Week 5: Embedded SQL

Update Statements

Embedded SQL

- → Traditional applications often need to "**embed**" SQL statements inside the instructions of a procedural programming language (C, COBOL, etc.)
- → Programs with embedded SQL use a **pre-compiler** to manage SQL statements. Embedded statements are preceded by '**$**' or '**EXEC SQL**'
- → Program variables may be used as parameters in the SQL statements (preceded by ':')
- → `select` statements producing a single row and update statements can be embedded easily.
- → The SQL environment offers a predefined variable `sqlcode` which describes the execution status of an SQL statement (=0 if it executed successfully).

Interactive vs. Non-Interactive SQL

- → **Non-interactive SQL**: Statements are included in an application program written in a host language — such as C, Java, COBOL
- → **Interactive SQL**: Statements input from terminal; DBMS outputs to screen
- → Interactive SQL is inadequate for most uses:
  - ✓ It may be necessary to process the data before output;
  - ✓ Amount of data returned not known in advance;
  - ✓ SQL has limited expressive power — ***note: not Turing-complete.***

Application Program

- → ***Host language***: A conventional programming language (*e.g*., C, Java) that supplies control structures, computational capabilities, interaction with physical devices.
- → ***SQL***: supplies ability to interact with database.
- → ***Using the facilities of both***: the application program can act as an intermediary between the user at a terminal and the DBMS.

Preparation

- Before any SQL statement is executed, it must be ***prepared*** by the DBMS:
  - What indices can be used?
  - In what order should tables be accessed?
  - What constraints should be checked?
- Decisions are based on schema, table sizes, etc.
- Result is a ***query execution plan.***
- Preparation is a complex activity, usually done at run time, justified by the complexity of query processing.

Introducing SQL Into the Application

- SQL statements can be incorporated into an application program in two different ways.
- ***Statement Level Interface*** (SLI): Application program is a mixture of host language statements and SQL statements and directives.
- ***Call Level Interface*** (CLI): Application program is written entirely in host language.
- SQL statements are values of string variables that are passed as arguments to host language (library) procedures

Statement Level Interface

- SQL statements and directives in the application have a ***special syntax*** that sets them off from host language constructs

  e.g., **EXEC SQL** ***SQL_statement***
- ***Pre-compiler*** scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS.
- ***Host language compiler*** then compiles program.

Statement Level Interface

- SQL constructs in an application take two forms:
  - Standard SQL statements (***static*** or ***embedded*** SQL): Useful when SQL portion of program is known at ***compile time***
  - Directives (***dynamic*** SQL): Useful when SQL portion of program not known at compile time. Application constructs SQL statements ***at run time*** as values of host language variables that are manipulated by directives
- Pre-compiler translates statements and directives into arguments of calls to library procedures.

Call Level Interface

- →Application program written entirely in host language (no precompiler)

  Examples: JDBC, ODBC
- →SQL statements are values of string variables constructed *at run time* using host language

  Similar to dynamic SQL
- →Application uses string variables as arguments of library routines that communicate with DBMS

  e.g. executeQuery("*SQL query statement*")

Static SQL

```
EXEC SQL BEGIN DECLARE S;
  unsigned long num_enrolled;
  char crs_code;
  char SQLSTATE [6];
EXEC SQL END DECLARE SE;
      .........
EXEC SQL SELECT C.NumEnrolled
    INTO :num_enrolled
    FROM Course C
    WHERE C.CrsCode = :crs_code;
```

Variables shared by host and SQL

":" used to set off host variables

- →Declaration section for host/SQL communication.
- →Colon convention for value (WHERE) and result (INTO) parameters.

Status

```
EXEC SQL SELECT C.NumEnrolled
    INTO :num_enrolled
    FROM Course C
    WHERE C.CrsCode = :crs_code;
if ( !strcmp (SQLSTATE, "00000") ) {
    printf ( "statement failed" )
};
```

***Out* parameter**

***In* parameter**

Connections

- →To connect to an SQL database, use a connect statement

  CONNECT TO *database_name* AS *connection_name* USING *user_id*

Transactions

- →No explicit statement is needed to begin a transaction: A transaction is initiated when the first SQL statement that accesses the database is executed.
- →The mode of transaction execution can be set with

  SET TRANSACTION READ ONLY

  ISOLATION LEVEL SERIALIZABLE
- →Transactions are terminated with COMMIT or ROLLBACK statements.

