Module 6: Applications Programming with Dynamic SQL

Fall 2026

Cheriton School of Computer Science

CS 338: Intro to Database Management

Outline

Unit 1: **Dynamic Embedded SQL (in C)**

Unit 2: ODBC

Dynamic Embedded SQL

Dynamic SQL

A protocol for using SQL within another language is *dynamic* if it enables arbitrary SQL source code to be constructed and executed at run-time.

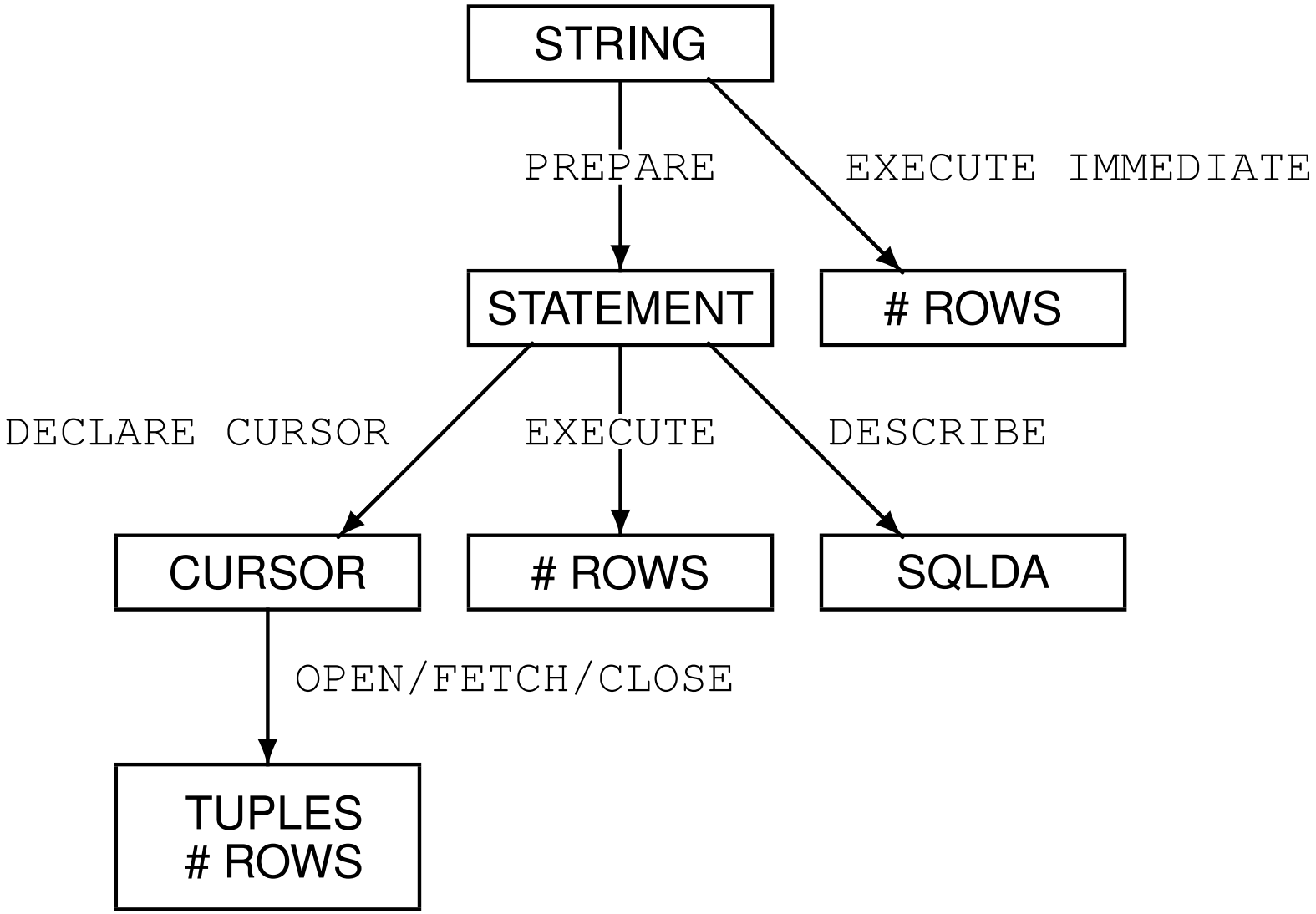
Problems:

- How do we know a string is a valid statement?
  $\Rightarrow$ parsing and compilation at run-time
- How do we execute valid statements?
  $\Rightarrow$ supplying parameters
  $\Rightarrow$ retrieving results
- Extreme case: *nothing* is known about the string at compile time.

