Lecture 6

Server and Security

(part 1)

OUTLINES

9.1 The Three-Tier Architecture

9.4 Stored Procedures

OUTLINES

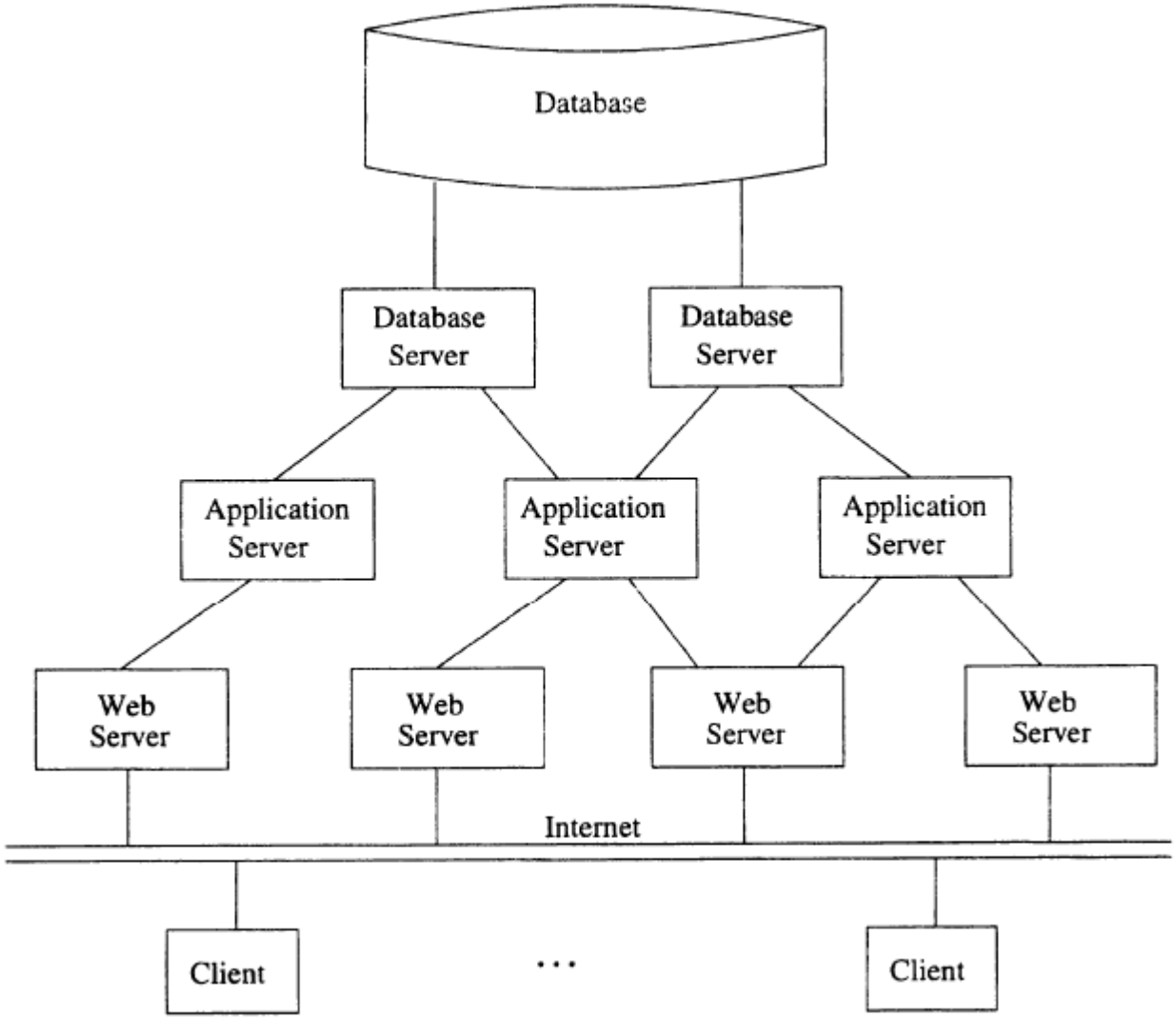
9.1 The Three-Tier Architecture

9.4 Stored Procedures

Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* --- talk to the user.
  2. *Application servers* --- execute the *business logic*.
  3. *Database servers* --- get what the app servers need from the database. (DBMS)

The Three-Tier Architecture



Example: Amazon

- Database holds the information about products, customers, etc.
- Business logic includes things like “what do I do after someone clicks ‘checkout’?”
  - Answer: Show the “how will you pay for this?” screen.

