The Paradise Cursor and Query classes

Kristin Tufte
February 8, 1996

1 Introduction

This document describes the Paradise Cursor and Query classes, which allow the user to embed SQL queries in a C++ program. The Query class provides a simple mechanism for executing a SQL query and is intended for queries which do not have an answer such as such as insert, update, and open/close database queries. The Cursor class provides a mechanism for executing a query and iterating over the results.

The Cursor class provides the user a simple 'get tuple' and 'next' interface to the tuples in the result, allowing the user to scroll forward through the tuples with no knowledge of the shipping of tuples from the client to the server. In addition, the Cursor class encapsulates and the specific database representation of the data types and allows the user to interact with high-level data type representations. At this time, cursors are read-only.

The rest of this document is organized into three parts: a description of the Query and Cursor class interfaces; information about the cursor library (libclcsr.a) and necessary include files; and an example to demonstrate the usage of the Query and Cursor classes.

2 Using the Query and Cursor Class Interfaces

The basic steps for using the Query and Cursor classes are:

- Declare a Query/Cursor object.
- Bind a query to the Query/Cursor object.
- Issue an execution command.
- Start a scan on the query result table. (Cursor objects only.)
- Use the next and getTuple methods to retrieve result tuples. (Cursor objects only.)

The Cursor class functionality is a superset of the Query class functionality. The functionality common to the Paradise Query and Cursor classes is described below followed by the functionality specific to the Cursor class. Cursors are run at degree 3 consistency. A transaction is begun when the query is executed (via exec() method) and is committed when the cursor is reset (via reset() method) or a new query is bound to the cursor.

The methods common to the Query and Cursor classes are:

Constructor When a Query or Cursor object is declared or allocated the object automatically checks to see if the current process has a connection with the Paradise server. If no connection exists, one is created.

int bind(char* queryStr) bind associates a SQL query with a particularly Query or Cursor object; however, it does not execute the query. bind takes one argument, a character string consisting of a SQL query. The query string can be created at run-time and is typically created using a function like sprintf thus providing dynamic embedded SQL. Bind can be called multiple times on one Query or Cursor object to use that object to execute several queries. It is not necessary to call reset in between calls to bind.

int exec() exec executes the query bound to the Query or Cursor object. A query must be bound to the object, using the bind method, before exec is called. If the query executed using exec has a result, the result tuples will be placed in a temporary table in the Paradise database and a handle to that table will be returned to the Cursor/Query object. Exec does not begin a scan of the results, scans must be initiated using the startScan method.

int reset() reset resets the Query or Cursor object so that it is not bound to any query. If there was a temporary result table from execution of a prior query, the temporary table will be dropped from the database at this time. In addition, for Cursor objects, the transaction will be committed at this time.

IMPORTANT: In order to avoid errors, you must call reset on all Query and Cursor objects prior to issuing the query to close the database. See the section on Temporary Result Tables below for more information.

The Cursor-specific functions are:

int startScan() startScan initiates a scan on the result of a query. startScan must be called before any of the functions listed below are called. startScan fetches the first batch of result tuples from the server to the client and sets up the Cursor object for scanning. After startScan is called, the cursor points to the first tuple in the result.

int getTuple(baseAdt& result)** getTuple returns the tuple currently pointed to by the cursor. getTuple returns an array of pointers to Paradise ADT(Abstract Data Types) instances. All ADTs are classes which are derived from the class baseAdt. What is actually returned is an array of baseAdt pointers and each pointer points to an instance of a particular ADT. Each ADT has methods that can be invoked on it to get the value of the object or to perform manipulations on the object.

The Cursor object allocates the ADT instances. The user should just pass a baseAdt** into the getTuple function and should not allocate any space. Since the Cursor object allocates the ADT instances, the instances are only valid until one of the following occurs: reset is called on the Cursor object, the Cursor object is bound to a new query (using bind) or the Cursor object is deallocated (either is deleted or goes out of scope).

int getObj(baseAdt*& obj, AdtInfo::AttrType& type, int attrIdx) getObj retrieves one object and its type from the tuple pointed to by the cursor. The object to retrieve is specified by its position in the tuple. The ADT instance is allocated by the Cursor object.

int getObj(baseAdt*& obj, AdtInfo::AttrType& type, char* attrName) This function is the same as the previous function, except the object to retrieve is specified by attribute name.

int getField(char*& field, int& size, AdtInfo::AttrType& type, int attrIdx) getField retrieves the database representation of one object from the tuple pointed to by the cursor. The object's type is also returned. The object to retrieve is specified by its position in the tuple and the ADT instance is allocated by the Cursor object.

int getField(char*& field, int& size, AdtInfo::AttrType& type, char* attrName) This function is the same as the previous function, except the object to retrieve is specified by attribute name.

int next() next moves the cursor forward one tuple.

int numAttrs() numAttrs returns the number of attributes in the result.

int attrLen(int attrIdx) attrLen returns the length of the attribute at position attrIdx in the result.