We develop an *adhoc* application that accepts an SQL statement as an argument, executes it, and prints out answers if the statement is a query.

Part 1 , 2 , 3

Dynamic Embedded SQL: a Roadmap



EXECUTE IMMEDIATE

To execute a *non-parametric* statement, use the command

```
EXEC SQL EXECUTE IMMEDIATE :string;
```

where `:string` is a host variable containing the ASCII representation of the command.

- `:string` may not be a command that returns answers, such as a query.
- EXECUTE IMMEDIATE should only be used for commands that will be executed only once by an app.

  $\Rightarrow$ `:string` is *compiled* each time control flow reaches the command

PREPARE

Run-Time Compilation

The SQL PREPARE command is used at run-time to *compile* SQL commands and to provide a *handle* to the generated code.

To compile, use the command

```
EXEC SQL PREPARE stmt FROM :string;
```

Code generated for `:string` can now be invoked via the `stmt` handle in other SQL commands.

$\Rightarrow$ enables recompilation of `:string` to be avoided

- `:string` may now contain parameters and be a query that returns answers.
- The `stmt` handle is *not* a host variable, and refers to generated code for `:string` that will not be reentrent.

  $\Rightarrow$ statement handles should not be used in recursion

Parametric Statements

How does an application communicate parameter values with statement handles for compiled SQL commands?

- Recall that host variables are used in static embedded SQL.
- In dynamic embedded SQL, there are two possibilities:
  1. Reconstruct the ASCII string encoding a command.
     $\Rightarrow$ will need to recompile
  2. Allow special *markers* in an ASCII string encoding a command.

Parameter Markers

ASCII representations of SQL commands can introduce `"?"` characters as parameter markers for locations of parameter values that will be substituted when the compiled commands are executed.

EXECUTE

To execute a prepared SQL command with parameter markers via a statement handle, use the EXECUTE command with the USING clause.

```
EXEC SQL EXECUTE stmt
         USING   :var1 [,...,:vark];
```

- Used for commands that do not return tuples.
  ⇒ database modification (`INSERT`, ...)
  ⇒ transaction management (`COMMIT`)
  ⇒ data definition (`CREATE TABLE`, ...)
- Values of host variables `:var1` through `:vark` are substituted for the parameter markers in order of appearance.
  ⇒ a mismatch will cause an SQL runtime error

Using SQL Cursors

To execute a prepared SQL query with parameter markers and a statement handle, use the USING clause with the OPEN command and follow the cursor iterator protocol.

```
EXEC SQL DECLARE cname CURSOR FOR stmt;

EXEC SQL OPEN  cname
         USING :var1 [,...,:vark];

EXEC SQL FETCH cname
         INTO  :out1 [,...,:outn];

EXEC SQL CLOSE cname;
```

- Host variables `:var1` through `:vark` supply values for parameter markers.
- Host variables `:out1` through `:outn` store values for a retrieved tuple.
- The number of retrieved tuples is defined in `sqlca.sqlerrd[2]`.

When Parameters and Results are not Known

DESCRIPTOR

An SQL descriptor is used to communicate information about parameters and results of a prepared SQL command.

SQL commands for descriptor use:

- `ALLOCATE DESCRIPTOR descr`
- `GET DESCRIPTOR descr <what>`
  `SET DESCRIPTOR descr <what>`

  where `<what>` indicates
  1. GET/SET a value for `COUNT`, or
  2. GET/SET properties for the *i*-th attribute: `VALUE :i <prop>`
     where `<prop>` can be `DATA`, `TYPE`, `INDICATOR`, ...
- `DESCRIBE [INPUT | OUTPUT] stmt INTO descr`

In practice, one uses an explicit host language `sqlda` data structure.