SQL Environments

- *Environment: a DBMS running at some installation*
- Schemas:collection of tables, views, assertions, triggers, and some other types of info.
  - CREATE SCHEMA <schema name> <element declarations>
  - SET SCHMEA < schema name >
- Catalogs:collection of Schemas
  - CREATE CATALOG <catalog name> ; SET CATALOG <catalog name>
- Cluster: collection of Catalogs; the maximum scope over which a query can be issued
- Connections
  - CONNECT TO <server name> AS <connection name> AUTHORIZATION <name and pwd>
  - SET CONNECTION <connection name>
  - DISCONNECT <connection name>
- Sessions:SQL operations performed while a connection is active
- Modules

SQL Environments

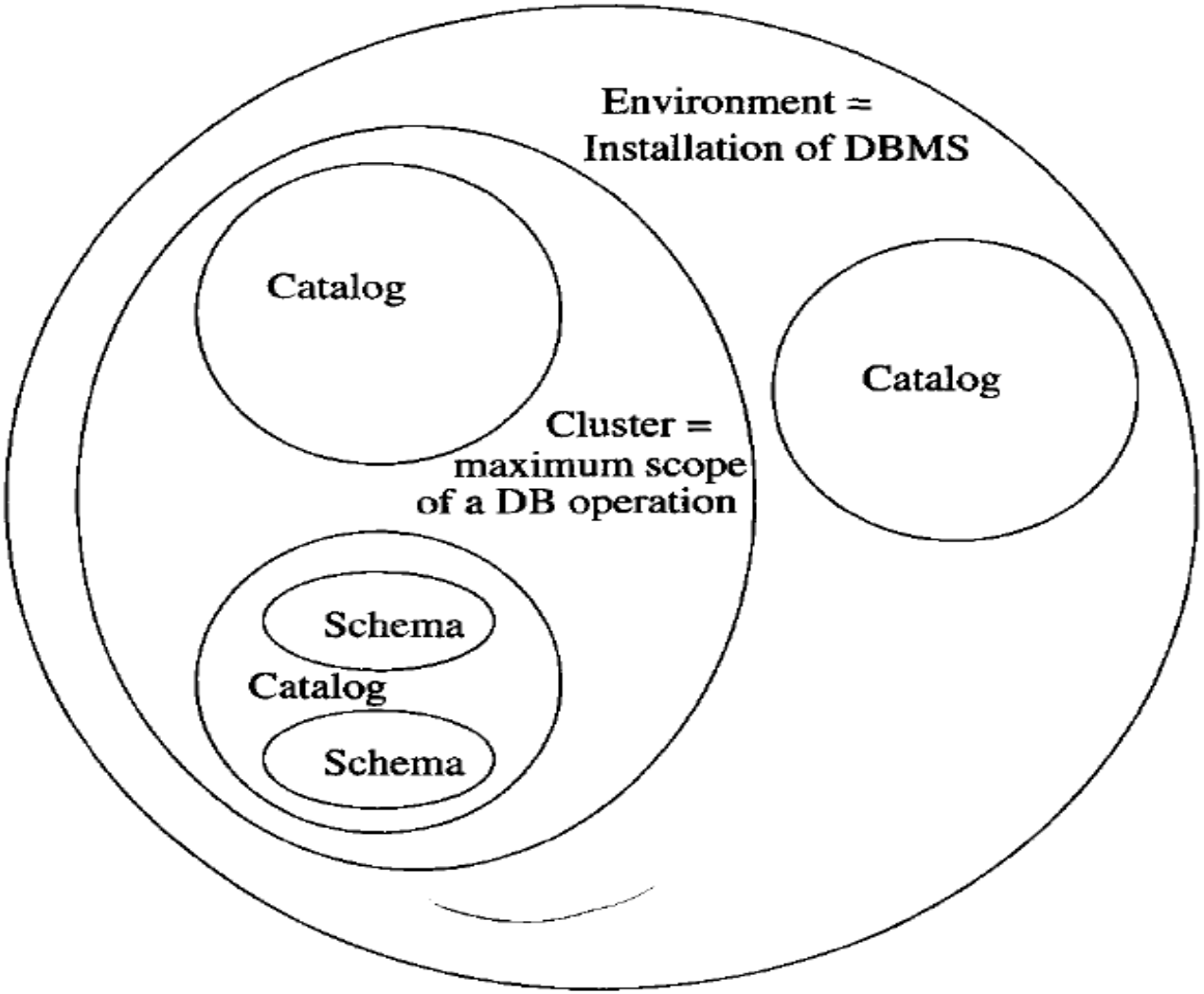


Figure 9.2: Organization of database elements within the environment

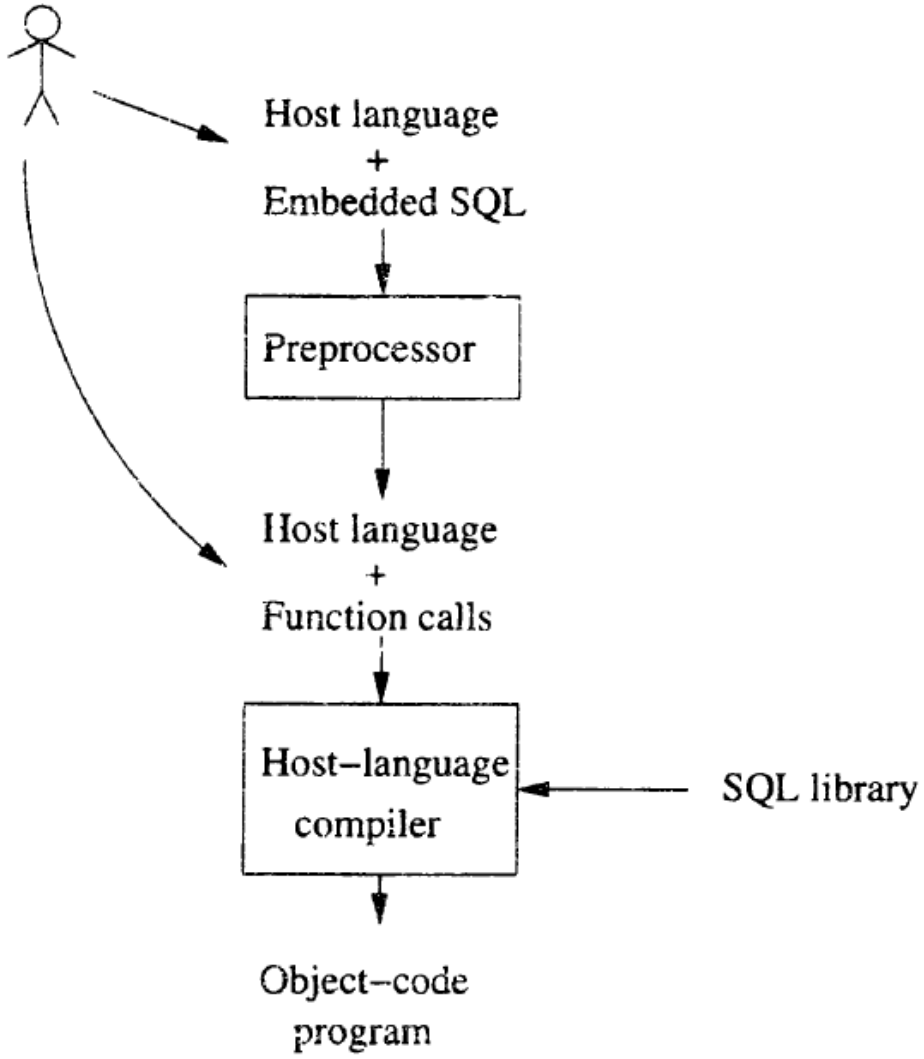
SQL in Real Programs

- We have seen only how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.
- Reality is almost always different: conventional programs interacting with SQL.

Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL).
2. SQL statements are embedded in a *host language* (e.g., C).
3. Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB).

Embedded vs. CLI



OUTLINES

9.1 The Three-Tier Architecture

9.4 Stored Procedures

Stored Procedures

- PSM, or "*persistent stored modules*," allows us to store procedures as database schema elements.
- PSM = a mixture of conventional statements (if, while, etc.) and SQL.
- Lets us do things we cannot do in SQL alone.

Basic PSM Form

CREATE PROCEDURE <name> (
    <parameter list>)
<optional local declarations>
<body>;

What are the differences between PROCEDURE and FUNCTION?

1) keywords
2) RETURNS
3) parameter list

- Function alternative:

CREATE FUNCTION <name> (
    <parameter list>) RETURNS <type>
<optional local declarations>
<body>;

Parameters in PSM

- Unlike the usual name-type pairs in languages like C and Java, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = input only, does not change value, is the default and can be omitted.
  - OUT = output only.
  - INOUT = both.
  - *Function Parameters may only be of made IN*.

Example: Stored Procedure

- Let's write a procedure that takes two arguments *oldAddr* and *newAddr*, and replaces the old address by the new everywhere it appears in MovieStar.