AdtInfo::AttrType attrType(int attrIdx) attrIdx returns the type of the attribute at position attrIdx in the result.

char* attrName(int attrIdx) Returns the name of the attribute at position attrIdx in the result.

Some additional features of the Query and Cursor classes are described below. These features include error handling and error codes and temporary result table handling.

Error handling When an error occurs, Query and Cursor objects print an error message and return an error code. If no error occurs, clNOERROR, which has the value 0, is returned.

The error codes are:

clNOERROR No Error.

clQU_ERROR Error executing a query. This could be the query bound to the Query/Cursor object, or could be a scan query when scanning the result of a query, or could be a drop table query.

clCURSOR_INVALID Invalid cursor operation, scan has not been started. Returned from many of the cursor-specific functions.

clNOT_BOUND Query/Cursor object has not been bound to a query. Returned only from the exec function.

clBAD_SCAN Can not start scan due to invalid result name.

clTP_ERROR Error in tuple manager interface.

clEND_TUPLES No more tuples in result. (No error message printed.)

clDROP_ERROR Error dropping temporary result table.

clNO_MATCH Attribute name does not match any in result.

Temporary Result Tables In Paradise all query results are placed in temporary tables on the server. Results of queries are obtained by scanning the temporary tables. The temporary tables are deleted when the reset function is called or when a new bind call is made. A standard sequence of queries to a database would be first an open database query, then a sequence of selection, insert, and update queries, followed by a close database query. To avoid errors deleting the temporary tables, it is necessary to call the reset function on any Cursor and Query objects before the close database query is issued.

3 Query and Cursor Class Library

The query/cursor library is named libclcsr.a and is located in ~paradise/installed/lib. All programs written with this library should include the header file cursor.h which is located in ~paradise/installed/include. This header file includes the Cursor and Query class declarations and also includes all declarations necessary for using the Paradise ADTs. (Pointers to ADT instances are returned from the Cursor getTuple and getObj methods.)

Once a program using the Query and/or Cursor interface has been written, it should be linked with the query/cursor library. This executable then forms a Paradise client which communicates with the server via RPC. The cursor library includes all the code for the RPC communication and the Paradise ADT code so that the program using the Query/Cursor interface can use the ADT methods.

4 Example

This section contains an example program which demonstrates the usage of the Query and Cursor classes.

```
main()
{
  Query    query;  // Declare a query object, the constructor for
           // this object will create a connection with
                   // the Paradise server.

  // Open the database
  char     queryStr[512];
  sprintf(queryStr, "open db wisc1");  // Create the query string
  query.bind(queryStr); // Bind the query string to the query object
  query.exec();  // Execute the query.

  // Now run a query on the database
  Cursor   cursor;  // Declare a cursor object

  // Create the query string.
  sprintf(queryStr, "select * from relnA where attrA < 10");

  // Bind the query string, execute the query, and start a
  // scan on the result table
  cursor.bind(queryStr);
  cursor.exec();
  cursor.startScan();

  baseAdt** result; // Note that no space is allocated.
  int      ret;

  // Iterate through all the tuples in the result and print them
  do {
    cursor.getTuple(result);
```

```
    for(int i=0; i<cursor.numAttrs(); i++)
      result[i]->print(NULL); // All ADTs have a print function
    cout << endl;
    ret = cursor.next();
  }
  while(!ret);

  // If all went well, ret == clEND_TUPLES here
  if(ret != clEND_TUPLES)
     cout << "Error" << endl;

  // Reset the query and cursor objects to drop the temporary
  // tables and end the cursor's transaction.
  query.reset();
  cursor.reset();

  // Finally, close the database.
  sprintf(queryStr, "close db wisc1");
  cursor.bind(queryStr);
  cursor.exec();
}
```