Integrity Constraint Checking

- Transaction moves database from an initial to a final state, both of which satisfy all integrity constraints but...
  - Constraints might not be true of intermediate states hence ...
  - Constraint checks at statement boundaries might be inappropriate

- SQL (optionally) allows checking to be deferred to transaction COMMIT
Deferred Constraint Checking

Schema:

```sql
CREATE CONSTRAINT NumberEnrolled
    CHECK ( .......)
    DEFERRABLE;
```

Application:

```sql
SET CONSTRAINT NumberEnrolled DEFERRED;
```

Transaction is aborted if constraint is false at commit time

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
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

```c
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;
```

- `st` is an SQL variable (only in SQL statements); 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`
Dynamic SQL

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

Parameters: Static vs Dynamic SQL

- **Static SQL:**
  - Names of (host language) parameters are contained in SQL statement and available to precompiler
    - Recall DECLARE SECTION
  - Symbol table has mappings between variable names, address and type information
  - Routines for fetching and storing argument values can be generated
  - Complete statement (with parameter values) sent to DBMS when statement is executed

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

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

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

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

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

Parameters – Dynamic SQL
(Case 1: 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
Parameters in Dynamic SQL (parameters supplied at runtime)

- **Case 2: Parameters are 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

---

Parameters in Dynamic SQL (parameters supplied at runtime)

- **A simple case**

```c
printf("Which column of CLASS would you like to see?\n");
scanf("%s", column); // get user input(Enrollment or Room)
// Incorporate user input into SQL statement
sprintf(my_sql_stmt, 
    "SELECT C.%s FROM CLASS C \n      WHERE C.CrsCode = ? AND C.Semester = ?", 
    column);
EXEC SQL PREPARE statements FROM :my_sql_stmt; 
EXEC SQL EXECUTE stmt 
  INTO: some_string_var 
  USING :crs_code, :semester;
```
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

Descriptors

temp = “SELECT C.NumEnrolled, C.Name FROM Course C \\
WHERE C.CrsCode = 'CS305' ”

1. Application uses GET DESCRIPTOR to fetch name, type, value
2. Then gets value into appropriate host variable
3. Then processes value
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 ... ...

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';
  ● The value of i^th attribute of the row returned to be stored in the value field of the i^th
    entry in st_output
Example: Getting Meta-Information from a Descriptor

// 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
// colnumber’s out parameter in the SQL statement described by st_output
EXEC SQL GET DESCRIPTOR ‘st_output’ VALUE :colnumber;
:coltype = TYPE, // predefined integer constants, such as SQL_CHAR, SQL_FLOAT,…
:collength = LENGTH,
:colname = NAME;

Example: Using Meta-Information to Extract Attribute Value

char strdata[1024];
int intdata;
... ...
switch (coltype) {
case SQL_CHAR:
    EXEC SQL GET DESCRIPTOR ‘st_output’ VALUE :colnumber :strdata=DATA;
    break;
case SQL_INT:
    EXEC SQL GET DESCRIPTOR ‘st_output’ VALUE :colnumber :intdata=DATA;
    break;
case SQL_FLOAT:
    ... ...
}

Put the value of attribute colnumber into the variable strdata
Example of using GET DESCRIPTOR

```
int collength, coltype, colcount;
char colname[265];
// Arrange variables for different types of data
char stringdata[1024];
int indata;
float floatdata;
// variable declarations for other types
// Store the number of columns in colcount
EXEC SQL GET DESCRIPTOR 'st_output' : colcount = COUNT;
for (i=0; i < colcount; i++) {
  // Get meta-information about the ith attribute
  // Note: type is represented by an integer constant, such as
  // SQL_CHAR, SQL_INTEGER, SQL_FLOAT, defined in a header file
  EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i
    : coltype = TYPE;
    : collength = LENGTH;
    : colname = NAME;
  printf("%s %s has value: ", colname);
  switch (coltype) {
    case SQL_CHAR:
      EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i: stringdata = DATA;
      printf(""L%s", stringdata); // print string value
      break;
    case SQL_INTEGER:
      EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i: indata = DATA;
      printf("%4d", indata); // print integer value
      break;
    case SQL_FLOAT:
      EXEC SQL GET DESCRIPTOR 'st_output' VALUE :i: floatdata = DATA;
      printf("%f", floatdata); // print floating point value
      break;
    // other cases
  } // switch
} // for loop
```

