Dynamic SQL

- **Problem**: Application might not know in advance:
  - The SQL statement to be executed
  - The database schema to which the statement is directed
- **Example**: User inputs database name and SQL statement interactively from terminal
- In general, application constructs (as the value of a host language string variable) the SQL statement at run time
- Preparation (necessarily) done at run time

29

Dynamic SQL

- SQL-92 defines syntax for embedding directives into application for constructing, preparing, and executing an SQL statement
  - Referred to as ***Dynamic SQL***
  - Statement level interface
- Dynamic and static SQL can be mixed in a single application

30

Dynamic SQL

```
strcpy (tmp, "SELECT C.NumEnrolled FROM Course C
              WHERE C.CrsCode = ?") ;
EXEC SQL PREPARE st FROM :tmp;
EXEC SQL EXECUTE st INTO :num_enrolled USING :crs_code;
```

placeholder

- st is an SQL variable; names the SQL statement
- tmp, crscode, num_enrolled are host language variables (note colon notation)
- crscode is an *in* parameter; supplies value for placeholder (**?**)
- num_enrolled is an *out* parameter; receives value from C.*NumEnrolled*

31

Dynamic SQL

- PREPARE names SQL statement st and sends it to DBMS for preparation
- EXECUTE causes the statement named st to be executed

32

Parameters: Static vs Dynamic SQL

- Static SQL -
  - Names of (host language) parameters are contained in SQL statement and available to precompiler
  - Address and type information in symbol table
  - Routines for fetching and storing argument values can be generated
  - Complete statement (with parameter values) sent to DBMS when statement is executed

```
EXEC SQL SELECT C.NumEnrolled
    INTO     :num_enrolled
    FROM     Course C
    WHERE C.CrsCode = :crs_code;
```

33

Parameters: Static vs. Dynamic SQL

- Dynamic SQL - SQL statement constructed at run time when symbol table is no longer present
- Case 1: Parameters are known at compile time

```
strcpy (tmp, "SELECT C.NumEnrolled FROM Course C
              WHERE C.CrsCode = ?") ;
EXEC SQL PREPARE st FROM :tmp;
```

  - Parameters are named in EXECUTE statement: *in* parameters in USING; *out* parameters in INTO clauses

```
EXEC SQL EXECUTE st INTO :num_enrolled USING :crs_code;
```

  - EXECUTE statement is compiled using symbol table
    - *fetch()* and *store()* routines generated

34

Parameters - Dynamic SQL (parameters known at compile time)

- Fetch and store routines are executed at client when EXECUTE is executed to communicate argument values with DBMS
- EXECUTE can be invoked multiple times with different values of *in* parameters
  - Each invocation uses same query execution plan
- Values substituted for placeholders by DBMS (in order) at invocation time and statement is executed

35

Parameters in Dynamic SQL (parameters supplied at runtime)

- Case 2: Parameters not known at compile time
- *Example*: Statement input from terminal
  - Application cannot parse statement and might not know schema, so it does not have any parameter information
- EXECUTE statement cannot name parameters in INTO and USING clauses

36

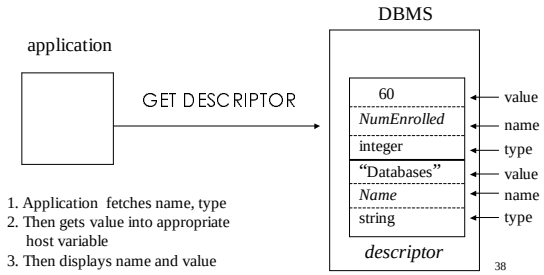
Parameters in Dynamic SQL (cont'd) (parameters supplied at runtime)

- DBMS determines number and type of parameters after preparing the statement
- Information stored by DBMS in a ***descriptor*** – a data structure inside the DBMS, which records the *name*, *type*, and *value* of each parameter
- Dynamic SQL provides directive GET DESCRIPTOR to get information about parameters (e.g., number, name, type) from DBMS and to fetch value of *out* parameters
- Dynamic SQL provides directive SET DESCRIPTOR to supply value to *in* parameters

37

Descriptors

temp = "SELECT C.*NumEnrolled*, C.*Name* FROM Course C \
WHERE C.*CrsCode* = 'CS305' "



