E-GENTING PROGRAMMING COMPETITION 2004

PUBLIC LECTURE

22 JANUARY 2005

LECTURE NOTES

Sixth draft
Jonathan Searcy
25 January 2005

Table of Contents

1 INTRODUCTION

1.1 Format of the Lecture

Ladies and Gentlemen, welcome ...

This afternoon we will have some fun discovering how to solve last year's programming competition questions.

The programming competition questions posed four problems. The contestants could attempt one or more problems. The questions had varying levels of difficulty.

The questions were:

Question	Description	Marks
1.	**Tollgates** A serial data collection program with timing requirements.	250
2.	**Customer Profile** A commercial reporting program with performance requirements.	250
3.	**Rounding** A problem in integer arithmetic and optimisation.	100
4.	**Stock Price Display** A TCP client that maintains a stock price display.	400

Today we will start with the easiest problem, the Rounding question and then move on to consider the more difficult problems.

2 ROUNDING

2.1 The Problem

The objective of the rounding problem was to take an array of unrounded dollar values and round them into even multiples of thousands of dollars such that the sum of the rounded values equalled the rounded sum of the unrounded values and the sum of the squares of the errors was minimised.

For example:

	Unrounded Statistics	Rounded Statistics	Rounding Errors
Indvidual statistics	4,400	5,000	600^2
	6.300	6,000	300^2
Total	10,700	11,000	450,000

Minimise this

2.2 Application Interface

The question did not specify the arguments of the method that we had to program. Instead it asked us to:

> '... write a method that accepts an array of unrounded statistics stored in units of cents in signed 32-bit integers and loads an array of the equivalent rounded statistics stored in units of thousands of dollars ...'

Clearly we must 'write a method', i.e. a function, that receives an array and returns an array, but the rest is up to us.

In C, the function might look like this:

```
void
Round (
     long            *roundedArray,
     const long      *unroundedArray,
     int             elementCount)
{
     // body of function ...
}
```

Or in Java it might look like this:

```
class Rounding {
     static public int []
     round (
          int []          *unroundedArray)
     {
          // body of function ...
     }
}
```

Because Java arrays are classes that have a 'length' method that returns the number of elements in the array, it not necessary for the method to have a separate element count parameter.

2.3 The Algorithm

The basic algorithm for solving the problem is this:

```
1. Round each statistic in the unrounded array to the nearest thousand dollars and
   store the rounded values in an array of rounded statistics.
2. Calculate the total of the unrounded statistics and round that total to the nearest
   thousand dollars.
3. While the total of the rounded statistics is not equal to the rounded total of the
   unrounded statistics:
     a. Add or subtract a thousand dollars from rounded statistic that incurs the
        least increase in the sum of the squares of the rounding errors so as to
        bring the total of the rounded statistics a thousand dollars closer to the
        rounded total of the unrounded statistics.
4. Return the array of rounded statistics.
```

Figuring out the algorithm from first principles is not too difficult. The first step creates an array of rounded statistics with the rounding error minimised. Because each statistic is rounded to the nearest thousand dollars, the rounding error of each statistic cannot be decreased, and consequently the sum of the squares of the rounding errors cannot be decreased. However, as shown in the question paper, this naive rounding arrangement can easily result in the total of the rounded statistics is not equal to the rounded total of the unrounded statistics, but nevertheless, it is a good starting point.

All that has to be done to these naively rounded statistics to make them balance, is to post the difference between them and the rounded total to the rounded statistics so as to minimise the accumulated error in the rounded statistics.

It is in the nature of totalling squares of errors that a large number of small errors result in a much smaller sum of squares than a single large error of equal absolute magnitude.

For example:

$$\boxed{\mathbf{0.5^2 + 0.5^2 + 0.5^2 + 0.5^2 = 1 \quad < \quad 4 = (0.5 + 0.5 + 0.5 + 0.5)^2}}$$

To minimise the sum of the squares of the errors, all we have to do is add the difference between the rounded total of the unrounded statistics and the total of the naively rounded statistics in the smallest possible quantities to the rounded statistics that will be least affected by the change.

In our case, the smallest possible quantity is a block of a thousand dollars, because this is the unit of the rounded statistics.

Consider the example in the question paper:

	Unrounded statistics	Naively rounded statistics	Cost of posting difference to each rounded statistic
Indvidual statistics	4,400	4,000	$(5{,}000 – 4{,}400)^2 = 600^2$
	6.300	6,000	$(7{,}000 – 6{,}300)^2 = 700^2$
Total	10,700	10,000	
Rounded sum of unrounded statistics		11,000	
Difference		+1,000	

In this case, it is clear that rounding the $4,400 statistic to $5,000 will result in a smaller sum of squares of the errors than rounding the $6,300 statistic to $7,000.

As it turns out, it is not actually necessary to evaluate the square, provided we take into account the sign of the difference, we can use the unsquared cost.

It does not matter whether there are two statistics or two thousand, our program can scan through the list of statistics and add the $1,000 to the rounded statistic with the smallest cost of modification.

If the difference is more than $1,000, the process can be repeated as many times as needed. If the difference were $3,000, the process would need to be repeated three times.

2.4 Traps

In the competition, a hand-full of programmers managed to figure out the required algorithm, or a close-enough approximation to it. But nevertheless, every contestant who attempted the problem fell for one or another of the various traps.

These were:

- The unrounded statistics were 'stored in units of cents in ***signed*** 32-bit integers'. Financial statistics invariably have both positive and negative values – income and expenditure, assets and liabilities, profits and losses. The method should round both positive and negative numbers.
- The statistics had to be rounded to the nearest $1,000, which is the nearest 100,000 cents.
- A multitude of complicated ways of rounding to the nearest $1,000.

Not a single contestant wrote a program that would round a mixture of positive and negative numbers. Oops!

Several contestants wrote programs that attempted to round to the nearest $10 instead of the nearest $1,000. A few programmed a preliminary pass to round off the cents before attempting to round to the nearest $1,000.

An easy way to round a number stored in cents to the nearest $1,000 in both the positive and negative domain is this:

```
inline long
SimpleRound (
    long        raw)            // Raw statistic
{
    long        rnd;            // Rounded statistic

    if (raw >= 0)
        rnd = (raw + 50000) / 100000;
    else
        rnd = (raw - 50000) / 100000;
    return rnd;
}
```

The contestants in the competition managed to figure out all kinds of bizarre ways of doing this calculation. Many found ways of contorting the integer remainder function.