The SQLDA Data Structure

The `sqlda` data structure is an SQL *description area* that defines what attributes that are parameters and query answers look like, e.g., where the data is located.

This is how the RDBMS communicates with an application.

The data structure contains (among other things):

1. the string `'SQLDA '` (for identification);
2. the number of allocated entries for attributes;
3. the number of actual attributes, 0 if none; and
4. for each attribute:
   - 4.1 a numeric code of its type,
   - 4.2 the length of storage for its value,
   - 4.3 a pointer to a data variable,
   - 4.4 a pointer to a indicator variable, and
   - 4.5 its name (a string and a length).

SQLDA ala DB2

```
struct  sqlname          /* AttributeName                    */
{
  short      length;     /* Name length [1..30]              */
  char       data[30];   /* Variable or Column name          */
};

struct  sqlvar           /* Attribute Descriptor             */
{
  short          sqltype;   /* Variable data type            */
  short          sqllen;    /* Variable data length          */
  char          *SQL_POINTER sqldata; /* data buffer          */
  short         *SQL_POINTER sqlind;  /* null indiciator      */
  struct sqlname sqlname;   /* Variable name                 */
};

struct  sqlda         /* Main SQLDA                          */
{
  char   sqldaid[8]; /* Eye catcher = 'SQLDA   '            */
  long   sqldabc;    /* SQLDA size in bytes=16+44*SQLN      */
  short  sqln;       /* Number of SQLVAR elements           */
  short  sqld;       /* Number of used SQLVAR elements      */
  struct sqlvar  sqlvar[1];  /* first SQLVAR element        */
};
```

SQLDA ala ORACLE6

```
struct SQLDA {
  long     N; /* Descriptor size in number of entries      */
  char  *V[]; /* Arr of addresses of main variables (data) */
  long   L[]; /* Arr of lengths of data buffers            */
  short  T[]; /* Arr of types of buffers                   */
  short *I[]; /* Arr of addresses of indicator vars        */
  long     F; /* Number of variables found by DESCRIBE     */
  char  *S[]; /* Arr of variable name pointers             */
  short  M[]; /* Arr of max lengths of attribute names     */
  short  C[]; /* Arr of current lengths of attribute names */
  char  *X[]; /* Arr of indicator name pointers            */
  short  Y[]; /* Arr of max lengths of ind. names          */
  short  Z[]; /* Arr of cur lengths of ind. names          */
};
```

DESCRIBE via SQLDA

A prepared statement can be *described* via an `sqlda` data structure with the SQL DESCRIBE command.

```
EXEC SQL DESCRIBE stmt INTO :sqlda
```

The result is:

- the number of result attributes

  $\Rightarrow$ 0 when not a query

- for every attribute in an answer tuple when stmt is a query:
  1. its name and length, and
  2. its type.

SQLDA and Parameter Passing

One uses an **SQLDA** descriptor to also supply parameters.

Descriptors can therefore substitute host variables as the argument of the SQL USING clause.

```
EXEC SQL EXECUTE stmt
         USING DESCRIPTOR :sqlda;

EXEC SQL OPEN cname
         USING DESCRIPTOR :sqlda;

EXEC SQL FETCH cname
         USING DESCRIPTOR :sqlda;
```

An Example Application: `adhoc.sqc`

`adhoc` is an application that executes an SQL statement provided as its argument on the command line.

Declarations:

```
#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

EXEC SQL BEGIN DECLARE SECTION;
   char  db[6] = "cs448";
   char  sqlstmt[1000];
EXEC SQL END DECLARE SECTION;

struct sqlda *select;
```

adhoc.sqc (cont.)

