Module 7: Advanced SQL

Edited by Radhika Sukapuram

Database System Concepts, 7th Ed.

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language. Why ?

- Not all queries can be expressed in SQL
- Non-declarative actions
  - printing a report
  - interacting with a user
  - sending the results of a query to a graphical user interface

  cannot be done from SQL

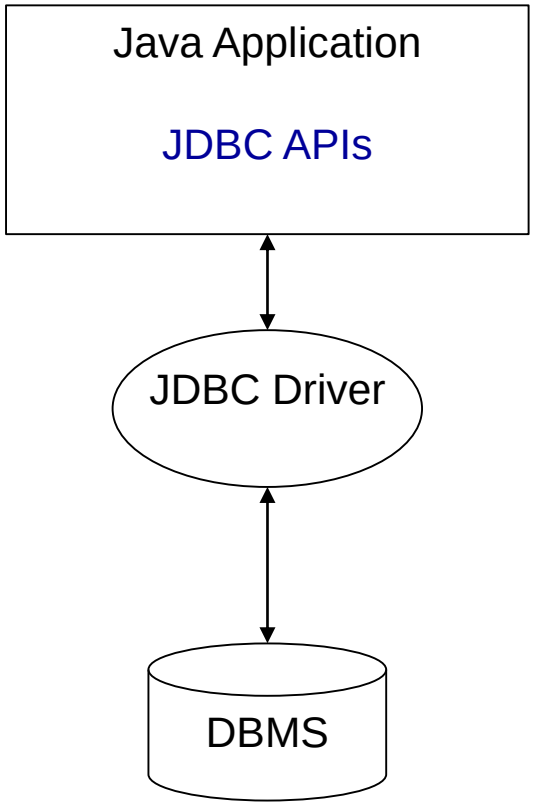
Accessing SQL from a Programming Language (Cont.)

There are two approaches to accessing SQL from a general-purpose programming language

- Dynamic SQL: A general-purpose program
  - can connect to and communicate with a database server using a collection of functions/methods
  - Construct an SQL query as a character string and submit
  - Receive results into variables a tuple at a time
  - Two standards specifying APIs
    - Java Database Connectivity
    - Open Database Connectivity
- Embedded SQL
  - SQL statements are embedded
  - translated at compile time into function calls
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

Java Database Connectivity: JDBC

JDBC



JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports
  - querying and updating data
  - retrieving query results
  - metadata retrieval
    - 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

JDBC Code

```
public static void JDBCexample(String userid, String passwd)
    {
     try (Connection conn = DriverManager.getConnection(
           "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
           Statement stmt = conn.createStatement(); //To send SQL statements
         )
     {
           /*… Do Actual Work ….     */
     }
     catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
Resources opened in “try (….)” syntax are automatically closed at the end of the try block

JDBC Code for Older Versions of Java/JDBC

```
public static void JDBCexample(String userid, String passwd)
  {
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
      Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
      Statement stmt = conn.createStatement();
        … Do Actual Work ….
      stmt.close();
      conn.close();
    }
    catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Class.forName is not required from JDBC 4 onwards. The try with resources syntax in previous slide is preferred for Java 7 onwards.

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values(' 77987' , ' Kim' , ' Physics' ,
98000)");
} catch (SQLException sqle)
{
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
                        "select dept_name, avg (salary)
                         from instructor
                         group by dept_name");
while (rset.next()) {
    System.out.println(rset.getString("dept_name") + " " +
                            rset.getFloat(2)); //Attribute Position
}
```

JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features

JDBC Code Details

- Getting result fields:
  - **rs.getString(“dept_name”) and rs.getString(1) are equivalent if dept_name is the first argument of select result.**
- Dealing with Null values