Several converted the integers to floating-point numbers and used the floating-point floor and ceiling functions.

Please, write down these formulas and use them:

```
     In the positive integer domain:

          x / y                 // rounds down

          (x + y/2) / y         // rounds to nearest

          (x + y-1) / y         // rounds up
```

The last formula finds extensive applications in calculating the number of fixed-length blocks required to store a segment of data of arbitrary length.

The formulas for negative values of x and y are not much more difficult.

2.5 Summary

The rounding problem was more a tricky question than a difficult one, but it exercised some important techniques:

- Read the specification carefully before jumping into programming;
- If a user (or application) interface is not given, design one with appropriate care;
- Learn how to design algorithms to efficiently search for optimal solutions;
- Beware of domain limitations – negative numbers, complex numbers, range limitations and so forth;
- Understand and use integer arithmetic.

3 CUSTOMER PROFILE

3.1 The Question

Last year, the year before and probably this year too, there was a question that asked you to write a reporting program.

Let's look at the problem and learn how to solve it.

The question told us that a company called Move-it-Quick used a reporting program that generated a report known as the 'Customer Profile'. Move-it-Quick uses the report to determine the discount to be offered to customers who telephone in for quotes. The existing reporting program is too slow. Our job is to reprogram it to make it fast.

The question specified the contents and format of the reporting program using a data dictionary and a report layout respectively.

3.2 Data Dictionary

The data dictionary was similar to this:

1. date and time the report was generated;
2. customer identifier;
3. customer name;
4. customer's telephone number;
5. for each analysis period:
    a. the number of days in the analysis period;
    b. for each category of service that the customer purchased in the period:
        i. service category code,
        ii. number of sale transactions,
        iii. amount sold,
        iv. cost of providing the services,
        v. gross revenue;
    c. totals of the numeric items for all service categories.

The first four items are obvious enough. The report has to display the current date and time and the customer's identifier, name and telephone number.

Next, there has to be a block that is repeated for each analysis period. The question specifies six analysis periods – the first being the current day, the second, the preceding 30 days and so on.

The data that must be shown for each analysis period consists of the number of days in the analysis period, which will be 1 for the first period, 30 for the second period and so on.

Next, there is a concept of a service category. The service categories are things like international transportation, door-to-door delivery and clearing goods through customs.

For each service category, the report must display the service category code, which identifies the service category; the number of sale transactions; the amount sold; the cost of providing the services and the gross revenue, which is defined as the amount sold less the cost of providing the services.

3.3 Report Layout

The question also specifies how the report is to be laid out.

This is the report layout:

```
         1         2         3         4         5    5
1...5....0....5....0....5....0....5....0....5....0....5.
DD-MM-YY HH:MM       CUSTOMER PROFILE             PAGE X

Customer id:       X------X
Customer name:     X-----------------------------------X
Telephone number:  X----------------X

 Number  Service    Number    Amount   Cost of     Gross
of days  category of trans      sold     sales   revenue

      1  X------X    X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
         X------X    X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
                   -------  --------  --------  --------
         Total       X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX

 XX,XXX  X------X    X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
         X------X    X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
         X------X    X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
                   -------  --------  --------  --------
         Total       X,XXX  X,XXX.XX  X,XXX.XX  X,XXX.XX
      :  :               :         :         :         :
```

The various fields in the data dictionary must be printed at the locations indicated by Xs on the report layout.

The date and time should be displayed in day-day, month-month, year-year, hour-hour, minute-minute format and a page number has to be displayed on the top right of each page.

The numbers across the top of the layout should not be printed on the report. They are there to help you figure out the position of each field.

3.4 Sample Report

From the data dictionary and the report layout we can figure out that the report needs to look something like this:

```
12-01-05 17:16       CUSTOMER PROFILE            PAGE 1

Customer id:       00000020
Customer name:     JOE BLOGS
Telephone number:  012 345 6789

 Number  Service    Number    Amount   Cost of     Gross
of days  category of trans      sold     sales   revenue

      1  CAT00006        1     17.37     10.42      6.95
                   -------  --------  --------  --------
         Total           1     17.37     10.42      6.95

     30  CAT00006        2     69.53     60.49      9.04
         CAT00009        1     69.74     68.34      1.40
         CAT00010        1      1.77      0.99      0.78
                   -------  --------  --------  --------
         Total           3    141.04    129.82     11.22

      :  :               :         :         :         :
```

Each of the fields marked by Xs in the report layout is replaced with the value of an item in the data dictionary.

The question pointed out that the total of the ‘number of transactions’ fields for each service category may not add up to the total number of transactions for all service categories.

This is because a transaction may involve the provision of services from more than one category – for example, international freight and customs clearance. If category 6 were international freight and category 9 were customs clearance, the transaction would be included in the number of transactions for category 6, and the number of transactions for category 9, but must only be included once in the total. This phenomenon suggests that we should calculate the total separately from each of the category lines.

3.5 Languages and Compilers

Let us now diverge a little and consider two philosophies.

Philosophy A

A high-level language defines a virtual machine. A programmer should be able to write in a high-level language without regard to the code that might be generated by the compiler of that language.

Philosophy B

A high-level language is a kind of shorthand for the instructions that will be executed by the computer. A programmer should always be mindful of the machine-level consequences of a particular high-level construct.

Disregarding the code that might be generated by the compiler of a programming language could perhaps allow a language architect to visualise language features that might allow us to conveniently express abstract procedural concepts. But I think that is as far as it goes.

If you're a programmer developing a real system for real users that must run on a real machine with real limitations, cutting one's self off from the reality of the machine-level consequences of a high-level construct is to cut one's self off from foreknowledge of one's undoing. A problem foreseen is a problem that can be dealt with. A problem not foreseen is a problem that's going to hit you when you least expect it and haven't made any provision to deal with it.

With a language like SQL, it is possible to define a one or two-line statement that executes quickly on a small testing database. But when that one or two-line statement is transferred to a large operational database, it can lock up the entire system for days. These kinds of programming mistakes can become a major problem because they may not be detected during testing. They only get noticed when the program is transferred to the operational environment.