1. Application fetches name, type
2. Then gets value into appropriate host variable
3. Then displays name and value

38

Dynamic SQL Calls when Descriptors are Used

```
… … construct SQL statement in temp …..
EXEC SQL PREPARE st FROM :temp;          // prepare statement

EXEC SQL ALLOCATE DESCRIPTOR 'desc';   // create descriptor
EXEC SQL DESCRIBE OUTPUT st USING
        SQL DESCRIPTOR 'desc';          // populate desc with info
                                        // about out parameters

EXEC SQL EXECUTE st INTO                // execute statement and
        SQL DESCRIPTOR AREA 'desc';     // store out values in desc

EXEC SQL GET DESCRIPTOR 'desc' …;      // get out values

… … similar strategy is used for in parameters … …
```

39

Example: Nothing Known at Compile Time

```
sprintf(my_sql_stmt,
        "SELECT * FROM %s WHERE COUNT(*) = 1",
        table);   // table – host var; even the table is known only at run time!

EXEC SQL PREPARE st FROM :my_sql_stmt;
EXEC SQL ALLOCATE DESCRIPTOR 'st_output';

EXEC SQL DESCRIBE OUTPUT st USING SQL DESCRIPTOR 'st_output'
```

- The SQL statement to execute is known only at run time
- At this point DBMS knows what the exact statement is (including the table name, the number of *out* parameters, their types)
- The above statement asks to create descriptors in st_output for all the (now known) *out* parameters

```
EXEC SQL EXECUTE st INTO SQL DESCRIPTOR 'st_output';
```

40

Example: Getting Meta-Information from a Descriptor

- More details in the book:

```
// Host var colcount gets the number of out parameters in the SQL statement
// described by st_output
EXEC SQL GET DESCRIPTOR 'st_output' :colcount = COUNT;

// Set host vars coltype, collength, colname with the type, length, and name
// of the 3-d out parameter in the SQL statement described by st_output
EXEC SQL GET DESCRIPTOR 'st_output' VALUE 3;
    :coltype = TYPE,  // predefined integer constants, such as SQL_CHAR, SQL_FLOAT,…
    :collength = LENGTH,
    :colname = NAME;
```

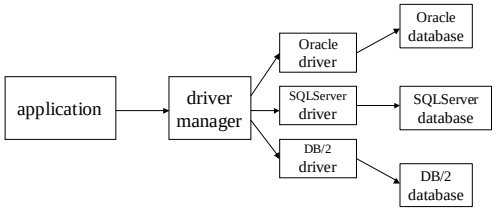
41

JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS. Can be interfaced to any DBMS that has a JDBC driver

42

JDBC Run-Time Architecture



43

Executing a Query

```
import java.sql.*;      -- import all classes in package java.sql

Class.forName (driver name);      // static method of class Class
                                  // loads specified driver

Connection con = DriverManager.getConnection(Url, Id, Passwd);
      // Static method of class DriverManager; attempts to
      // connect to DBMS; if successful, creates a connection
      // object, con, for managing the connection

Statement stat = con.CreateStatement ();
      // Creates a statement object stat;
      // statements have executeQuery() method
```

44

Executing a Query

```
String query = "SELECT  T.StudId  FROM  Transcript T" +
               "WHERE   T.CrsCode = 'cse305' " +
               "AND  T.Semester = 'S2000' ";
ResultSet  res = stat.executeQuery (query);
```

- *Creates a result set object,* res*.*
- *Prepares and executes the query.*
- *Stores the result set produced by execution in* res *(analogous to opening a cursor).*
- *The query string can be constructed at run time (as above).*
- *The input parameters are plugged into the query when the string is formed (as above)*

45

Preparing and Executing a Query

```
String query = "SELECT  T.StudId  FROM  Transcript T" +
               "WHERE  T.CrsCode = ?  AND  T.Semester = ?";
```

placeholders

```
PreparedStatement  ps = con.prepareStatement ( query );
```

- *Prepares the statement*
- *Creates a prepared statement object,* ps*, containing the prepared statement*
- Placeholders *(***?***) mark positions of* in *parameters; special API is provided to plug the actual values in positions indicated by the* **?***'s*