Example: Course Deregistration

```
EXEC SQL CONNECT TO :dbserver;
if ( ! strcmp (SQLSTATE, “00000”) ) exit (1);
  …..
EXEC SQL DELETE FROM Transcript T
  WHERE T.StudId = :studid AND T.Semester = ‘S2000’
               AND T.CrsCode = :crscode;
if (! strcmp (SQLSTATE, “00000”) ) EXEC SQL ROLLBACK;
else {
  EXEC SQL UPDATE Course C
        SET C.Numenrolled = C.Numenrolled – 1
        WHERE C.CrsCode = :crscode;
   if (! strcmp (SQLSTATE, “00000”) ) EXEC SQL ROLLBACK;
   else EXEC SQL COMMIT;
}
```

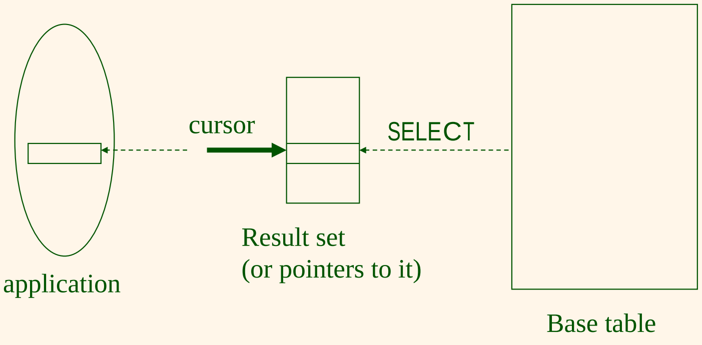
Buffer Mismatch Problem

- →Problem: SQL deals with tables (of arbitrary size); host language program deals with fixed size buffers
  - ✓How is the application to allocate storage for the result of a SELECT statement?
- →Solution: Fetch a single row at a time
  - ✓Space for a single row (number and type of *out* parameters) can be determined from schema and allocated in application

Cursors

- →*Result set* – set of rows produced by a SELECT statement
- →*Cursor* – pointer to a row in the result set.
- →Cursor operations:
  - ✓*Declaration*
  - ✓*Open* – execute SELECT to determine result set and initialize pointer
  - ✓*Fetch* – advance pointer and retrieve next row
  - ✓*Close* – deallocate cursor

Cursors (cont'd)



cursor

SELECT

application

Result set
(or pointers to it)

Base table

Example of Cursor Use

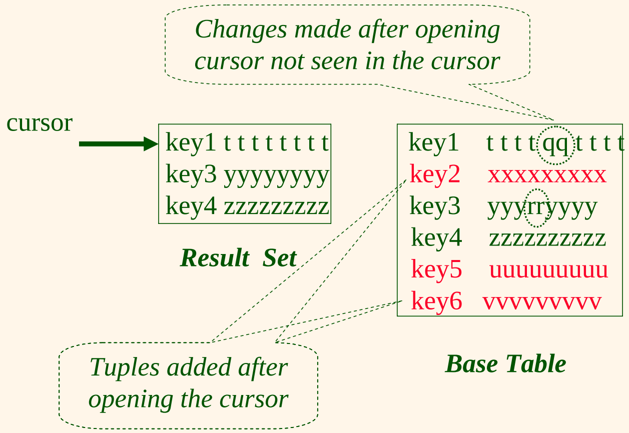
```
EXEC SQL DECLARE GetEnroll INSENSITIVE CURSOR FOR
   SELECT T.StudId, T.Grade        — cursor is not a schema element
   FROM Transcript T
   WHERE T.CrsCode = :crscode AND T.Semester = 'S2000';
   .........
EXEC SQL OPEN GetEnroll;
if ( !strcmp ( SQLSTATE, "00000")) {... fail exit... };
   ..........
EXEC SQL FETCH GetEnroll INTO :studid, :grade;
while ( SQLSTATE = "00000") {
   ... process the returned row...
   EXEC SQL FETCH GetEnroll INTO :studid, :grade;
}
if ( !strcmp ( SQLSTATE, "02000")) {... fail exit... };
   ..........
EXEC SQL CLOSE GetEnroll;
```