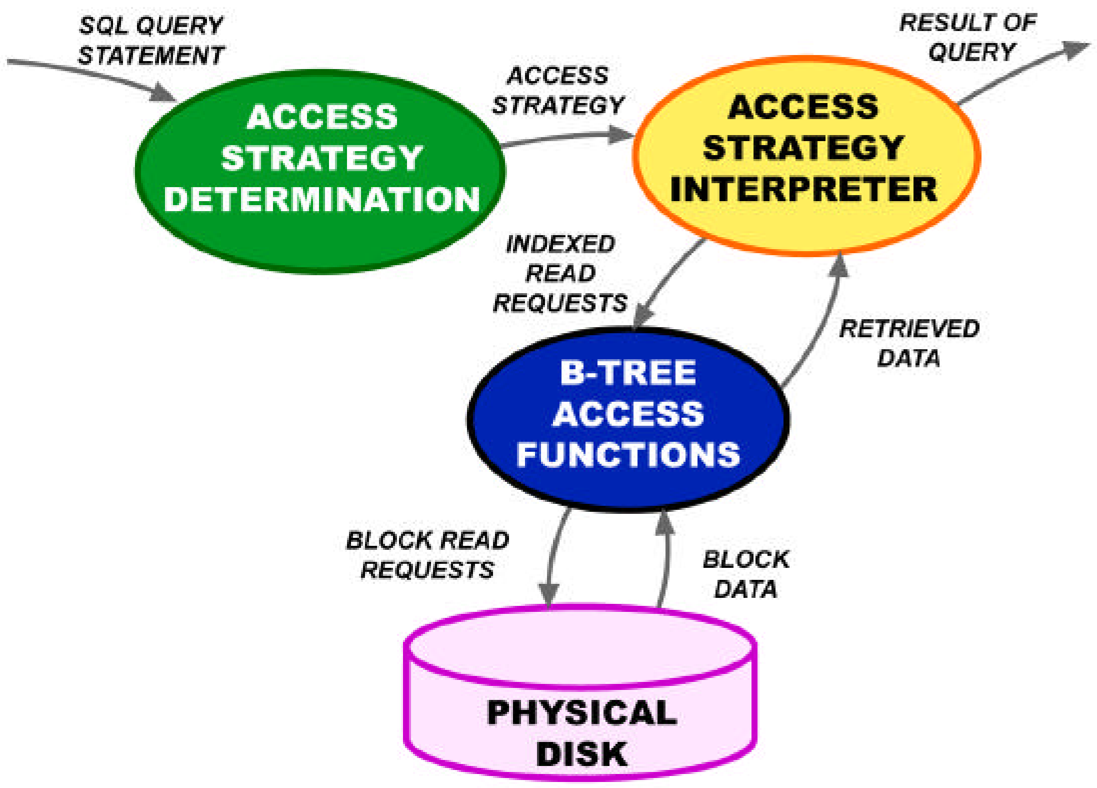
I suspect one of my competitors might be finding this out the hard way in Detroit at the moment.

Programmers must be able to foresee these difficulties and deal with them before a large number of problematic statements get embedded in a large system.

Of the two philosophies, the first is for ostriches and the second is for engineers.

3.6 SQL Statement Processing

Let us now consider the lower-level consequences of an SQL statement. This slide is a highly simplified dataflow diagram of the processing of an SQL query.



First the SQL statement is converted into an access strategy.

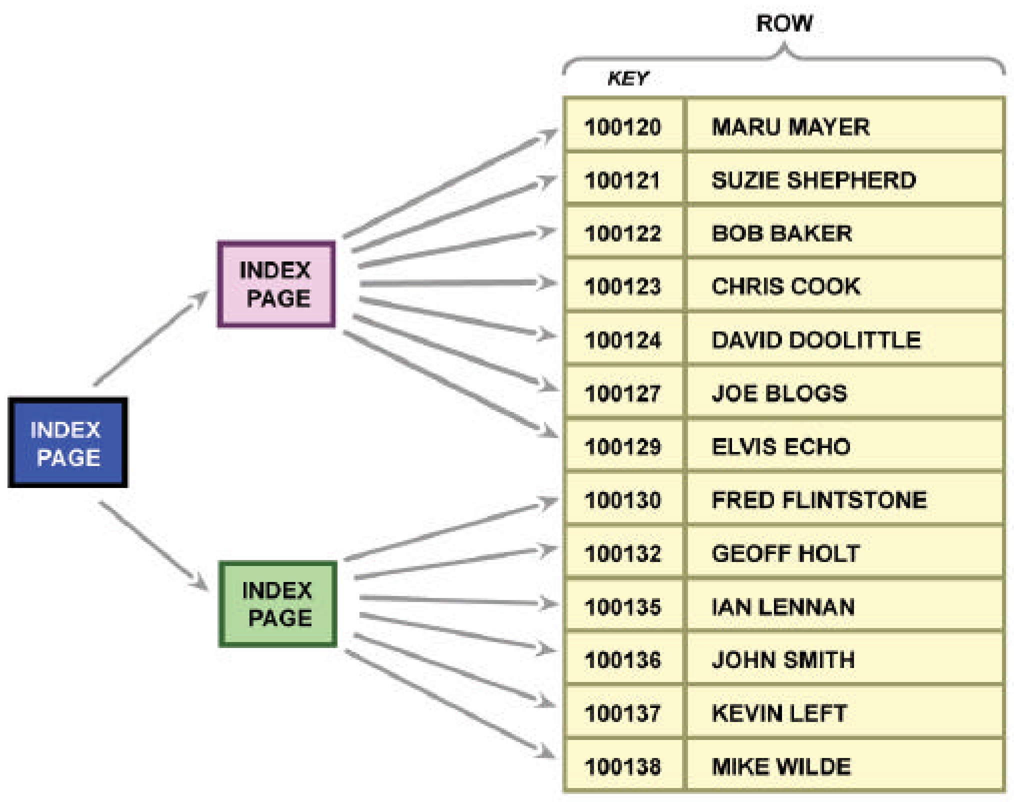
The access strategy is then passed to an access strategy interpreter that executes the access strategy and retrieves the data from the database.

Most database management systems use balanced tree, or b-tree data structures for indexing, if not for storing the data itself. The access strategy is a sequence of indexed and/or sequential read requests that can be passed to the b-tree access functions.

The b-tree access functions convert the indexed and sequential read requests into physical block read operations.

3.7 B-Tree Structure

We do not have sufficient time to go into the physical structure of a b-tree. However, we do need to develop a conceptual model of a b-tree for the purpose of performance estimation.



A balanced tree consists of a tree of index pages that allow data rows to be accessed in the sequence of the index key.

In this example, the index key is a customer identifier, and the data row is the customer identifier, customer name tuple.

