Embedded SQL, Cursors and Data Structures

Mark Sitkowski C.Eng, M.I.E.E
www.designsim.com.au

This discussion is not for the faint of heart, nor those who are software-challenged so, if you don't understand SQL, or you don't write 'C', or you much prefer MS Windows, perhaps you might be better off reading about something else, like Excel, or Outlook.

The discussion centres around the extraction of huge amounts of data from a database, where 'huge' means hundreds of millions of rows. The methods described in this article, are the result of remedial measures, applied during a project, with which I was involved a while back, where the data extraction time was reduced from four days, to forty minutes.

A data mart had to be refreshed from a database, where the minimum table size was 100 million rows. Initially, this was done on a Saturday morning, with the fond hope of completion by Monday morning but, as the database grew, the starting point was moved to Friday night then, as it grew some more, users were advised to try to access the data mart on Tuesday and, when this was changed to Wednesday, we decided something had to be done differently.

An examination of the code in the stored procedures driving the refresh revealed that the data was fetched from the database by the innermost of 12 (yes, twelve) nested cursors, executing queries, some of which contained an ORDER BY.

Since the focal point of our attention is the database, from which and into which we wish to transfer extremely large quantities of data, it might be good to start by gently examining the nuts and bolts of how this is done.

When we invite our database engine to execute a SELECT statement, two things happen:

- We run our SQL statement through the database server's SQL interpreter
- It performs a read operation on the areas of disk containing the table or tables, which are the subject of our SELECT statement.

If our choice had been a DELETE, UPDATE or INSERT operation, we would have been performing a disk write.

Interpreters are not known for their lightning speed, and disk I/O is the slowest operation that a process can perform. Therefore, we can infer that we should minimise the amount of work done by the SQL, and minimise the amount of reading or writing that the database engine does to the disk. Specifically:

1. Don't do the logic in SQL
   Consider a cursor, which is an SQL construct used for extracting multiple rows from a database, executing the following statement on a table of ten million rows:
   SELECT one, two, three FROM there
   WHERE one = 1 AND two = 2 AND three = 3

This will execute very slowly, since on every row of the full table scan, which we need to do, we will have to perform three comparisons. It is far more efficient to truncate the predicate to perform only one comparison:

```
SELECT one, two, three FROM there WHERE one = 1
```

and perform the remaining two comparisons in our application:

```
while(sqlca.sqlcode == 0){
   EXEC SQL FETCH cur INTO :one, :two, :three;
   If(two != 2 || three != 3) continue;
   …….
}
```

2. Never ask the database engine to do ORDER BY
   ORDER BY entails the database engine doing a sort on your data. In order to do the sort, it has to use temporary disk space within its file system. If the database is not idle, but has other users performing operations, and the quantity of data to be sorted is huge, two things will happen:
   - You will run out of temporary space
   - The operation will take a long, long time.

   There is a Unix utility, called qsort() which can do, in a few milliseconds, what any database engine can do in a few minutes. Further, it performs an in-place sort, so we don't need huge amounts of temporary space and, far more importantly, we do no disk I/O.
   The synopsis of qsort() is:

```
void qsort((void  *)pointer_to_data,  size_t num_elements, size_t sizeof(element),
int (*cmp_function)(void *, void *);
```

   where cmp_function() is a user-supplied comparison function for doing the actual sorting. Although the definition looks quite ugly, all it means is that the function has to conform to the following rules:

   a) It accepts two elements as arguments
   b) It returns 0 if the elements are equal
   c) It returns+1 if the first argument is greater than the second
   d) It returns –1 if the first argument is less than the second.

   Now, let us assume that the cursor is executing the statement

```
SELECT one, two, three FROM there WHERE one = 1
```

   but we would ideally like it to produce an ordered list, as produced by

```
SELECT one, two, three FROM there WHERE one = 1
ORDER BY one
```

   Then, instead of using the ORDER BY, we would load the data into an array of structures, which look like

```
struct data {
    int one;
    int two;
    int three;
};
struct data datarray[100000];
```

Then our comparison function would look like this:

```
cmp_data(void *p1, void *p2)                    /* cmp_data */

{

struct data *q1, *q2;

   q1 = (struct data *)p1;
   q2 = (struct data *)p2;
   if(q1->one < q2->one) return(-1);
   else if(q1->one > q2->one) return(1);
   return(0);
}                                       /* cmp_data */
```

and qsort() would be called like: this:

```
qsort((void  *)data_p,  100000, sizeof(struct data), cmp_data);
```

If we need to ORDER BY more than one variable, we need to adapt the comparison function accordingly. For example, if our original SQL was:

```
SELECT one, two, three FROM there WHERE one = 1
ORDER BY one, two, three
```

we would need the following modification:

```
cmp_data(void *p1, void *p2)                    /* cmp_data */

{

struct data *q1, *q2;

   q1 = (struct data *)p1;
   q2 = (struct data *)p2;
   if(q1->one < q2->one) return(-1);
   else if(q1->one > q2->one) return(1);
   else if(q1->one == q2->one){
      if(q1->two < q2->two) return(-1);
      else if(q1->two > q2->two) return(1);
      else if(q1->two == q2->two){
          if(q1->three < q2->three) return(-1);
          else if(q1->three > q2->three) return(1);
          else if(q1->three == q2->three){
              return(0);
          }
      }
   }
}                                       /* cmp_data */
```

If the ‘one’ elements are equal, we make our decision based on the ‘two’ elements and, if these latter are equal, we use the ‘three’ elements.