The Procedure

```
CREATE PROCEDURE Move (
    IN oldAddr   VARCHAR(255),
    IN newAddr   VARCHAR(255)
)
UPDATE MovieStar
SET address = newAddr
WHERE address = oldAddr;
```

Parameters are both read-only, not changed

The body --- a single update

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.
  - CALL < procedure name> (<argument list>)
- Example:

```
CALL Move('Baldwin Av','King St');
```

- Functions used in SQL expressions wherever a value of their return type is appropriate.

Kinds of PSM statements – (1)

1、RETURN <expression> sets the return value of a function.

- Unlike C, etc., RETURN *does not* terminate function execution.

2、DECLARE <name> <type> used to declare local variables.

- Declaration must precede executable statements.

3、BEGIN . . . END for groups of statements.

- Separate statements by semicolons.

Kinds of PSM Statements – (2)

4、Assignment statements:

SET <variable> = <expression>;

– Example: `SET b = 'Bud';`

5、Statement labels:

give a statement a label by prefixing a name and a colon.

IF Statements

- Simplest form:

```
IF <condition> THEN
  <statements(s)>
END IF;
```

- Add ELSE <statement(s)> if desired, as:

```
IF <condition> THEN
   <statements(s)>
ELSE
   <statements(s)>
END IF;
```

- Add additional cases by ELSEIF <statements(s)>:

```
IF <condition> THEN
   <statements(s)>
ELSEIF <condition>THEN
   <statements(s)>
ELSEIF
   …
ELSE
   <statements(s)>
END IF;
```

Example (IF)

Let us write a function to take a year ***y*** and a studio ***s***, and return a boolean that is **TRUE** if and only if studio ***s*** produced at least one *comedy* movie in year ***y*** or did not produce any movies at all in that year.

```
CREATE FUNCTION BandW(y INT, s CHAR(15)) RETURNS BOOLEAN
 IF NOT EXISTS(
      SELECT * FROM Movie WHERE year = y AND studioName = s)
 THEN RETURN TRUE;
 ELSEIF 1 <=
       (SELECT COUNT(*) FROM Movie WHERE year = y AND
        studioName = s AND  genre = 'comedy')
 THEN RETURN TRUE;
 ELSE RETURN FALSE;
 END IF;
```

Loops

- Basic form:

```
LOOP
      <statements>
END LOOP;
```

```
<loop name>: LOOP
     <statements>
END LOOP;
```

- Exit from a loop by:

```
LEAVE <loop name>;
```

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        LEAVE loop1;
        . . .
END LOOP;
```

← If this statement is executed . . .

← Control winds up here

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

```
DECLARE c CURSOR FOR <query>;
```

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

  OPEN c;

  – The query of *c* is evaluated, and *c* is set to point to the first tuple of the result.

- When finished with *c*, issue command:

  CLOSE c;

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

  FETCH FROM c INTO x_1, x_2,…,x_n ;

- The *x* 's are a list of variables, one for each component of the tuples referred to by *c*.
- *c* is moved automatically to the next tuple.

Breaking Cursor Loops – (1)

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

Breaking Cursor Loops – (2)

- Each SQL operation returns a *status*, which is a 5-digit character string.
  - For example,
    - 00000 = “Everything OK”
    - 02000 = “Failed to find a tuple.”
- In PSM, we can get the value of the status in a variable called SQLSTATE.

Breaking Cursor Loops – (3)

- We may declare a *condition*, which is a boolean variable that is true *if and only if* SQLSTATE has a particular value.

```
DECLARE <name> CONDITION FOR
SQLSTATE <value>;
```

- Example: We can declare condition `Not_Found` to represent 02000 by:

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

Breaking Cursor Loops – (4)

- The structure of a cursor loop is thus:

```
OPEN c;
cursorLoop: LOOP
  …
  FETCH c INTO … ;
  IF NotFound THEN LEAVE cursorLoop;
  END IF;
  …
END LOOP;
CLOSE c;
```

Example: Loop

```
1)  CREATE PROCEDURE  MeanVar(
2)       IN  s CHAR(15), OUT  mean  REAL, OUT  variance REAL)
5)  DECLARE  Not-Found  CONDITION  FOR  SQLSTATE  '02000';
6)  DECLARE  MovieCursor  CURSOR  FOR
          SELECT  length FROM  Movie WHERE  studioName  =  s;
7)  DECLARE  newLength  INTEGER;
8)  DECLARE  moviecount  INTEGER;
    BEGIN
9)      SET  mean  =  0.0; 10)   SET  variance =  0.0; 11)   SET  moviecount  =  0;
12)    OPEN  HovieCursor;
13)    movieLoop:  LOOP
14)         FETCH  Moviecursor  INTO  newlength;
15)         IF Not-Found  THEN  LEAVE  movieLoop  END  IF;
16)         SET  moviecount  = moviecount  +  1;
17)         SET  mean  =  mean  +  newlength;
18)         SET  variance =  variance +  newLength  *  newlength;
19)    END  LOOP;
20)    SET  mean  =  mean/movieCount;
21)    SET  variance =  variance/movieCount  - mean  * mean;
22)    CLOSE  Moviecursor;
    END  ;
```

Other Loop Forms

1、
```
FOR <loop name> AS <cursor name> CURSOR FOR
    <query>
DO
    <statements>
END FOR;
```
2、
```
WHILE <condition> DO
    <statements>
END WHILE;
```
3、
```
REPEAT
    <statements>
UNTIL <condition>
END REPEAT;
```

Example: For