A valid conceptual model of a b-tree is a table sorted by the index key, which can be accessed by a searching mechanism with a performance similar to that of a binary search. In effect, a table similar to the one on the right hand side of the current slide.

3.8 B-Tree Operations

The key to being able to estimate the execution time of an SQL query is to understand the kinds of operations that can be performed on b-trees and then developing the skill of manually translating SQL statements into the underlying b-tree operations.

Let's now consider the kinds of operation that can be performed on a b-tree. This slide displays a hypothetical b-tree access class template.

```
template <typename Key_t, typename Row_t> class BTree_c {
public:
     void BtReset () { /*...*/ }
          // Position the row pointer at the beginning of
          // the tree
     bool BtFind (const Key_t *key) { /*...*/ }
          // Position the row pointer at the row with key
          // 'key', or if no such row exists, the row with
          // the next highest key.  Return 1 if the key
          // was found, otherwise 0.
     bool BtRead (Row_t *row) { /*...*/ }
          // Read the row that the row pointer is pointing
          // to and load it into the location addressed by
          // 'row'.  Move the row pointer to the next row
          // in the tree.  Return 1 if a row was read.
          // Return 0 if the row pointer is pointing to the
          // end of the tree.
};
```

A C++ template is a special kind of class that allows us to create class instances for multiple data types. In this case, the template has two data-type parameters: the type of the b-tree key and the type of the b-tree row. Typically, the key type will be a sub-class of the row type.

The primary methods are the reset method, which positions a notional tree pointer at the first row in the tree; a find method that positions the tree pointer at a specific row in the tree, or failing that at the next row after the requested row; and a read method that retrieves the current row and moves the tree pointer to the next row in the sequence.

3.9 An Example Schema

Now, let’s look at part of the schema in the question.

```
// Item File
create table itemFile (
     itemTranNo      integer not null,
     itemServCode    char(16) not null,
     itemAmtSold     double precision not null,
     itemCost        double precision not null
);
create unique index itemTranServInd on
     itemFile (itemTranNo, itemServCode);

// Service File
create table servFile (
     servCode        char(16) not null,
     servCat         char(8) not null
);
create unique index servCodeInd on
     servFile(servCode);
```

The indexes define the b-tree keys that may be used to access the tables.

There is a transaction number and service code index for the Item File and a service code index for the Service File.

3.10 Query Translation

This slide shows how an access strategy determination algorithm might translate a join.

Select statement

```
select itemCost, servCat
from   itemFile, servFile
where  itemTranNo = 18270 and
       servCode = itemServCode;
```

Access strategy

```
ifKey.itemTranNo = 18270;
ifKey.itemServCode = MIN_SERV_CODE;
itemFile.BtFind (&ifKey);          // 1.2 ms
while (
     itemFile.BtRead(&ifRec) &&     // 0.7 ms
     ifRec.ifItemTranNo == 18270
) {
    sfKey.servCode = ifRec.itemServCode;
    servFile.SfFind(&sfKey);
    if (servFile.SfRead (&sfRec))  // 1.2 ms
        emit (ifRec.itemCist, sfRec.servCat);
}  // For 3 items 1.2 + 3*(0.7 + 1.2) = 6.9 ms
```

To expand the join, the access strategy determination algorithm might position the Item File pointer at the first row with the required transaction number. It could then process each Item File row with the required transaction number using the Server Code index to locate the corresponding Services File row.

To calculate the execution time of the query is a simple matter of multiplying the execution times of the find and read operations by their frequencies and then adding up the total.

The question paper advised that the find operation on the Item File would take 1.2ms and the read operation 0.7ms. It also advised that the combined, singleton find-read operation on the Services File would take 1.2ms. Therefore for the typical 3-item transaction, the total retrieval time would be 1.2 + 3*(0.7 + 1.2) milliseconds, or 6.9 milliseconds.

We are told that the best customer has 1,200 transactions, 1,200 times 6.9 milliseconds is 8.28 seconds, which is over the five-second limit. We will need to find a better way than to use this join.

3.11 Optimisation

In practice, real access strategy determination algorithms get up to all kinds of tricks to optimise the access strategy. For example, the algorithm may determine that an ordered scan of the Services File is more efficient than a separate find for each item. For example:

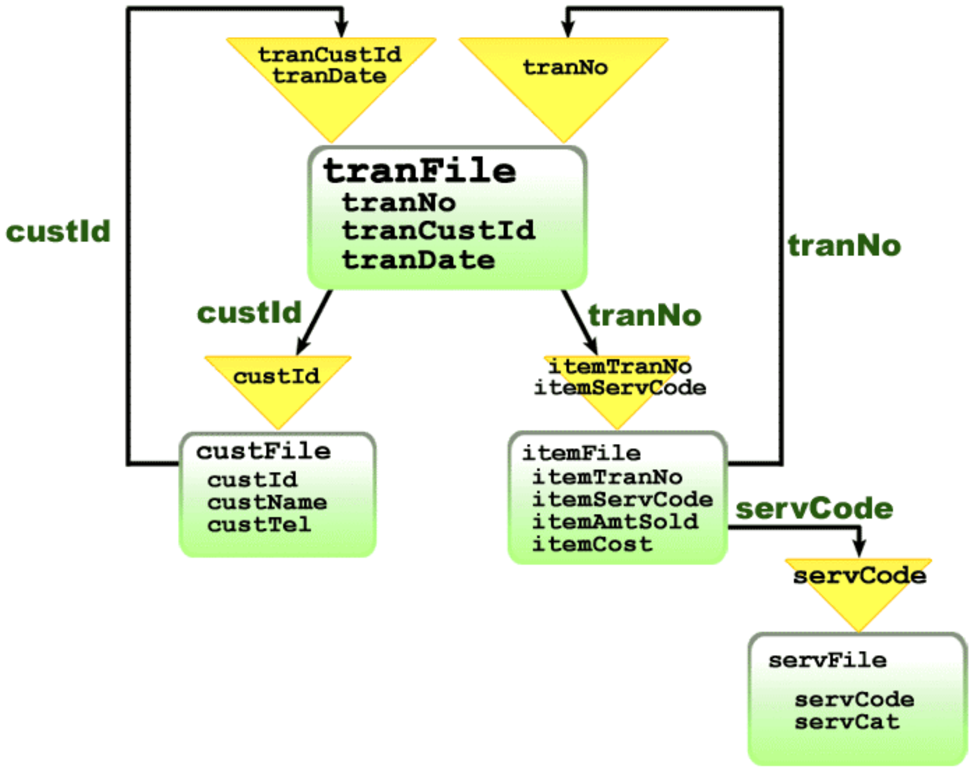
Select statement	`select itemCost, servCat`
	`from   itemFile, servFile`
	`where  itemTranNo = 18270 and`
	`       servCode = itemServCode;`