1. Run each cursor in a separate process
   It is probable that any application program for high volume data extraction will be using more than one cursor. If this is the case, then we would prefer to run all cursors simultaneously, from separate processes.
   In theory, this will reduce the total run time of all of the cursors, to the run time of the slowest.
   In practice this depends on a number of factors:
   - Does the database support a ‘dirty read’?
     If the database locks the rows in a table, which is being accessed for any write operation, then our read operation will have to wait for the write to finish.
     Oracle will give certain concessions, if we do:
     EXEC SQL SET TRANSACTION READ ONLY;
     while Informix explicitly allows:
     EXEC SQL SET ISOLATION DIRTY READ
     and DB2 includes isolation level in the cursor definitions.
   - How many other applications are accessing the same tables?
     Obviously, if we have only one process, accessing the area of disk containing our tables, we will be able to achieve a higher read rate.

   The above notwithstanding, running multiple cursors from separate processes will always lead to a performance advantage.
   On a single-processor machine, the separate processes will each occupy a separate slot in the process table. If we have ‘n’ cursors, this will give our application ‘n’ times the CPU time of a single process.
   On a multi-processor machine, in addition to the above advantage, we will almost certainly get a processor for each cursor.

1. Use a PREPARE statement for cursors.
   To avoid duplicating the boilerplate code that creates a cursor, it is more convenient to define the SQL in a character string, and to pass it in to one function, which makes a generic cursor.

```
char *astring = “SELECT one, two, three FROM there \
                  WHERE one = 1 AND two = 2 AND three = 3”;

char *bstring = “SELECT four, five, six FROM here
                  WHERE four != 4 AND five != 5 AND six != 6”;
/*
 *      We can then call the function below, like this:
 *              cursors(astring, 1);
 *              cursors(bstring, 2);
 */

cursors(char *curstring, int which)          / * cursors */

{

   switch((pid = fork())){
     case -1:
        printf("Fork from cursors() (%d) failed\n", which);
        perror("Fork");
```

```
    exit(-1);
break;
case 0:                              /* this is the child process */
    if(setpgrp() == -1){
        printf("Warning: Cursor Child can't set pgrp\n");
    }

    /* Connect to DB */
    EXEC SQL
        CONNECT :user IDENTIFIED BY :pswd AT DB_XYZ  using :dbase;

    if(sqlca.sqlcode != 0){
        printf("PID %d:Connection refused:%s\n", getpid(),
                                      sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }
    printf("PID %d (%d):Connected to DB %s\n", getpid(), which, dbase);

    EXEC SQL AT DB_XYZ SET TRANSACTION READ ONLY;

     /* Prepare the cursor from the incoming string */

    EXEC SQL AT DB_XYZ PREPARE xcur FROM :curstring;
    if(sqlca.sqlcode != 0){
        printf("PID %d:Can't prepare cursor:%s\n", getpid(),
                                      sqlca.sqlerrm.sqlerrmc);
        EXEC SQL AT DB_XYZ  ROLLBACK WORK RELEASE;
        exit(-1);
    }

    /* declare the cursor */
    EXEC SQL AT DB_XYZ DECLARE gcur CURSOR FOR xcur;
    if(sqlca.sqlcode != 0){
        printf("Can't declare cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        EXEC SQL AT DB_XYZ ROLLBACK WORK RELEASE;
        exit(-1);
    }
    /* Now open it */
    EXEC SQL AT DB_XYZ OPEN gcur;
    if(sqlca.sqlcode != 0){
        printf("Can't open cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        EXEC SQL AT DB_XYZ ROLLBACK WORK RELEASE;
        exit(-1);
    }
     /* Now, the individual cursor-specific code */
    switch(which){
         case 1:
          while(!(sqlca.sqlcode == 1403 || sqlca.sqlcode == 100)){
               EXEC SQL AT DB_XYZ
                   FETCH gccur INTO :one, :two, :three;
          }
          break;
          case 2:
               /* fetch cursor 2 */
          break;
```

```
                case 3:
                     /* fetch cursor 3 */
                 break;
                 etc….
            }
    default:                         /* back in the parent process */
       printf("Child process %d running\n", pid);
     break;


}                                              /* cursors */
```

2. Use the Oracle array cursors for input or output

   Oracle has the facility of fetching a cursor into a host variable which is an array. The advantages of this are considerable. With one SQL request, we can fetch, not just one row, but any number, like 20000, or 30000
   Equally, we can perform an INSERT from an array host variable, and load 50000 rows, or however many we feel should form our syncpoint.