```
1)  CREATE PROCEDURE  MeanVar(
2)       IN  s CHAR(15), 3)  OUT  mean  REAL, 4)  OUT  variance REAL)
5)  DECLARE  moviecount  INTEGER;
    BEGIN
6)  SET  mean  =  0.0;
7)  SET  variance  =  0.0;
8)  SET  moviecount  =  0;
9)      FOR  movieLoop  AS  Moviecursor CURSOR  FOR
                  SELECT  length FROM  Movie WHERE  studioNme  =  s;
10)     DO
11)          SET  moviecount  = moviecount  +  1;
12)          SET  mean  =  mean  +  length;
13)          SET  variance  =  variance +  length  *  length;
14)     END  FOR;
15)     SET  mean  = mean/movieCount;
16)     SET  variance  =  variance/rnovieCount  - mean  * mean;
     END  ;
```

Queries in PSM

- There are several ways that SELECT-FROM-WHERE queries are used in PSM:
  1. Subqueries can be used in condition, or in general, any place a subquery is legal in SQL.
  2. queries producing one value can be the right side in an assignment statements .
  3. Single-row SELECT . . . INTO.
  4. Cursors.

SELECT . . . INTO

- A single-row select statement is a legal statement in PSM
  - This statement has an INTO clause that specifies variables into which the components of the single returned tuple are placed

Example:

```
CREATE PROCEDURE SomeProc(IN studioName CHAR(15))

DECLARE presNetWorth INTEGER;

SELECT netWorth
INTO presNetWorth
FROM Studio, MovieExec
WHERE presC# = cert# AND Studio.name = studioName;
    ...
```

Exceptions in PSM

- A SQL system indicates error conditions in SQLSTATE.
- Declare a piece of code, *exception handler*,
  - invoked whenever one of a list of these error codes appears in SQLSTATE.

Exceptions in PSM

- DECLARE <where to go> HANDLER FOR <condition list>

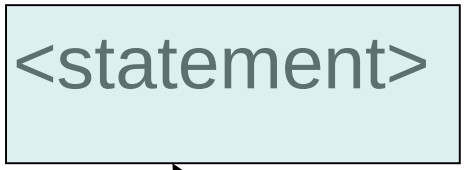
  <statement>

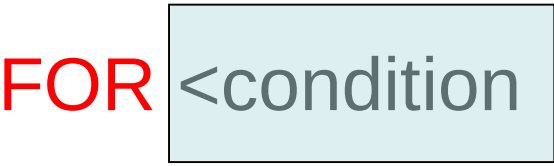
An indication of where to go after the handler has finished:

1) CONTINUE executes the statement after the one that raised the exception
2) EXIT leaves the BEGIN…END block in which the handler is declared. And the statement after this block is executed next.
3) UNDO is the same as EXIT, except the block is a transaction, and is aborted by the exception.

Exceptions in PSM

- DECLARE <where to go> HANDLER FOR <condition list>





A list of exception conditions that invoke the handler when raised.

Code to be executed when one of the associated exceptions is raised.

Example

```
CREATE FUNCTION GetYear(t VARCHAR(255))  RETURNS INTEGER
   DECLARE Not-Found CONDITION FOR SQLSTATE '02000';
   DECLARE Too-Many CONDITION FOR SQLSTATE '21000';
   BEGIN
      DECLARE EXIT HANDLER FOR Not-Found, Too-Many
          RETURN NULL;
      RETURN (SELECT year FROM Movie WHERE title = t);
   END ;
```

Example: Using PSM Function and Procedure

- CALL Procedure anywhere SQL statement can appear
  - function as part of an expression.

```
INSERT INTO StarsIn(movieTitle, movieyear, starName)
    VALUES('Remember the Titans',
           GetYear('Remember the Titans'), 'Denzel Washington');
```

…The End of This Lecture…

Q&A