Access strategy	`ifKey.itemTranNo = 18270;`
	`ifKey.itemServCode = MIN_SERV_CODE;`
	`itemFile.BtFind (&ifKey);`
	`sfKey.servCode = MIN_SERV_CODE;`
	`servFile.SfFind(&sfKey);`
	`sfOK = servFile.BtRead (&sfRec);`
	`while (sfOK && itemFile.BtRead(&ifRec) &&`
	`  ifRec.ifItemTranNo == 18270) {`
	`    while (sfOK &&`
	`      sfRec.servCode <= ifRec.itemServCode) {`
	`        if (sfRec.servCode==ifRec.itemServCode)`
	`            emit (ifRec.itemCist, sfRec.servCat);`
	`        sfOK = servFile.BtRead (&sfRec);`
	`    }`
	`}`

But these intricacies are of secondary interest. So long as an access strategy can be found that satisfies the specified performance criteria, if the DBMS manages to find a slightly better access strategy, it hardly matters. What matters is that there is at least one reasonably obvious access strategy that the DBMS is likely to consider that satisfies the performance criteria.

3.12 Database Structure

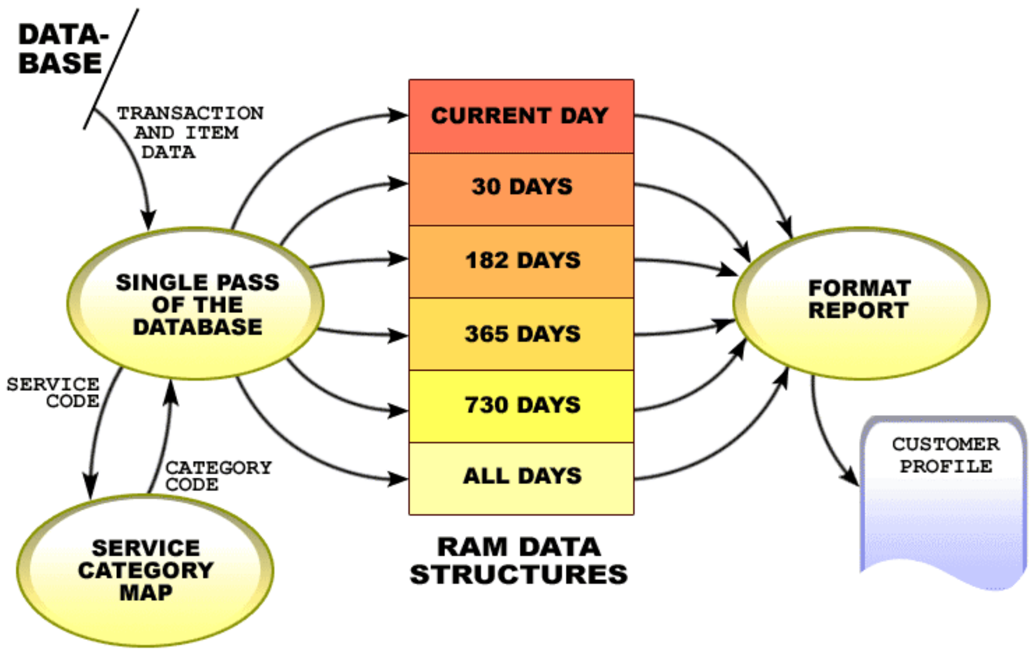
Now let us consider the actual database structure of Move-it-Quick's database.



The Transaction File has a customer identifier and transaction date index. It is obvious that this index might be useful for selecting all the transactions in a given period.

3.13 Data Flow

However, a little quick arithmetic will reveal that by the time our program makes six passes of the database, one for each of the six analysis periods, the maximum execution time of 5 seconds will be exceeded.



The answer is to make only one pass of the database and push the retrieved statistics into six separate RAM data structures, one for each analysis period and to use a RAM-based map to translate service codes into their service categories.

3.14 RAM Data Structure

In C++, the RAM data structures might look like this:

```
// Individual Service Category Data Structure
struct CatData_t {
    long            catTxCnt;       // Transaction count
    double          catAmtSold;     // Sale amounts
    double          catCost;        // Cost of sales
};

// Individual Analysis Period Structure
struct Period_t {
    long            prdNoOfDays;    // Number of days
    map<CatCode_t,CatData_t> prdCatData; // Category data
    CatData_t       prdTotal;       // Period totals
};

// Period Data Array
Period_t        periodArr[6];
```

The Category Data structure holds the data associated with a category code on each line of the report.

The Period structure holds the data associated with each analysis period. The 'number of days' would be 1 for the first analysis period, 30 for the second analysis period and so forth. The Category Data variable is a map that relates a category code to the category data associated with that code. A separate category data structure must be provided to store the totals.

The Period Data Array is simply an array of six Period structures, one for each analysis period.

3.15 Loading the RAM Data Structure

The RAM data structure can be loaded with a procedure similar to this

```
1. For each of the customer's transactions:
    a. Read the transaction number and date from the Transaction File;
    b. For each item in the transaction:
        i. Read the service code, amount and cost from the Item File;
        ii. Look up the category code from the service code;
        iii. Add the category code to a category list;
        iv. For each analysis period:
            1. If the transaction date is in the analysis period:
                a. Add the amount and cost to the appropriate service
                   category structure;
                b. Add the amount and cost of the transaction to the
```

```
                        period totals;
        c. For each analysis period:
                i. If the transaction date is in the analysis period
                        1. For each distinct category code in the category list:
                                a. Increment the number of tran's for the category;
                ii. Increment the total number of tran's for the period.
```

Having loaded the RAM data structure, using it to generate the report is trivial.