46

Preparing and Executing a Query

```
String  crs_code, semester;
……..
ps.setString(1, crs_code);     // set value of first in parameter
ps.setString(2, semester);     // set value of second in parameter

ResultSet  res = ps.executeQuery ( );
        // Creates a result set object, res.
        // Executes the query
        // Stores the result set produced by execution in res.

while  (res.next () )  {                    // advance the cursor
      j = res.getInt ("StudId");            // fetch output int-value
      …process output value…
}
```


Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable, changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable, changes made to the tuples in a result set after the creation of that set are visible through the set

Result Set

```
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE  );
```

- Any result set type can be declared *read-only* or *updatable* – CONCUR_UPDATABLE (assuming SQL query satisfies the conditions for updatable views)
- Current row of an updatable result set can be updated or deleted, and a new row can be inserted, causing changes in base table

```
res.updateString ("Name", "John" );   // update attribute "Name" of
                                      // current row in row buffer.
res.updateRow ( );   // install current row buffer in res and underlying table
```


Handling Exceptions

```
try {
    ...Java/JDBC code...
}  catch ( SQLException  ex ) {
    …exception handling code...
}
```

- try/catch is the basic structure within which an SQL statement should be embedded
- If an exception is thrown, an exception object, *ex*, is created and the catch clause is executed
- The exception object has methods to print an error message, return SQLSTATE, etc.

Transactions in JDBC

- Default for a connection is
  - Transaction boundaries
    - *Autocommit mode*: each SQL statement is a transaction.
    - To group several statements into a transaction use con.setautocommit (false)
  - Isolation
    - default isolation level of the underlying DBMS
    - To change isolation level use con.setTransactionIsolationLevel (TRANSACTION_SERIALIZABLE)
- With autocommit off:
  - transaction is committed using con.commit().
  - next transaction is automatically initiated (chaining)
- Transactions on each connection committed separately

SQLJ

- A statement-level interface to Java
  - A dialect of embedded SQL designed specifically for Java
  - Translated by precompiler into Java
  - SQL constructs translated into calls to an SQLJ runtime package, which accesses database through calls to a JDBC driver

SQLJ

- Has some of efficiencies of embedded SQL
  - Compile-time syntax and type checking
  - Use of host language variables
  - More elegant than embedded SQL
- Has some of the advantages of JDBC
  - Can access multiple DBMSs using drivers
  - SQLJ statements and JDBC calls can be included in the same program

53

SQLJ Example

```
#SQL {
    SELECT  C.Enrollment
    INTO :numEnrolled
    FROM Class C
    WHERE C.CrsCode = :crsCode
          AND C.Semester = :semester
 };
```

54

Example of SQLJ Iterator

```
#SQL iterator GetEnrolledIter (int studentId, String studGrade);
GetEnrolledIter iter1;

#SQL iter1 = {
     SELECT  T.StudentId as "studentId",
             T.Grade as "studGrade"
     FROM  Transcript T
     WHERE  T.CrsCode  =  :crsCode
              AND T.Semester =  :semester
};
```

Method names by which to access the attrs StudentId, Grade

55

Iterator Example (Cont'd)

```
int id;
String grade;
while  (iter1.next()  {
      id  =  iter1.studentId();
      grade = iter1.studGrade();
         … process the values in id and grade …
};

iter1.close();
```

56

ODBC

- Call level interface that is database independent
- Related to SQL/CLI, part of SQL:1999
- Software architecture similar to JDBC with driver manager and drivers
- Not object oriented
- Low-level: application must specifically allocate and deallocate storage

57

Sequence of Procedure Calls Needed for ODBC

```
SQLAllocEnv(&henv);                  // get environment handle
SQLAllocConnect(henv, &hdbc);    // get connection handle
SQLConnect(hdbc, db_name, userId, password);  // connect
SQLAllocStmt(hdbc, &hstmt);        // get statement handle
SQLPrepare(hstmt, SQL statement); // prepare SQL statement
SQLExecute(hstmt);
SQLFreeStmt(hstmt);                // free up statement space
SQLDisconnect(hdbc);
SQLFreeEnv(henv);                  // free up environment space
```

58

ODBC Features

- Cursors
  - *Statement handle* (for example hstmt) is used as name of cursor
- Status Processing
  - Each ODBC procedure is actually a function that returns status

    RETCODE retcode1;
    Retcode1 = SQLConnect (…)
- Transactions
  - Can be committed or aborted with

    SQLTransact (henv, hdbc, SQL_COMMIT)

59