Start up and **prepare** the statement:

```
int main(int argc, char *argv[]) {
   int i, isnull; short type;
   printf("Sample C program : ADHOC interactive SQL\n");

   EXEC SQL WHENEVER SQLERROR  GO TO error;

   EXEC SQL CONNECT TO :db;
   printf("Connected to DB2\n");

   strncpy(sqlstmt,argv[1],1000);
   printf("Processing <%s>\n",sqlstmt);

   EXEC SQL PREPARE stmt FROM :sqlstmt;

   init_da(&select,1);

   EXEC SQL DESCRIBE stmt INTO :*select;

   i= select->sqld;
```

adhoc.sqc (cont.)

. . . its a query:

```
if (i>0) {
  printf("      ... looks like a query\n");

  /* new SQLDA to hold enough descriptors for answer */
  init_da(&select,i);

  /* get the names, types, etc... */
  EXEC SQL DESCRIBE stmt INTO :*select;

  printf("Number of select variables <%d>\n",select->sqld);
  for (i=0; i<select->sqld; i++ ) {
    printf("  variable %d <%.*s (%d%s [%d])>\n",
                  i,
                  select->sqlvar[i].sqlname.length,
                  select->sqlvar[i].sqlname.data,
                  select->sqlvar[i].sqltype,
                  ( (select->sqlvar[i].sqltype&1)==1 ?
                                          "": " not null"),
                  select->sqlvar[i].sqllen);
  }
  printf("\n");
```

adhoc.sqc (cont.)

. . . more processing for queries: prepare buffers and print a header.

```
for (i=0; i<select->sqld; i++ ) {
  select->sqlvar[i].sqldata=malloc(select->sqlvar[i].sqllen);
  select->sqlvar[i].sqlind=malloc(sizeof(short));
  *select->sqlvar[i].sqlind = 0;
};

for (i=0; i<select->sqld; i++ )
  printf("%-*.*s ",select->sqlvar[i].sqllen,
                   select->sqlvar[i].sqlname.length,
                   select->sqlvar[i].sqlname.data);
printf("\n");
```

adhoc.sqc (cont.)

. . . more processing for queries: fetch and print answers.

```
  EXEC SQL DECLARE cstmt CURSOR FOR stmt;
  EXEC SQL OPEN cstmt;
  EXEC SQL WHENEVER NOT FOUND GO TO end;
  for (;;) {
    EXEC SQL FETCH cstmt USING DESCRIPTOR :*select;
    for (i=0; i<select->sqld; i++ )
      if ( *(select->sqlvar[i].sqlind) < 0 )
        print_var("NULL", select->sqlvar[i].sqltype,
                  select->sqlvar[i].sqlname.length,
                  select->sqlvar[i].sqllen);
      else
        print_var(select->sqlvar[i].sqldata,
                  select->sqlvar[i].sqltype,
                  select->sqlvar[i].sqlname.length,
                  select->sqlvar[i].sqllen);
    printf("\n");
    };
end: printf("\n");
```

adhoc.sqc (cont.)

. . . otherwise it is a simple statement: just execute it.

```
    } else {
      printf("      ... looks like an update\n");

      EXEC SQL EXECUTE stmt;
    };

    /* and get out of here */
    EXEC SQL COMMIT;
    EXEC SQL CONNECT reset;
    exit(0);

error:
    check_error("My error",&sqlca);
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    EXEC SQL ROLLBACK;
    EXEC SQL CONNECT reset;
    exit(1);
}
```

Running `adhoc.sqc`

```
./adhoc "select * from author"

Sample C program : ADHOC interactive SQL
Connected to DB2
Processing <select * from author>
      ... looks like a query
Number of select variables <3>
  variable 0 <AID (496 not null [4])>
  variable 1 <NAME (453 [22])>
  variable 2 <URL (453 [42])>

AID NAME           URL
  1 Toman, David   http://db.uwaterloo.ca/~david
  2 Chomicki, Jan  http://cs.buffalo.edu/~chomick
  3 Saake, Gunter  NULL
```

Dynamic Embedded SQL Summary

- Given a string:
  1. if nothing known, use `DESCRIBE`;
  2. if a simple statement used once, use `EXECUTE IMMEDIATE`;
  3. otherwise use `PREPARE`.

- Given a statement handle obtained via `PREPARE`:
  1. if a simple statement used once, use `EXECUTE`;
  2. for the query otherwise, use `DECLARE CURSOR` and then process as an ordinary cursor.