```
int a = rs.getInt(“a”);
if (rs.wasNull()) Systems.out.println(“Got null value”);
```

Prepared Statement

- PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values(?,?,?,?)");
  pStmt.setString(1, "88877");
  pStmt.setString(2, "Perry");
  pStmt.setString(3, "Finance");
  pStmt.setInt(4, 125000);
  pStmt.executeUpdate();
  pStmt.setString(1, "88878");
  pStmt.executeUpdate();
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - **stmt.executeUpdate(** "insert into instructor values(' " + ID + " ', ' " + name + " ', ' " + dept name + " ', ' "+ balance + ")" **)**;
  - What if name is "D' Souza"?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ’ " + name + "’ "
- Suppose the user, instead of entering a name, enters:
  - X’ or ’Y’ = ’Y
- then the resulting statement becomes:
  - "select * from instructor where name = ’ " + "X’ or ’Y’ = ’Y" + "’ "
  - which is:
    - select * from instructor where name = ’X’ or ’Y’ = ’Y’
  - User could have even used
    - X’; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  "select * from instructor where name = ’X\’ or \’Y\’ = \’Y’
  - **Always use prepared statements, with user inputs as parameters**

Metadata Features

- ResultSet metadata
  - An object that can be used to get information about the types and properties of the columns in a ResultSet object.
- E.g.after executing query to get a ResultSet rs:
  - **ResultSet rs = stmt.executeQuery(…);**
    ResultSetMetaData rsmd = rs.getMetaData();

    for(int i = 1; i <= rsmd.getColumnCount(); i++) {

        System.out.println(rsmd.getColumnName(i));

        System.out.println(rsmd.getColumnTypeName(i));

    }
- How is this useful?
  - Possible to execute query without knowing the schema of the result

Metadata (Cont)

- Database metadata
- **Connection conn = DriverManager.getConnection(..);**
  DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema. It is possible to set a
// schema
// The value “%” has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
while( rs.next()) {
        System.out.println(rs.getString("COLUMN_NAME"),
                                rs.getString("TYPE_NAME");
}
```

Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
//    TABLE means only regular tables (not VIEWs)
ResultSet rs = dbmd.getTables (“”, "", “%", new String[] {“TABLE”});
while( rs.next()) {
        System.out.println(rs.getString(“TABLE_NAME“));
}
```

Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();

