Software Design For Performance Sensitive Applications

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

Introduction

When an application needs to process a large quantity of data in a very short time, elementary design techniques either fail completely, or take such a long time to produce results, that the application effectively cannot perform its function.
This paper discusses advanced software architecture which need to be employed in cases where the schoolbook approaches are simply inadequate.
It is assumed that the computer has been equipped with as much RAM is will be needed by the database and interacting processes, and that elementary precautions, such as separating the data and log files onto individual disks, have been taken.
The main discussion centres around database data retrieval and manipulation questions, with specific reference to SQL queries, embedded in 'C' program modules.
First, a few introductory facts:

- **Reading and writing data**

1. The heads on a high performance hard disk take approximately 7 to 12 ms to seek to a given block of data. This is known as the seek time.
2. The data can then be read or written at a rate of between 1 and 40 microseconds per byte. This gives a burst transfer rate of approximately 1MByte to 25 kbytes per second, assuming that the blocks making up the data are fairly contiguous, and located within the same cylinder. The reason for the vast disparity in transfer rates is because Unix relies on caching or buffering of the data in RAM which boosts transfer rates of small amounts of data over short periods, but these rates are unsustainable over long periods.
3. The access time of semiconductor memory is approximately 10ns for static memory, and 20ns for dynamic memory (due to the need to include a refresh cycle in between reads and writes)
   This gives a maximum data transfer rate of (32 bits x 100,000,000), or 400 Mbytes per second, assuming a 32 bit bus, and that the address and data bus of the CPU are capable of operating at 100MHz.

- **Databases and SQL**

A database is a collection of disk files, supervised by a process called the database engine, which is capable of executing ASCII-coded instructions in SQL. It does so by means of an interpreter, which parses the SQL, and calls the appropriate function, with the appropriate arguments within the database engine.
Other examples of interpreted languages are PHP and Ruby.

The database engine communicates with its various clients by means of TCP/IP, and listens for incoming connection requests on a well-known port (1521 in the case of Oracle). As each connection is made, the database engine either forks a child process, or starts a new thread of execution, to service the request, leaving the parent process to listen for new incoming connection requests. When the client terminates the connection, the forked process (or thread) dies.

- **Stored Procedures**

It may be seen from the above, that there is a limit on the maximum data transfer rate which can be achieved from the database.
The time taken to access any data comprises the fixed overhead of the interpreter processing time, plus the variable overhead of the disk access time. Variable, since there is no guarantee of contiguous data, and since the Unix kernel's process scheduling algorithm puts a process to sleep as soon as the process performs I/O. The process remains asleep until the disk controller signals the kernel that I/O is complete.
Stored procedures are an attempt to reduce the overhead of the parser and interpreter, by pre-processing the query within the database engine.
In DB2, stored procedures are compiled to object code, which is either dynamically linked to the engine at process start time, or dynamically linked when the procedure is invoked. The queries are performed as machine-code instructions, which call the appropriate functions in the DB2 engine.
Oracle does not support compiled stored procedures, preferring instead to pre-load the SQL into memory, and interpret it from there. This gives a marginal speed improvement over reading the queries from standard input.

- **Embedded SQL and 'C'**

All databases support embedded SQL, as defined in the ANSI standard.
Typically, a database-specific pre-processor examines the 'C' source file, and replaces the SQL statements with instructions, which load the sqlstm structure with data needed by the database engine, to interpret the query. At runtime, the process will make a TCP/IP connection with the database engine, and pass the sqlstm structure to it across the socket interface. When the query has completed, the results are sent back via the same socket.

Performance Considerations

It may be inferred from the above, that SQL queries primarily entail the searching of disk files for data. It may further be inferred, that performing such a search by the use of stored procedures, adds an additional overhead of having the logic of the query, such as conditional statements and loops, implemented via an interpreter.
Embedded SQL ('Pro*C, under Oracle) was designed to overcome this latter deficiency, and to enable the designer to take advantage of several freedoms associated with compiled code.

Consider the situation where two large tables, of the order of 1 million rows each, are joined in an SQL query, in order to retrieve data common to both.
Before the query can succeed, two nested loops have to execute, where the inner loop reads one table, while the outer loop reads the other. At each iteration, a comparison is made of the two or more items of data, which are the subject of the query.

If the comparison is of numerical data, then two to four bytes from one table are compared with the same number from the other. However, if character fields are to be compared, a virtual third loop has to run along both strings, to perform the comparison.
The run time of two nested loops is given by the relationship:

T = Touter + (Nouter * Tinner)

Where
Touter is the time for one complete iteration of the outer loop
Nouter is the number of steps the outer loop takes
Tinner is the time for one complete iteration of the inner loop

Thus, it may be seen, that for the query to run to completion, 1,000,000,000,000 disk accesses have to be made and, for each access, a comparison must be made of at least two data items. Neglecting this latter, and assuming a disk with a buffered 1 Mbyte/second data transfer rate would give a cycle time of 1 second to the inner loop, which would be repeated a million times, to give a total run time of approximately 1 million seconds, or 277 hours.

Patently, most databases can perform this kind of comparison in much less time, and this is achieved by use of indexes on the tables.
An index is either a B-tree or a hash table ('unique index') which provides a way of minimising the number of disk accesses necessary to retrieve a given piece of data, by avoiding a linear search.

Of the two techniques, a hash table is by far the fastest, since it is content-addressable – i.e the key is formed from the data and its address. This means that data may be retrieved by random access, rather than by linear search.
However, a hash table occupies as much space as the data it stores, so it can only practically store the addresses of one or two variables, meaning that the B-tree is more commonly used.

Comparisons in RAM

A better approach to performing searches on high volume data, is to perform the search in RAM, as opposed to searching the disk.