Remember to supply correct host variables/sqlda for all parameter and answer tuples!

Outline

Unit 1: Dynamic Embedded SQL (in C)

Unit 2: **ODBC**

ODBC: An SQL Call Level Interface (CLI)

- An interface defined by a C library of function calls defined by Microsoft.
- For CLI, applications are developed entirely in a host language.

  $\Rightarrow$ no pre-compilation so no type checking at compile time

  $\Rightarrow$ less transparent compared to static embedded SQL
- The SQL CLI standard incorporates ODBC and X/Open standards.

Three kinds of *handles* in an ODBC program for:

1. an environment (exactly one per application thread);
2. a database engine connection; and
3. an SQL statement.

Recall that dynamic embedded SQL has statement handles only.

$\Rightarrow$ an executing thread has at most one ongoing transaction

Connect and Disconnect in ODBC

```
int main()
{
   SQLHENV   henv;
   SQLHDBC   hdbc;
   SQLRETURN rc;
   SQLCHAR   server[SQL_MAX_DSN_LENGTH + 1] = "DBCLASS";
   SQLCHAR   uid[19] = "<your uid>";
   SQLCHAR   pwd[31] = "<your password>";

   SQLAllocEnv(&henv);
   SQLAllocConnect(henv, &hdbc);

   rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
   if (rc != SQL_SUCCESS) {
          printf("Error connecting to %s\n", server); exit(1);
   } else printf("Connected to %s\n", server);

/* DO SOMETHING HERE */

   SQLDisconnect(hdbc);
   SQLFreeConnect(hdbc);
   SQLFreeEnv(henv);
}
```

Exception Handling in ODBC (errors)

- Each `SQLxxx` function returns an error code.

  $\Rightarrow$ similar to `libc` functions

  $\Rightarrow$ should be checked after each `SQLxxx` call

- Return codes:
  1. `SQL_SUCCESS`
  2. `SQL_ERROR`

- The `SQLError` function is used to obtain more details.

SQL Statements in ODBC

- `SQLAllocStmt` (allocates a statement handle)
- `SQLExecDirect` (execute)
- `SQLPrepare` (compile statement)
- `SQLExecute` (execute compiled statement)
- `SQLSetParam` (initialize a procedure parameter)
- `SQLNumResultCols` (number of result columns)
- `SQLBindCol` (“host variables” in ODBC)
- `SQLGetData` (obtaining values of result columns)
- `SQLFetch` (cursor access in ODBC)
- `SQLError` (obtains diagnostics)
- `SQLRowCount` (number of affected rows)

...

- `SQLFreeStmt` (frees a statement handle)

Parameters

1. Parameter markers:

   $\Rightarrow$ specified by `'?'` in the text of the query

   `SQLNumParams`
   `SQLBindParameter`

2. Results of queries:

   $\Rightarrow$ specified by the number of resulting columns

   `SQLNumResultsCol`
   `SQLDescribeCol`
   `SQLBindCol` or `SQLGetData`

3. Number of affected tuples (updates):

   `SQLRowCount`

Parameter (cont'd)

The parameters are *bound* to buffers by `SQLBindParameter`.

```
SQLBindParameter(stmt-handle,
                 param-nr,
                 inp/out,
                 c-type,
                 db-type,
                 db-prec,
                 db-scale,
                 val-ptr,val-len,
                 val-NULL-ptr)
```

Substitutes a value pointed to by `val-ptr` (with length `val-len`, indicator variable `val-NULL-ptr`, and C data type `c-type`) for the `param-nr`-th parameter of `stmt-handle` (using database type `db-type`).