Reference resolved at compile time,
Value substituted at OPEN time

Cursor Types

- →***Insensitive cursor*: Result set (effectively) computed and stored in a separate table at OPEN time**
  - ✓**Changes made to base table subsequent to OPEN (by any transaction) do not affect result set**
  - ✓**Cursor is read-only**
- →***Cursors that are not insensitive*: Specification not part of SQL standard**
  - ✓**Changes made to base table subsequent to OPEN (by any transaction) can affect result set**
  - ✓**Cursor is updatable**

Insensitive Cursor



Changes made after opening cursor not seen in the cursor

cursor

Result Set
key1 t t t t t t t t
key3 yyyyyyyy
key4 zzzzzzzzz

Result Set

Base Table	
key1	t t t t qq t t t t
key2	xxxxxxxxx
key3	yyyrryyyy
key4	zzzzzzzzzz
key5	uuuuuuuuu
key6	vvvvvvvvv

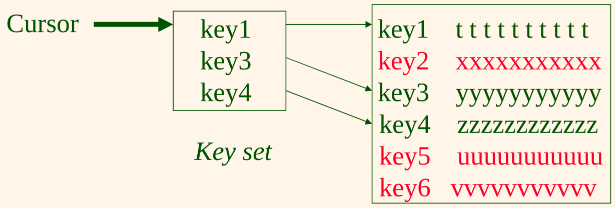
Base Table

Tuples added after opening the cursor

Keyset-Driven Cursor

- → Example of a cursor that is not insensitive.
- → Primary key of each row in result set is computed at open time.
- → UPDATE or DELETE of a row in base table by a concurrent transaction between OPEN and FETCH might be seen through cursor.
- → INSERT into base table, however, not seen through cursor.
- → Cursor is updatable.

Keyset-Driven Cursor



Base table

Tuples added after cursor is open are not seen, but updates to key1, key3, key4 are seen in the cursor.

Cursors

```
DECLARE cursor-name [INSENSITIVE] [SCROLL]
     CURSOR FOR  table-expr
    [ ORDER BY column-list ]
    [ FOR {READ ONLY | UPDATE [ OF column-list ] } ]
```

For updatable (not insensitive, not read-only) cursors

```
    UPDATE  table-name                      — base table
         SET  assignment
        WHERE CURRENT OF  cursor-name
    DELETE  FROM  table-name                — base table
         WHERE CURRENT OF  cursor-name
```

Restriction – *table-expr* must satisfy restrictions of updatable view

Scrolling

- → If SCROLL option not specified in cursor declaration, FETCH always moves cursor forward one position
- → If SCROLL option is included in DECLARE CURSOR section, cursor ca arbitrary ways around result set:

Get previous tuple

```
FETCH PRIOR FROM GetEnroll INTO :studid, :grade;
```

- Also: FIRST, LAST, ABSOLUTE n, RELATIVE n

Stored Procedures

- →Procedure – written in a conventional algorithmic language
  - ✓Included as schema element (stored in DBMS)
  - ✓Invoked by the application
- →Advantages:
  - ✓Intermediate data need not be communicated to application (time and cost savings)
  - ✓Procedure's SQL statements prepared in advance
  - ✓Authorization can be done at procedure level
  - ✓Added security since procedure resides in server
  - ✓Applications that call the procedure need not know the details of database schema – all database access is encapsulated within the procedure

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 :crscode;
```

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*

Dynamic SQL

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

Parameters for Static SQL

For Static SQL:

- →Names of (host language) parameters are contained in SQL statement and available to pre-compiler.
- →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;
```

Parameters for 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**

Parameters for 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)

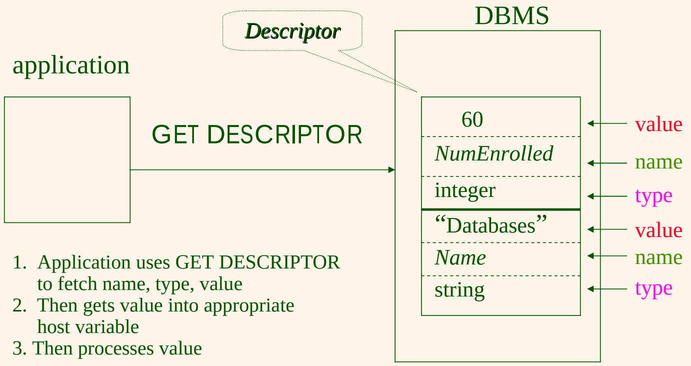
- →**Case 2: Parameters *not* known at compile time**
- →***Example*: Statement input from terminal**
  - ✓**Application cannot parse statement and might not know schema, so it does not have any parameter information**
- →**EXECUTE statement cannot name parameters in INTO and USING clauses**