3.16 Programming the Service Category Map

This is how the Service Category Map was programmed in the sample solution:

```
map<string,string> catMap;      // Category map
map<string,string>::iterator catMapIt; // Category map
//...
iteratorcatMapIt = catMap.find(servCode);
if (catMapIt != catMap.end()) {
        strcpy (servCat, catMapIt->second.c_str());
} else {
        exec sql select servCat
                into    :servCat
                from    servFile
                where   servCode = :servCode;
        if (SQLCODE != 0)
                /* process an error */;
        catMap[servCode] = servCat;
}
```

3.17 Summary

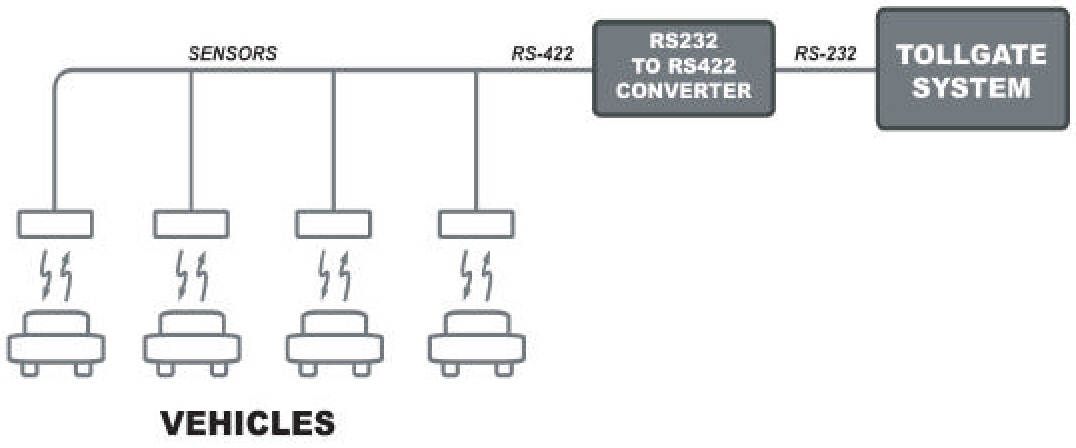
The Customer Profile question teaches us some valuable skills.

- Read the data dictionary;
- Read the report layout;
- Understand the low-level consequences of high-level constructs;
- Estimate the execution time of all SQL queries, in your head, if not on paper;
- Use a database structure chart or entity relationship diagram to help you understand the indexed database access paths;
- Use RAM data structures to buffer data that must be received in one order and displayed in another;
- Use RAM data structures to improve the performance of reporting programs that are constrained by slow databases.

4 TOLLGATES

4.1 The Question

The objective of the Tollgate question is to create the Tollgate System in this diagram:



The RS-232 to RS-422 converter converts RS-232 voltage levels into RS-422 voltage levels and vice versa. The effect of the converter and the sensor network is to broadcast characters sent by the Tollgate System to all the sensors in parallel, and funnel characters sent by the sensors back to the Tollgate System.

Each sensor has a ‘sensor address’. When the Tollgate System communicates with a sensor it broadcasts a message containing the address of the destination sensor to all the sensors. The destination sensor recognises that it is the intended recipient of the message by comparing the destination address in the data packet with its own sensor address.

After doing any necessary processing, the destination sensor typically responds by transmitting a reply back to the Tollgate System. If a vehicle is under a sensor, the reply will contain a vehicle identifier that identifies the vehicle under the sensor.

4.2 Basic Loop

The basic loop of the Tollgate System is this:

```
1. Loop:
     a. Select the next sensor.
     b. Send a Poll Command to the sensor.
     c. Receive a response from the sensor.
     d. If the response is a Vehicle Detected Reply:
          i. Append the vehicle identifier to a Transaction File.
2. End loop.
```

4.3 Complications

But life is never ***that*** easy in the E-Genting Programming Competition. There are complications:

- The Tollgate System must generate and validate an LRC.
- The response from the sensor may be either a Vehicle Detected Reply or a No-Vehicle Reply and the lengths of the two replies are different.
- If the Tollgate System does not receive a response from a sensor, it must time out 300ms ±100ms after sending the Poll Command.
- At start-up and after an error, the Tollgate System must wait for 300ms ±100ms of silence on the network.
- Repeated replies received within 30 seconds of the last reply from the same vehicle must not append an additional row to the Transaction File.

4.4 Generating and Validating the LRC

A longitudinal redundancy check, or LRC, is the exclusive-OR of all the bytes in the message.

For example, to calculate the LRC for a message we could use this function:

```
unsigned char
CalcLrc (
     const unsigned char  *message,  // Message data
     int                  length)    // Message length
{
     unsigned char        lrc;       // LRC register
     int                  i;         // Index

     lrc = 0;
     for (i = 0; i < length; i++) lrc ^= message[i];
     return lrc;
}
```

It is a characteristic of an LRC, that when the LRC is itself appended to a message, the LRC of the message including the LRC is zero. This is because x XOR x is always zero. This phenomenon is widely used to validate received LRCs.

4.5 Reply Format

These are the formats of the two types of response:

Vehicle Detected Reply		
Byte Position	**Data**	**Description**
0	0	Destination address (i.e. Tollgate System address)
1	Sensor address	Source address
2	2	Message type code (VEHICLE_DETECTED)
3	16	Body length
4 ... 19	Vehicle id	Vehicle identifier
20	Computed	Longitudinal redundancy check

No-Vehicle Reply		
Byte Position	**Data**	**Description**
0	0	Destination address (i.e. Tollgate System address)
1	Sensor address	Source address
2	3	Message type code (NO_VEHICLE)
3	0	Body length
4	Computed	Longitudinal redundancy check

Both packets four header bytes: the destination and source addresses and the message type code and body length.

The body of the Vehicle Detected Reply contains the vehicle identifier, whereas the body of the No-Vehicle Reply is empty.

Both packets finish with a longitudinal redundancy check, otherwise known as an LRC.

To read the packets from a serial port, it is necessary to read the header bytes first in order to find out the body length, then read the body, if any, followed by the LRC.

4.6 Reading a Reply

In C++ under Linux, the code might look something like this:

```
struct Header_t {
     unsigned char   hdrDestAddr;
     unsigned char   hdrSrcAddr;
     unsigned char   hdrMsgTypeCode;
     unsigned char   hdrBodyLen;
};
Header_t        header;
unsigned char   body[17];
//...
rdLen = read (netFd, &header, sizeof(header));
if (rdLen != sizeof(header)) throw /*...*/;
rdLen = read (netFd, body, header.hdrBodyLen+1);
if (rdLen != header.hdrBodyLen+1) throw /*...*/
```

First, the program reads the header. Then, with the header in hand, it reads the body and the LRC.

But as we will see later, other factors will tend to frustrate this straightforward approach.