In most cases, a comparison is not performed on every column of a table, but on data specifically relevant to the query. Also, it may be seen, that speeding up the operation of either the inner or the outer loop by several orders of magnitude has a dramatic effect on the overall performance of the query.

Taking the above example, if the outer loop accessed data at ! Mbyte per second, from the disk, and the inner loop accessed data at 400Mbytes per second, from RAM, the inner loop cycle time would now be (1,000,000 x 2.5e-9) or 2.5ms, repeated 1 million times to give an overall run time of 2500 seconds, or approximately 42 minutes.

Simple extrapolation shows that performing the comparison of two arrays or linked lists will further reduce the run time to (2.5e-3 + (1e6 * 2.5e-9)) = 5ms.
Note, that these hypothetical figures are for the reading and comparison of 1 byte of data. The numbers increase dramatically as the number of bytes increase, since the comparison is performed inside the innermost loop, and is a square-law multiplier to both loop run times.

Faster Cursors

To take advantage of the above technique, the data must first be placed in memory, which necessitates one full table scan of one of the two tables, accompanied by the creation of a linked list, or a dynamically allocated array. The time to perform this latter must be added to the total run time.

The list or array is filled by the use of one of two techniques.

1. A loop is set up in ‘C’, which executes an SQL cursor until no more data is found.
2. A loop is set up in ‘C’ which executes a simple SELECT statement until no more data is found.

Both of these techniques will load data into host variables in approximately the same amount of time, but there is an enhancement, which can be applied to extracting data by use of a cursor.

Both DB2 and Oracle support the loading of cursor data into non-scalar host variables, so each variable effectively becomes an array of, for example, 32,000 elements.
Because the disk is being read in bursts, the maximum data transfer rate can, effectively be sustained for each pass of the cursor. This means that 32,000 times the data can be read in one FETCH and, unless the system is very heavily loaded, within the same time.
In other words, If we perform a query such as

SELECT a INTO :aa

where ‘aa’ is defined as ‘int a’ it will run 32000 times slower than the same query, where aa is defined as ‘int aa[32000]’.

Hash Tables
Even the dramatic improvement, achieved through reading the data in this way, still may not produce results which will be acceptable within the time window allocated to the process.

A further increase in performance may be obtained by not using a linked list, but by using a locally-generated hash table instead.

The ‘C’ programming language supports the creation of local hash tables, provided that:

- Only one exists at a time
- The table contains one pointer to a key, and one pointer to data.

The choice of key is important, and is analogous to the choice of a primary key in a database table. If the data item is not unique, collisions will occur in the retrieval process. Unlike a database table, a hash table has no constraints, and will overwrite duplicate keys.

Another caveat is implied by the word ‘pointer’. The table actually contains the address, stored as an ASCII string, of the variable originally entered. When making comparisons, it is the address which is retrieved by the key, not the data. This means that comparisons must be made in several stages:

- Search on key
- Retrieve data at address retrieved by key
- Compare with reference data
- If necessary, retrieve address of associated data.

The reward for all this effort is an extremely short access time, irrespective of the size of table.

Tuning 'C' code

Writing code for a performance sensitive application requires certain constraints be applied, which mean a departure from the pedestrian practices tolerated for mundane applications.

The most significant performance enhancement may be obtained by the in-lining of as much code as possible.
Consider the machine code added by the compiler to execute a subroutine call:

1. Calling routine places its return address on the stack
2. Calling routine places the called routine arguments on the stack, one by one.
3. Unconditional jump to called routine address
4. Called routine extracts its arguments, one by one, from the stack.
5. Called routine places all of its local variables on the stack, one by one.
6. Called routine initialises its local variables.
7. Called routine runs.
8. Called routine removes its local variables from the stack
9. Called routine removes return address from the stack
10. Called routine places its return value on the stack
11. Unconditional jump to return address
12. Calling routine reads return value from stack.

Every function call incurs this penalty, and these stack operations, although fast, add a large overhead to a program if they are in the wrong place.
Function calls are essential to properly-designed modular programs. However, in performance-sensitive applications, they should be avoided inside loops, and never, under any circumstances, should they be placed in the innermost of nested loops.

The alternative to the function call is the preprocessor macro.

```
#define open_cur(a, b) "EXEC SQL OPEN a USING :b\
                    if(sqlca.sqlcode != 0){\
                        printf("Error opening cursor %s: %s", a, sqlca.sqlerrm.sqlerrmc);\
                        exit(-1);\
                    }"
```

The above can be used in the source code as a function call:

```
 open_cur(my_cursor, my_var);
```

but, when the preprocessor runs, it replaces the open_cur() call with the expanded macro, and the compiler then generates in-line code.

Stdio library

The standard I/O library, which is included by almost every 'C' program, contains many of the commonly used string and I/O functions. Some of these are implemented as macros, and some as subroutines, and the file stdio.h explains which is which.

When it is necessary to perform a stdio function, such as strcmp(), in the innermost of several nested loops, it is worth checking whether the call is a macro, or a

subroutine. Most of these functions are trivial to implement, and the performance benefits can be significant if the function is implemented in-line.

To illustrate the simplicity of such substitution, strcmp(), which is often used in loops to perform comparisons, is implemented with the following:

char ***p1, *p2;** **/*** declared somewhere else */
char ***s1, *s2;** **/*** local strings, for comparison */

```
p1 = s1;
p2 = s2;
while(*p1 != '\0'){
    If(*p1 != *p2){
       /* do something, as they don't match */
        break;
    }
    p1++;
    p2++;
 }
```

Conclusion

When dealing with high volumes of data, traditional software design techniques fail. By adopting certain measures, however, two or three orders of magnitude of performance improvement can be achieved.