Parameters in Dynamic SQL
(Case 2: 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’;
```

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

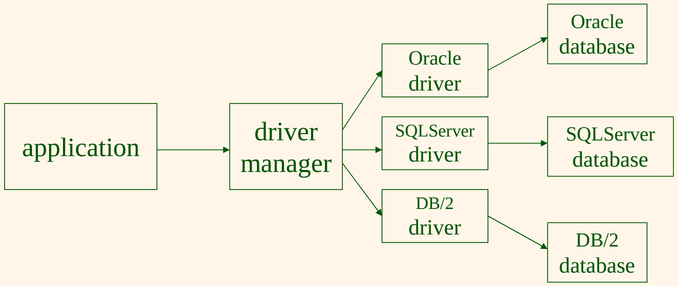
```
char strdata[1024];
int intdata;
… … …
switch (coltype) {
case SQL_CHAR:
 EXEC SQL GET DESCRIPTOR ‘st_output’ VALUE :colnumber strdata=DATA
 break;
case SQL_INT:
 EXEC SQL GET DESCRIPTOR ‘st_output’  VALUE   :colnumber
   :intdata=DATA;
 break;
case SQL_FLOAT:
   … … …
}
```

Put the value of attribute colnumber *into the variable* strdata

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
- →Part of SQL:2003

JDBC Run-Time Architecture



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*

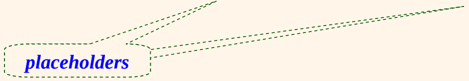
Executing a Query (cont'd)

```
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 = ?";
```



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*

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…
}
```

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**

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)
- →*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

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().

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

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

SQLJ Example

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

Example of SQLJ Iterator

- →Similar to JDBC's ResultSet; provides a cursor mechanism

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

Method names by which to access the attributes ***StudentId*** *and* ***Grade***

```
#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)

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

iter1.close();
```