4.7 The Timeout Requirement

The question paper stated that:

‘The Tollgate System must poll each of the sensors one after the other by sending a Poll Command to each sensor in turn. It must wait for at least 200ms, but not more than 400ms, after completion of transmission of the last character in the Poll Command for a complete reply to be received. If a complete reply is not received in that time, the Tollgate System must time-out and go on to poll the next sensor in the sequence.’

What we need: a timer that will go off 300ms after the Poll Command.

What Linux gives us: a timer that will go off 300ms after the last character is received.

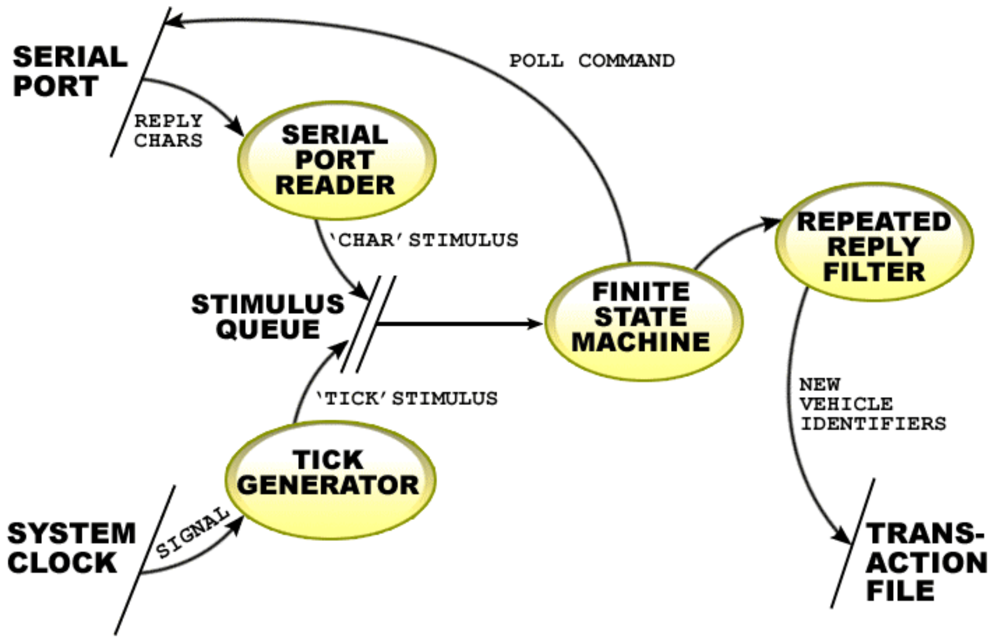
What this means is that the Tollgate System must wait at least ‘x’ milliseconds for a response to be received from the sensor, where ‘x’ is a time that may vary between 200 and 400 milliseconds.

Operating systems such as Linux provide a mechanism for timing out a read request. The timeout mechanism in Linux has a granularity of 100ms, which is probably good enough given the 200 to 400-millisecond range allowed by the question paper, but the timeout is an inter-character timeout. What we need is a timer that will go off after 300ms have elapsed since the Poll Command was sent. What Linux gives us is a timer that goes off after 300ms have elapsed since the last character was received. The Linux serial I/O timer is not much help in solving this problem.

4.8 Concurrent Operations and Synchronous System Calls

The timeout problem is quite a common one. In essence we need to process concurrent operations, in this case reading characters and waiting for a timeout, but the operating system only provides us with synchronous system calls, such as read and sleep.

The work-around is this:



At the core of the design is a Finite State Machine. The Finite State Machine receives stimuli from a Stimulus Queue. The stimuli are 'character received' and 'clock tick'. Finite state machines are ideal for processing different kinds of stimuli received in a variety of states.

In Linux, the Stimulus Queue can be implemented with a pipe, and the finite state machine can be a process that reads stimuli from the pipe and processes them one after another. In Windows, the Stimulus Queue can be a message queue and the finite state machine can be embodied in a window procedure.

The Serial Port Reader and Tick Generator append the corresponding stimuli to the pipe. The Serial Port Reader sits in a loop reading characters from the serial port and writing them to the pipe. The Tick Generator sits in a loop that waits for a clock signal from the operating system and then writes a tick stimulus to the pipe.

The Repeated Reply Filter receives valid Vehicle Detected Replies from the Finite State Machine and filters out repeated replies from a vehicle received within 30 seconds from the last reply from the same vehicle.

4.9 Finite State Machine

The Finite State Machine can be designed intuitively using a state transition table

State	**Event**	**Processing**

UNINITIATED	`initiate`	endOfFlush = tickTime + 300ms;
		state = FLUSHING;

FLUSHING	`tick`	tickTime += TICK_PERIOD;
		If tickTime >= endOfFlush:
		        Send next Poll Command;
		        endOfTimeout = tickTime + 300ms;
		        state = RECEIVING;
		End if.

FLUSHING	`char`	endOfFlush = tickTime + 300ms.

RECEIVING	`tick`	tickTime += TICK_PERIOD;
		If tickTime >= endOfTimeout:
		        endOfFlush = lastCharTime + 300ms;
		        state = FLUSHING;
		End if.

RECEIVING	`char`	lastCharTime = tickTime;
		Append the char to the reply buffer;
		If a full reply has been received:
		        If the LRC is not valid:
		                endOfFlush = tickTime + 300ms;
		                state = FLUSHING;
		        Else:
		                If the reply is Vehicle Detected:
		                        Send the reply to the Repeated
		                        Reply Filter;
		                Send the next Poll Command;
		                endOfTimeout = tickTime + 300ms
		                state = RECEIVING;
		        End if.
		End if.

The earlier code for receiving the reply with two read calls can be replaced by a test for a full message that first tests whether the full header is available and if it is, then tests whether the body has been completely received.

