is not exceeded

- Returns 0 on success and -1 if capacity is exceeded

- See book (page 177) for details

■ Signaling of exception conditions, and declaring handlers for exceptions

> **declare** *out_of_classroom_seats* **condition**
> **declare exit handler for** *out_of_classroom_seats*
> **begin**
>
> …
> .. **signal** *out_of_classroom_seats*
> **end**

- The handler here is **exit** -- causes enclosing **begin..end** to be exited

- Other actions possible on exception

External Language Routines

■ SQL:1999 permits the use of functions and procedures written in other languages such as C or C++

■ Declaring external language procedures and functions

create procedure dept_count_proc(**in** *dept_name* **varchar**(20),
                 out count **integer**)
language C
external name ' /usr/avi/bin/dept_count_proc'

create function dept_count(*dept_name* **varchar**(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'

External Language Routines

- SQL:1999 allows the definition of procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries.

- Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

- Declaring external language procedures and functions

    create procedure dept_count_proc(**in** *dept_name* **varchar**(20),
                                                                out count **integer**)
    language C
    external name ' /usr/avi/bin/dept_count_proc'

    create function dept_count(*dept_name* **varchar**(20))
    returns integer
    language C
    external name '/usr/avi/bin/dept_count'

External Language Routines (Cont.)

■ Benefits of external language functions/procedures:

● more efficient for many operations, and more expressive power.

■ Drawbacks

● Code to implement function may need to be loaded into database system and executed in the database system's address space.

  ‣ risk of accidental corruption of database structures

  ‣ security risk, allowing users access to unauthorized data

● There are alternatives, which give good security at the cost of potentially worse performance.

● Direct execution in the database system's space is used when efficiency is more important than security.

Security with External Language Routines

■ To deal with security problems, we can do on of the following:

- ● Use **sandbox** techniques

  ▸ That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.

- ● Run external language functions/procedures in a separate process, with no access to the database process' memory.

  ▸ Parameters and results communicated via inter-process communication

■ Both have performance overheads

■ Many database systems support both above approaches as well as direct executing in database system address space.

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

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

- **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