```
// Arguments below are:  Catalog, Schema, and Table
// The value “”  for Catalog/Schema indicates current catalog/schema
//  The value null indicates all catalogs/schemas
// Retrieves a description of the given table's primary key columns.
//They are ordered by COLUMN_NAME
ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString(“KEY_SEQ”),
                                rs.getString("COLUMN_NAME");
}
```

Use of extracting metadata

- For tasks such as writing a database browser
- Making code for such tasks generic

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

Other JDBC Features

- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively -- (ResultSet)
    - get data from these objects by getBytes() -- (Blob)
  - associate an open stream with Java Blob or Clob object to update large objects (PreparedStatement)
    - blob.setBlob(int parameterIndex, InputStream inputStream).
    - parameterIndex: the position of the attribute to write into

JDBC Resources

- JDBC
  - Oracle: https://docs.oracle.com/javase/tutorial/jdbc/index.html
  - MySQL: https://dev.mysql.com/doc/connector-j/8.0/en/

ODBC

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 Microsoft spreadsheets, Microsoft Access etc. can use ODBC

Embedded SQL

Embedded SQL

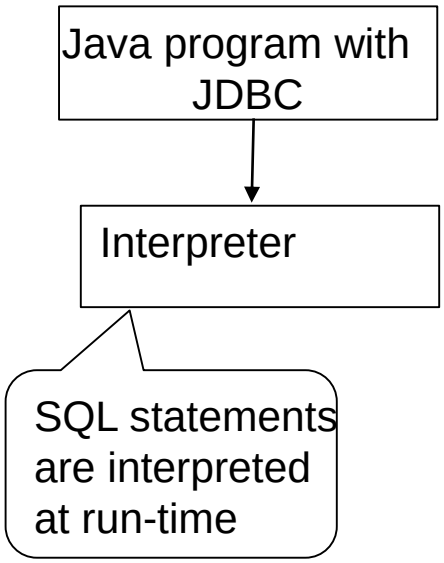
- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/I
- A language in which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

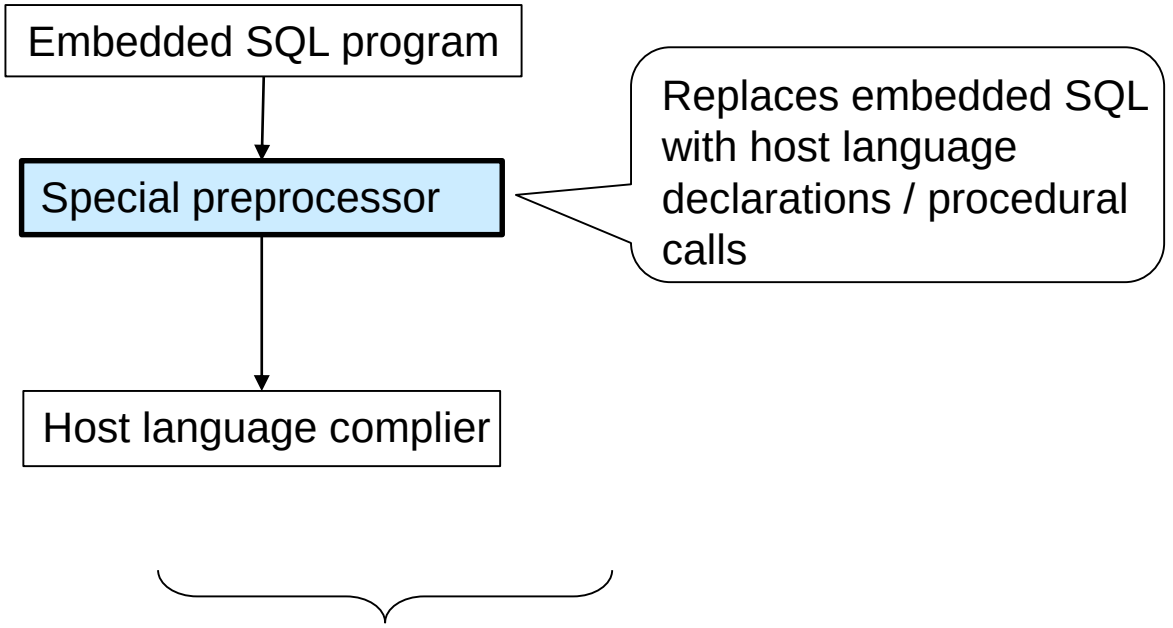
  EXEC SQL <embedded SQL statement >;

  Note: this varies by language:

  - In Java embedding uses # sql { …. }; //Called SQLJ

JDBC and Embedded 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 host language can be used in embedded SQL
  - The syntax for declaring the variables follows the usual host language syntax.

    EXEC-SQL BEGIN DECLARE SECTION;

    int *credit_amount* ;

    EXEC-SQL END DECLARE SECTION;
  - To use in embedded SQL, prefix a colon

    :*credit_amount* = 100;

Embedded SQL (Cont.)

- Incorporate the SQL error handling mechanism (SQL Communication Area)

  EXEC SQL INCLUDE SQLCA;

- Pre-defined variables are set to known values to indicate error

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 language
  - Specify the query in SQL as follows:

```
EXEC SQL
    declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount;
```

Embedded SQL (Cont.)

- The query can be executed as:

  EXEC SQL **open** *c* ;

  - results are saved in a temporary relation.
  - uses *credit_amount* of the host language when executed.
- Place one tuple in the query result on host language variables:

  EXEC SQL **fetch** *c* **into** :*si, :sn;*

  - Repeated calls to fetch get successive tuples in the query result
  - A variable SQLSTATE in SQLCA is set to '02000' to indicate no more data is available

Embedded SQL (Cont.)

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

  EXEC SQL **close** *c* ;

- The above details vary with the language.
  - Java embedding defines Java iterators to step through result tuples.

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Add 100 to the salary of every instructor in the Music department
- 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;
```

Updates Through Embedded SQL contd.