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

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
```

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

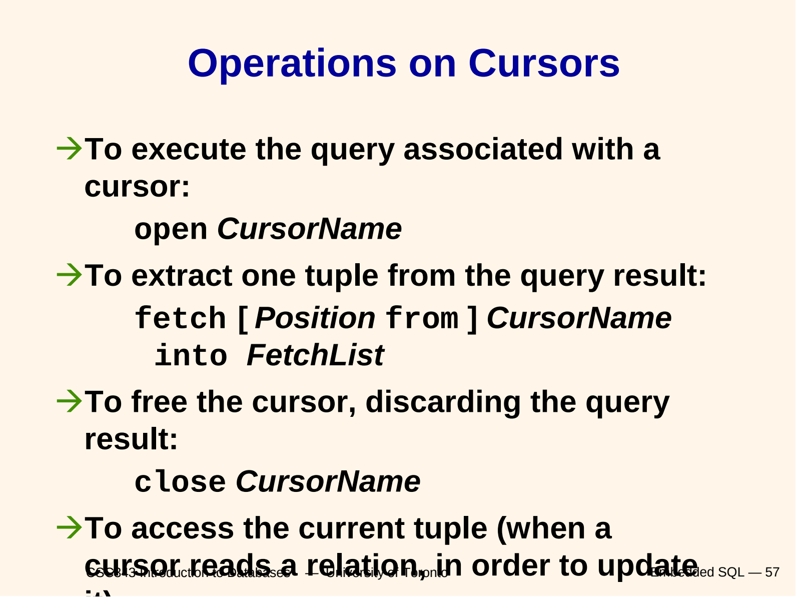
        RETCODE retcode1;
        Retcode1 = SQLConnect (…)
- →Transactions
  - ✓Can be committed or aborted with

        SQLTransact (henv, hdbc,
        SQL_COMMIT)

Cursors

- →Fundamental problem with database technology: ***impedance mismatch*** — traditional programming languages process records one-at-a-time (tuple-oriented); SQL processes tuple sets (set-oriented).
- →Cursors solve this problem: A cursor accesses the result of a query in a set-oriented way, returns tuples for the program to process one-by-one.
- →Syntax of cursor definition:

  `declare` *CursorName* `[ scroll ]`
  `cursor for` *SelectSQL*
  `[ for < read only | update [ of` *Attribute*



Operations on Cursors

- To execute the query associated with a cursor:

  open *CursorName*

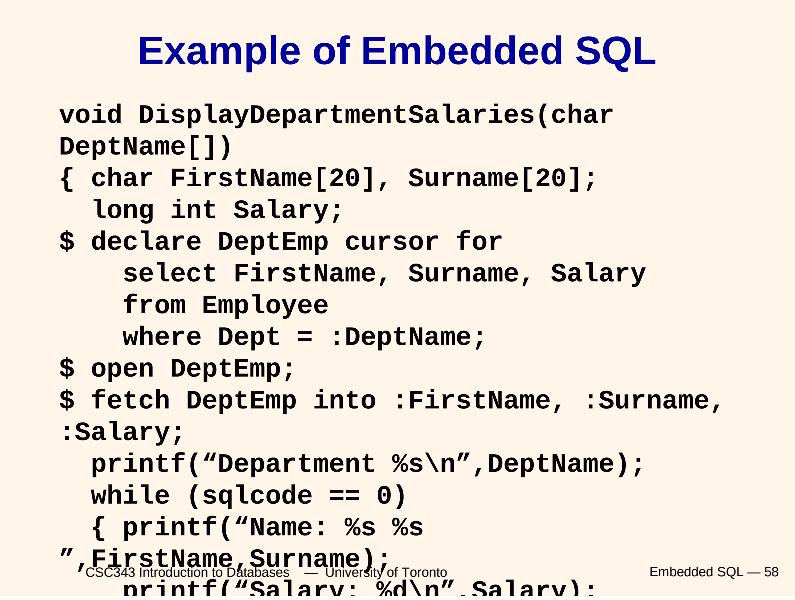
- To extract one tuple from the query result:

  `fetch` [*Position* `from`] *CursorName*
  `into` *FetchList*

- To free the cursor, discarding the query result:

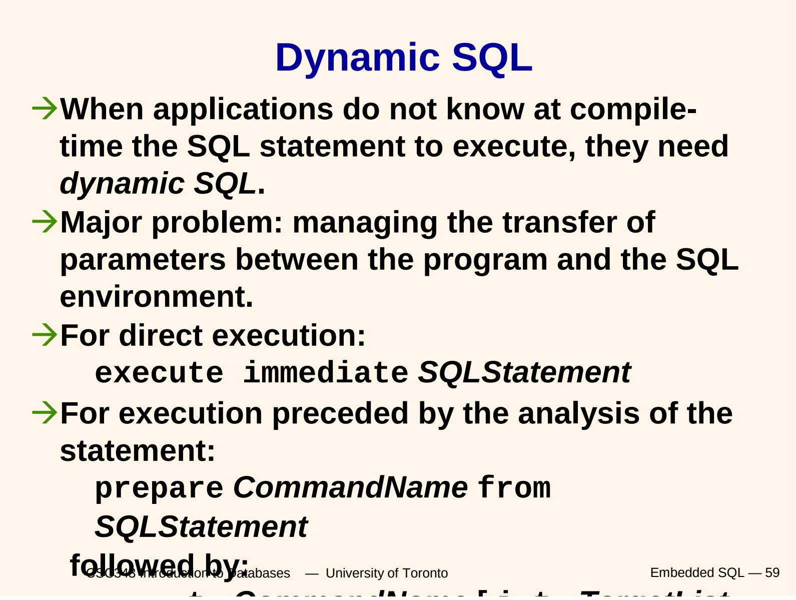
  `close` *CursorName*

- To access the current tuple (when a cursor reads a relation, in order to update it)



Example of Embedded SQL