   - The array FETCH is performed like this:

   a) Declare all input host variables as arrays:

```
#define FCH 32000

struct  data{
    int one[FCH];
    int two[FCH];
    int three[FCH];
};
```

   b) Define the cursor string:

```
char *astring = "SELECT one, two, three FROM there \
                    WHERE one = 1 AND two = 2 AND three = 3";
```

   c) PREPARE, DECLARE and OPEN the cursor, as described above.

   d) FETCH the cursor into the host variables.

   This is a little more complicated than a usual FETCH, since it is unlikely that the size of our array divides exactly into the number of rows in the table. We need to know how many rows were returned, with each iteration of the cursor. This is achieved by using the sqlerrd[2] member of the sqlca structure, which holds a running total of the rows returned.

   For convenience, we define two local variables:

```
int prev_fetch;               /* the number of rows we got last time */
int current_fetch;            /* the number of rows in this fetch */
```

   We then FETCH the cursor into our array variables:

```
while(!(sqlca.sqlcode == 1403 || sqlca.sqlcode == 100)){
                EXEC SQL AT DB_STG FETCH gccur INTO :data.one,
                                                    :data.two,
                                                    :data.three;
                                         /* 32000 in one FETCH */
                if(sqlca.sqlcode != 0){
                   if(sqlca.sqlcode != -1405){  /* ignore fetched NULL */
                      EXEC SQL AT DB_XYZ  CLOSE gccur;
```

```
                    if(sqlca.sqlcode != 0){
                        printf("Can't close cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
                        EXEC SQL AT DB_XYZ  ROLLBACK WORK RELEASE;
                        exit(-1);
                    }
                    break;
                }
            }
            if(sqlca.sqlerrd[2] == 0) break; /* no data */

            /*
             * sqlca.sqlerrd[2] is a running total, so we must
             * subtract the previous total, to get the current
             * number fetched
             */
            current_fetch = sqlca.sqlerrd[2] – prev_fetch;
            prev_fetch = sqlca.sqlerrd[2];
            if(current_fetch == 0) break;
}
```

1.

This chapter will concentrate on features of the Oracle, DB2 and Informix embedded SQL implementations, which can be exploited to achieve performance gains, in the context of a multi-dimensional architecture. Where a feature is supported on one system, and not another, we will suggest possible alternative approaches, where such exist.

1. Multiple Connections

- Oracle
  Oracle permits simultaneous connections to multiple databases.
  Within the DECLARE SECTION of code, we define a symbolic name for each database, as per:

  ```
  EXEC SQL DECLARE LOCAL_DB DATABASE;
  EXEC SQL DECLARE REMOTE_DB DATABASE;
  ```

  Then, in the body of the program, we can connect to both databases:

  ```
  EXEC SQL
  CONNECT :name IDENTIFIED BY :password AT LOCAL_DB USING
  :database1

  EXEC SQL
  CONNECT :name IDENTIFIED BY :password AT REMOTE_DB USING
  :database2
  ```

  This binds the database names in the variables ‘database1’ and ‘database2’ to LOCAL_DB and REMOTE_DB, so that the latter can be used throughout the rest of the program.

The syntax for DB2 is different:
EXEC SQL
CONNECT TO :database USER :user USING :password [IN SHARE MODE]

The optional ‘IN SHARE MODE’ permits multiple transactions to occur within the one connection.

can be as simple as:
EXEC SQL
CONNECT DATABASE :database

1. In the body of the program, SQL statements are entered, with the prefix
   EXEC SQL
   Oracle differs, in providing the ability to simultaneously execute SQL on different databases. In order to achieve this, the preamble is changed to:
   EXEC SQL AT SYMBOLIC_DB_NAME

- Oracle can simultaneously connect to multiple databases from within one process. However, having done so, each EXEC SQL statement must explicitly state where to EXEC the SQL. Simultaneous connections have the obvious advantage, that we can take data from one database, process it and, immediately, deposit it in another.
- DB2 can only connect to one database at a time, from within one process, but can connect to any number, sequentially. This means that, if we need two simultaneous connections, we must create two processes, make the connections, and transfer the data between processes via an inter-process communication link.
- Informix supports multiple simultaneous connections, in a manner similar to Oracle but uses a different syntax to route the SQL statements to the various connections.
- DB2 and Informix both have the concept of a ‘Dirty Read’, meaning that parallel access, by multiple processes to a table is possible. DB2 also permits parallel INSERT and UPDATE access. These abilities permit improving the throughput on what is, traditionally, a huge bottleneck, namely, the row-locking, which occurs in database tables, when they are multiply-accessed.
- Oracle will permit a transaction to be labelled ‘Read Only’, but is rather vague as to how much parallelism it permits. In much the same way, Insert and Update transactions can have an Isolation Level associated with them, but the actual parallelism is left to the discretion of the database engine.
- Perhaps, to make up for the above shortcoming, Oracle offers an extremely powerful tool, in the form of its array cursors. These permit Select, Insert and Update operations to be performed via host variables, each of which is an array. The advantage of this approach, is that, instead of making 50000 calls to the database engine, to Insert 50000 rows, we only need to make one. Similarly, to retrieve 50000 rows, we only need a single Select statement.