Example

```
SQLCHAR  stmt[] = "UPDATE author SET url = ? WHERE aid = ?";

SQLINTEGER aid;
SQLCHAR    s[70];
SQLINTEGER ind;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLPrepare(hstmt, stmt, SQL_NTS);

printf"(Enter Author ID:  "); scanf("%ld",&aid);
printf"(Enter Author URL: "); scanf("%s", s);

rc = SQLBindParameter(hstmt, 1,
              SQL_PARAM_INPUT, SQL_C_CHAR,
              SQL_CHAR, 0, 0, s, 70, &ind);

rc = SQLBindParameter(hstmt, 2,
              SQL_PARAM_INPUT, SQL_C_SLONG,
              SQL_INTEGER, 0, 0, &aid, 0, NULL);

rc = SQLExecute(hstmt);
```

Late Binding of Parameters

- Executing a statement *without* binding all parameters can be accomplished by setting the last argument of `SQLBindParameter` to point to the constant `SQL_DATA_AT_EXEC`.

  $\Rightarrow$ results in an `SQL_NEED_DATA` error code

- Parameter values can be supplied when detecting this error code by repeated use of two functions.

  `SQLParamData` (to determine which argument)
  `SQLPutData` (to supply data)

- Allows updating several rows using one update statement, but supplying different values for each individual row to be updated.

Answers in OBDC

A number of functions exists to obtain output values computed by an SQL statement.

- Number of affected rows:

  `SQLRowCount`

- Answers to queries:

  `SQLBindCol` (to bind variables before execution)

  `SQLGetData` (to get values after execution)

- Obtaining the next tuple of a query evaluation:

  `SQLFetch`

  $\Rightarrow$ the result of `SQLFetch` is just a result code

A Query with `SQLBindCol`

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER   rows;
struct { SQLINTEGER ind;
         SQLCHAR    s[70];
       } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
                (SQLPOINTER)pubid.s, 8, &pubid.ind);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
                (SQLPOINTER)title.s, 70, &title.ind);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
   printf("%-8.8s %-70.70s\n", pubid.s, title.s);

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

A Query with `SQLGetData`

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER   rows;
struct { SQLINTEGER ind;
         SQLCHAR    s[70];
       } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
   rc = SQLGetData(hstmt, 1, SQL_C_CHAR,
            (SQLPOINTER) pubid.s, 8, &(pubid.ind));
   rc = SQLGetData(hstmt, 2, SQL_C_CHAR,
            (SQLPOINTER) title.s, 70, &(title.ind));
   printf("%-8.8s %-70.70s \n", pubid.s, title.s);
}

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

Column Descriptions in ODBC

There are functions to determine the number of results columns for queries, and column name and type information for each.

```
SQLNumResultCols(hstmt, &num)
SQLDescribeCol(hstmt,ColNo,
    ColNamebuf, sizeof(ColNamebuf),
    NULL,
    &sqltype,&sqlprec,&sqlscale,
    &ifNullable );
```

Example:

```
SQLINTEGER sqlprec;
SQLSMALLINT i, num, sqltype, sqlscale, nullable;
SQLCHAR name[32];

rc = SQLNumResultCols(hstmt, &num);

for (i=0; i<num; i++) {
   rc = SQLDescribeCol(hstmt,i+1,name,32,NULL,
                  &sqltype,&sqlprec,&sqlscale,&nullable);
   printf("attribute %d is %s (%d,%ld,%d,%d)\n"
               i,name,sqltype,sqlprec,sqlscale,nullable);
}
```

Transaction Management in ODBC

- The start of a new transaction happens implicitly with executable SQL commands:

  `SQLPrepare`,
  `SQLExecute`,
  `SQLExecDirect`, etc.

- A function is called to signal the end of a transaction:

  ```
  SQLTransact(henv, hdbc, <what>)
  ```

  where

  `<what>` = `SQL_COMMIT`, or
  `<what>` = `SQL_ROLLBACK`

ODBC Summary

- ODBC is at least as expressive as dynamic embedded SQL.
  $\Rightarrow$ not bound to a particular RDBMS engine
  $\Rightarrow$ single threads can access multiple RDBMS engines
- All statements are *dynamic*.
  $\Rightarrow$ no pre-compilation
  $\Rightarrow$ requires explicit binding of parameters
  (user is responsible for ensuring types match)
- An almost standard (ODBC, X/Open).
  $\Rightarrow$ supported by almost all RDBMS engines
  $\Rightarrow$ there are has 100's of functions