```
void DisplayDepartmentSalaries(char
DeptName[])
{ char FirstName[20], Surname[20];
  long int Salary;
$ declare DeptEmp cursor for
    select FirstName, Surname, Salary
    from Employee
    where Dept = :DeptName;
$ open DeptEmp;
$ fetch DeptEmp into :FirstName, :Surname,
:Salary;
  printf(“Department %s\n”,DeptName);
  while (sqlcode == 0)
  { printf(“Name: %s %s
”,FirstName,Surname);
    printf(“Salary: %d\n”,Salary);
```



Dynamic SQL

- When applications do not know at compile-time the SQL statement to execute, they need *dynamic SQL*.
- Major problem: managing the transfer of parameters between the program and the SQL environment.
- For direct execution:

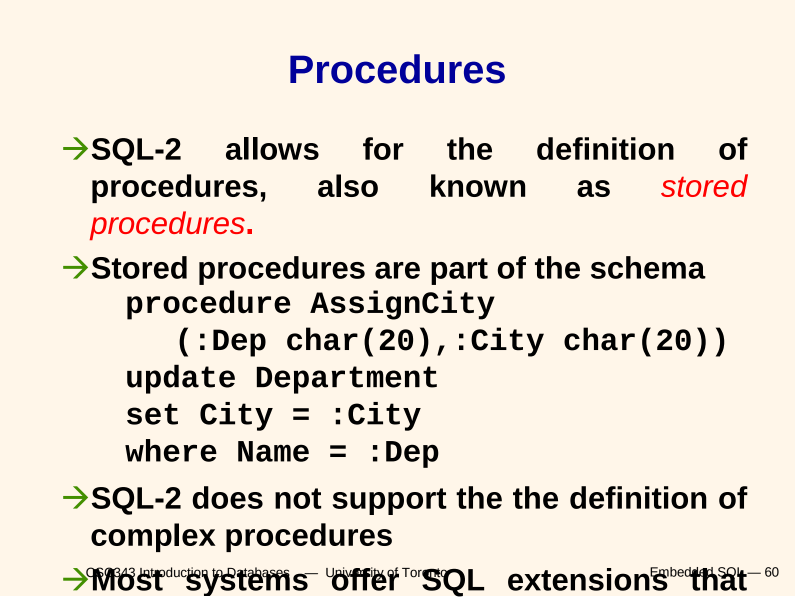
  `execute immediate` *SQLStatement*

- For execution preceded by the analysis of the statement:

  `prepare` *CommandName* `from`
  SQLStatement

  followed by:

  `execute` *CommandName* [`into` *TargetList*

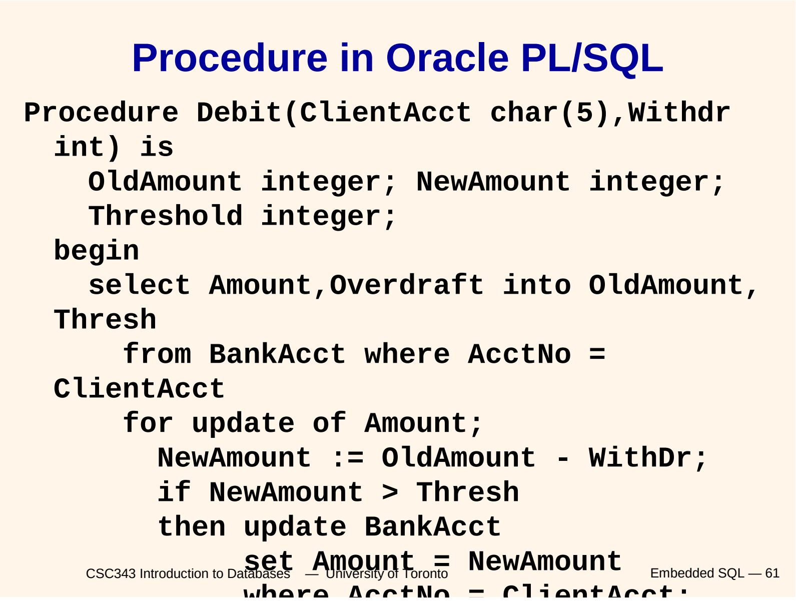


Procedures

- SQL-2 allows for the definition of procedures, also known as *stored procedures*.
- Stored procedures are part of the schema

```
procedure AssignCity
   (:Dep char(20),:City char(20))
update Department
set City = :City
where Name = :Dep
```

- SQL-2 does not support the the definition of complex procedures
- Most systems offer SQL extensions that



Procedure in Oracle PL/SQL

```
Procedure Debit(ClientAcct char(5),Withdr
  int) is
    OldAmount integer; NewAmount integer;
    Threshold integer;
  begin
    select Amount,Overdraft into OldAmount,
  Thresh
      from BankAcct where AcctNo =
  ClientAcct
      for update of Amount;
        NewAmount := OldAmount - WithDr;
        if NewAmount > Thresh
        then update BankAcct
             set Amount = NewAmount
             where AcctNo = ClientAcct;
```