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

```
EXEC SQL open c ;
do {
        EXEC SQL fetch c into :si, :sn, :sdept, :ssalary;
        if (ssalary < 10000){
        EXEC SQL
        update instructor
         set salary = salary + 100
        where current of c;}
} while  !strcmp(SQLCA.SQLSTATE,'02000')) ;
```

Functions and Procedures

Functions and Stored Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements
  - A rule on the minimum/maximum number of courses a student can take in a semester
  - Minimum number of courses an instructor must teach in a year
  - Maximum number of leaves an employee can take

Functions and Procedures contd.

- Advantages
  - Minimizes data transfer between application and database
  - Utilizes power of DBMS server
  - Different users can reuse the stored procedure – reduces application logic
  - Needs change only in one place if business logic changes
- These can be defined either by
  - the procedural component of SQL or
  - by an external programming language - Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions
  - Oracle : PL/SQL, Microsoft SQL Server: TransactSQL , PostgreSQL(PL/pgSQL)
  - MySQL: https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

Declaring SQL Functions

- Define a function:
  - input - name of a department
  - output- 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
```

Calling a function

- The function *dept_*count can be used to
  - find the department names and budget of all departments with more than 12 instructors.

    select *dept_name, budget*
    from *department*
    where *dept_*count (*dept_name*) > 12;
- A function can be used in an expression

Table Functions

- **table functions :** functions that can return tables as results
- Example: Return all instructors in a given department

```
create function instructors_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 = instructors_of.dept_name)
```

Passed parameter

- Usage

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

- Table functions can be thought of as parameterized views

SQL Procedures

- The *dept_count* function could instead be written as a 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
```

- **in** and **out** are parameters
  - Values are assigned to **in** parameters by the caller
  - **out** parameters have values are set in the procedure in order to return results.
- 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);
```

SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the same name so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language (Persistent Storage Module of SQL standard)
  - Warning: most database systems implement their own variant of the standard syntax below.
- **declare** – to declare variables
- **set** – to assign values to variables
- Compound statement: **begin** … **end,**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within compound statements
  - **begin atomic** … **end** for atomic transactions

Language Constructs for Procedures & Functions cont.

- 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  /*Fetches one row at a time*/
        where dept_name = 'Music'
 do
                set n = n + r.budget
 end for
```

- **leave** to exit the loop //Like a break statement in C
- **iterate** starts on the next tuple, from the beginning of the loop, skipping the remaining statements

Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

  if *boolean expression*
  then *statement or compound statement*
  elseif *boolean expression*
  then *statement or compound statement*
  else *statement or compound statement*
  end if

Example 1: procedure

```
create procedure GetCustomerLevel(
    in  p_customerNumber int(11),
    out p_customerLevel  varchar(10))
begin
    declare creditlim integer;

    select creditlimit into creditlim
    from customers
    where customerNumber = p_customerNumber;

    if creditlim > 50000 then
        set p_customerLevel = 'PLATINUM';
    elseif (creditlim <= 50000 AND creditlim >= 10000) then
        set p_customerLevel = 'GOLD';
    elseif creditlim < 10000 then
        set p_customerLevel = 'SILVER';
    end if;

end;
```

Example 2: procedure

- Registers student for a course after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - *takes (ID, course id, sec id, semester, year, grade)* –courses that students take
  - *classroom (building, room number, capacity)*
  - *section (course id, sec id, semester, year, building, room number, time slot id*)

```
-- Registers a student after ensuring classroom capacity is not exceeded
-- Returns 0 on success, and -1 if capacity is exceeded.
create function registerStudent(
                    in s_id varchar(5),
                    in s_courseid varchar (8),
                    in s_secid varchar (8),
                    in s_semester varchar (6),
                    in s_year numeric (4,0),
                    out errorMsg varchar(100)
returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
        from takes
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
        from classroom natural join section
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    if (currEnrol < limit)
        begin
            insert into takes values
                (s_id, s_courseid, s_secid, s_semester, s_year, null);
            return(0);
        end
    -- Otherwise, section capacity limit already reached
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
        || ' section ' || s_secid;
    return(-1);
end;
```