Cursors in Dynamic SQL and PSM

- Cursors can be defined similarly to static SQL
  ```sql
  my_sql_stmt = "SELECT T.StuId, T.Grade 
  FROM Transcript T 
  WHERE T.CrsCode = ? 
  AND T.Semester = ?";
  EXEC SQL PREPARE st2 FROM :my_sql_stmt;
  EXEC SQL DECLARE GETEnROLL INSENSITIVE CURSOR FOR st2;
  EXEC SQL OPEN GETEnROLL USING :crs_code, :semester;
  EXEC SQL FETCH GETEnROLL INTO :stud_id, :grade;
  EXEC SQL CLOSE GETEnROLL;
  ```

- Use of procedures stored on the server
  ```sql
  my_sql_stmt = "CALL Deregister(?,?,?)";
  EXEC SQL PREPARE st3 FROM :my_sql_stmt;
  EXEC SQL EXECUTE st3
  USING :crs_code, :semester, :stud_id;
  ```
JDBC (Java Database Connectivity)

- **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

- Part of SQL:2003
  - See object language bindings section
  - Also with SQLJ, statement-level interface to Java

---

JDBC Run-Time Architecture

![Diagram of JDBC Run-Time Architecture]
Executing a Query

```java
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
```

Executing a Query (cont’d)

```java
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)
```
Preparing and Executing a Query

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

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

Preparing and Executing a Query (cont’d)

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…
}

52

53
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; updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the 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)
- Updatable: Current row of an updatable result set can be changed or deleted, or a new row can be inserted. Any such change causes changes to the underlying database table

res.updateString (“Name”, “John”); // change the attribute “Name” of current row in the row buffer.
res.updateRow (); // install changes to the current row buffer in the underlying database 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.

Obtaining Metadata on Result Set and Status

- Used to know ResultSet structure
  ```
  ResultSet rs3 = stmt3.executeQuery("SELECT * FROM Table3");
  ResultSetMetaData rsm3 = rs3.getMetaData();
  ```
  ```
  int numCols = rsm3.getColumnCount();
  String colName = rsm3.getColumnLabel(1);
  String typeName = rsm3.getColumnTypeName(1);
  ```

- Methods for status processing
  ```
  Try{...}
  Catch (SQLException e){
      .... e.getMessage();
      .... e.getSQLState();  // “00XXX” (success), “01XXX” (warning), “02000” (no data)
      .... e.getErrorCode();
  }
  ```
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(TXN_SERIALIZABLE)`

- With autocommit off:
  - transaction is committed using `con.commit()`.
  - next transaction is automatically initiated (chaining)

- Transactions on each connection committed separately

---

Stored Procedure on the Server

- Use `CallableStatement` to call stored procedure

  CallableStatement cs5 =
  con5.prepareCall("call Deregister(?,?,?,?)");
  cs5.setString(1, crs_code);
  cs5.setString(2, semester);
  ....
  cs5.executeUpdate();

  Or

  ResultSet rs5 = cs5.executeQuery();
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
  - Part of SQL:2003

- Has some of **runtime 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 (**abstraction**)
  - Can access multiple DBMSs using drivers
  - SQLJ statements and JDBC calls can be included in the same program
  - More portable across DBMSes

**SQLJ Example**

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

```sql
EXEC SQL BEGIN DECLARE SECTION;
  unsigned long num_enrolled;
  char *crs_code, *semester;
  char SQLSTATE [6];
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT C.NumEnrolled INTO num_enrolled
FROM Course C
WHERE C.CrsCode = :crs_code AND
  C.Semester = :semester;
```
Example of SQLJ Iterator

- Similar to JDBC’s ResultSet; provides a cursor mechanism

```sql
#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
};
```

Iterator Example (cont’d)

- Similar to JDBC’s ResultSet; provides a cursor mechanism

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

iter1.close();
```
ODBC (Open Database Connectivity)

- 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

Sequence of Procedure Calls Needed for ODBC

In ODBC 3.x, replaced by SQLAllocHandle

- 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

In ODBC 3.x, replaced by SQLFreeHandle
ODBC Features

- **Prepare and Execute**
  - SQLPrepare(hstmt, ... SQL statement...)
  - SQLBindParameters(hstmt, 1, SQL_PARAMETER_INPUT, SQL_C_SSHORT, SQL_SMALLINT, &int1);
  - SQLExecute(hstmt);

- **Cursors**
  - *Statement handle* (for example hstmt) is used as name of cursor
  - SQLBindCol, SQLFetch, SQLGetData, ...
  - SQLSetStmtOption for cursor type (e.g., SQL_CURSOR_KEYSET_DRIVEN)

- **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)