Exception handling

- Declaring an exception:

declare *table_not_found* **condition for** 1051; /*my_sql_errorcode for unknown table */

declare exit handler for *table_not_found*

begin

select 'Please create table abc first';

end

- Causing an exception:

create procedure *print()*

begin

select * from *abc*; //*abc* does not exist

end

Exceptions

- Signaling of exception conditions, and declaring handlers for exceptions

  declare <*condition_name***> condition for <**condition_value>
  declare <*where_to_go*> **handler for** <*condition_name*>
  begin
  …
  end

- <condition_value> may be
  - A well know integer value, depending on the RDBMS chosen
  - SQLSTATE <five_character_string>
- <where_to_go> may be
  - exit
  - continue

Exceptions cont.

- The handler says that if the condition in condition_value arises the action to be taken is to exit (continue) the enclosing **begin end** statement.
- Can raise an exception explicitly by **signal** <*condition_name*>
- Pre-defined conditions
  - **sqlexception**
  - **sqlwarning**
  - **not found**

```
declare continue handler for not found
begin
--body of handler
end;
```

Calling functions/procedures from JDBC

- Calling functions and procedures
  - Prepare the callable statement
    - CallableStatement cStmt1 = conn.prepareCall("{? = call <function_name>(?)}");
    - CallableStatement cStmt2 = conn.prepareCall("{call <procedure_name>(?,?)}");
  - Register data types of function return values and out parameters using registerOutParameter()
  - Set input values (as before)
  - Execute callable statements
    - cStmt1.excute();
  - Retrieve results (as before)

External Language Routines

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL
  - 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 space of the database system 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 memory of the database process.
    - 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

Revision: Constraints discussed so far

- Domain constraints
- **not null**, **unique** and **check** constraints, **primary keys** - on attributes, on *single* relations
- Referential integrity constraints – **foreign keys**, **check** constraints *across* relations

Revision: Constraints discussed so far

- **assertion**s
  - A predicate that the database must always satisfy
    - Example: For each tuple in student, *tot_cred* must equal the total credits of courses passed by the student
  - Most databases do not support checks or assertions
  - Hard to implement because it is unclear when to check them
    - Example: MIN must be greater (MAX must be less) than the sum of values of an attribute
    - CREATE ASSERTION SumSalary CHECK (10000 >= ALL (SELECT SUM(salary) FROM instructor GROUP BY department)
  - Equivalent functionality can be implemented using triggers
  - Triggers tell when the action associated with them must be executed

Triggers

- A **trigger** is a statement that is
  - executed automatically
  - as a side effect of a modification to the database.
- More general and flexible compared to other integrity constraints

Need for triggers

- Used to
  - Implement integrity constraints that cannot be specified using the constraint mechanism, in flexible ways
    - Preparing an order
      - A sales clerk enters name of item and ordered quantity
      - Current price is automatically inserted from another table (prevents manual errors, maintains a complete record)
  - For alerting / starting tasks automatically when certain conditions are met
    - When an inventory level falls below minimum, place an order automatically
- Can gather statistics on database modifications
  - If a user has made enough purchases in the previous month, give a discount
  - Use a separate application to alert the user

Designing triggers

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed:
    - When must the trigger be checked ? Upon an *event* [this is always specified]
    - What *condition* must be satisfied ? [not just prevention]
  - Specify the *actions* to be taken when the trigger executes [prevent the event, undo its effects, perform any action, even across unrelated tables]
- Triggers introduced in SQL standard in SQL:1999
  - 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

Triggering Events and Actions in SQL cont.

- Triggers can be activated *before* an event, which can serve as an extra constraint (preventing invalid updates). 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;
```

  - Event ? Condition ? Action ?

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;
- Event ? Condition ? Action ?

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
    - Triggers are already executed on the master copy, no need to replicate
    - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Trigger error leading to failure of critical transactions that set off the trigger
  - Cascading execution – an insert on the same relation, with a trigger on insert

End of Module 7

Edited by Radhika Sukapuram

Database System Concepts, 7th Ed.