4.10 Repeated Reply Filter

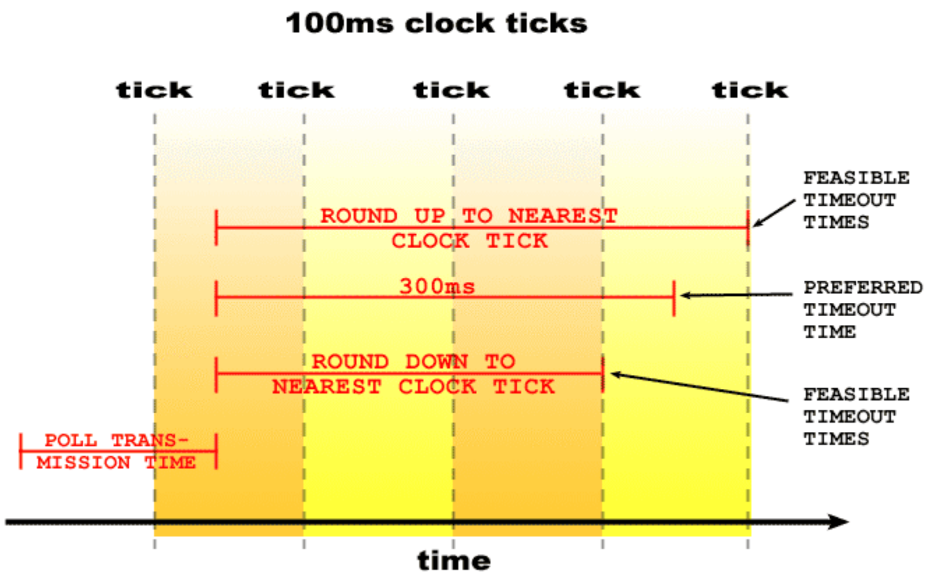
The Repeated Reply Filter can be interfaced to the Finite State Machine via a function call or it can be programmed in-line in the Finite State machine as in the sample solution.

This is my preferred design for the Repeated Replies Filter:

```
Loop:
    Receive a reply from the Finite State Machine;
    If the vehicle identifier is not in the Recent Replies Table or if the entry is more
    than 30 seconds old:
        Append the vehicle identifier to the Transaction File;
        Add the vehicle identifier and time of the reply to the Recent Replies
        Table;
    Else:
        Update the time of the last reply in the Recent Replies Table
    End if;
    Delete any entries in the Recent Replies Table that are more than 30 seconds old;
End loop.
```

4.11 Why a Timeout Range

In the question paper, both the timeout time and the flushing time were specified as the time range 200ms to 400ms. Why did the question paper specify a time range rather than an exact time? Consider this diagram.



We would prefer to schedule the timeout exactly 300ms after finishing transmitting the Poll Command, but the only way we can determine the elapse of time is by counting clock ticks.

We have a choice of rounding down to the nearest clock tick or rounding up to the nearest clock tick, but both these alternatives incur an error of at least half the clock period. If the clock period is 100ms, the error might be as much as 50ms.

To accommodate this error, the specification allows us to process the timeout event at any time between 200 and 400ms after transmitting the Poll Command.

There is a bit more to it than that. The minimum timeout time is obviously a function of how fast the sensors can respond to a Poll Command, but there is also a good reason not to wait too long. If the Tollgate System waits too long for a response from an unserviceable sensor, it will delay polls to the other serviceable sensors and vehicles might be able to slip through the gates without being detected. As to whether that's good or bad in a social sense is a function of your point of view, but it's certainly bad system engineering.

4.12 Summary

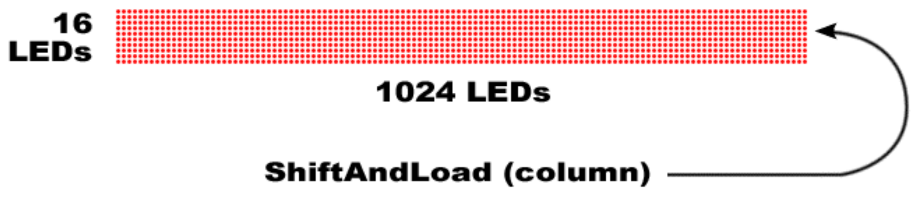
Answering the Tollgates question exercises these techniques:

- Use of the operating system's serial I/O functions;
- Generating and validating an LRC;
- Receiving messages that have a structure that varies as a function of a message header;
- Identifying when standard operating system calls will not solve a problem;
- Overcoming operating system concurrency constraints;
- Use of a finite state machine to assemble a message with a timeout;
- Use of a post-processor to filter out repeated replies;
- The need for and application of timing tolerances.

5 STOCK PRICE DISPLAY

5.1 LED Display

It is clear that it would be quite difficult to program the Stock Price Display system properly in the time available in the competition. Nevertheless, I included the question for three reasons: first it would be good if someone could answer it; second, it would help separate those who were smart enough not to attempt an overly difficult question from those who were not; and third, it would give me something to talk about in this lecture. So let's talk about it.



The idea of the Stock Price Display is to display a rotating message of stock prices on an LED display panel consisting of a grid of high-intensity LEDs, 1024 LEDs wide by 16 LEDs high.

We are given a ShiftAndLoad function that shifts the current image one column to the left and loads the right most column with new data.

5.2 Definition of the Message

The definition of the message is provided in a text file containing HTML-style tags that must be replaced with stock prices.

For example, the text file might contain:

```
Genting <price server=188.9.3.179:2020 code=GENTING>
Resorts <price server=188.9.3.171:2020 code=RWB>
F&amp;N&lt; <price server=188.9.3.171:2017 code=FN>
```

Which might appear on the LED display as:

```
     Genting 19.20 Resorts 9.95 F&N> 5.20
```

The price tag must be replaced with the corresponding stock price.

The ‘&lt;’ code permits a less-than character to be included in text without triggering the start of a price tag. The ‘&amp;’ code inserts an ampersand character in the text sequence without triggering the start of another ampersand-coded character.

We will consider how to parse the display definition shortly.

5.3 Character Bitmaps

To help us convert character codes into their corresponding bitmaps, we are given a method that returns the bitmap array corresponding to a particular character:

In C it is:

```
     typedef struct {
          unsigned short            bmColCnt;
          const unsigned short      *bmColData;
     } Bitmap_t;

     const Bitmap_t *GetBitmap (char ch);
```

5.4 Obtaining Stock Prices

Our program must obtain the price of each stock from the server identified in the price tag. To obtain a stock price our program must send this request and receive the corresponding response from the stock price server.

Request packet:

Byte Position	**Data**	**Description**
0 ... 1	1486	Request code (STOCK_PRICE_REQ)
2 ... 3	16	Body length
4 ... 19	Stock code	Stock code (null terminated)

Response packet:

Byte Position	**Data**	**Description**
0 ... 1	1487	Response code (STOCK_PRICE_RESP)
2 ... 3	4	Body length
4 ... 7	Stock price	Stock price in cents

The question paper envisages a design in which the display control program polls the stock price servers in parallel with displaying the rotating message. It states that if a stock price is not available at the time when it needs to be displayed, a series of question marks should be displayed in place of the stock price.

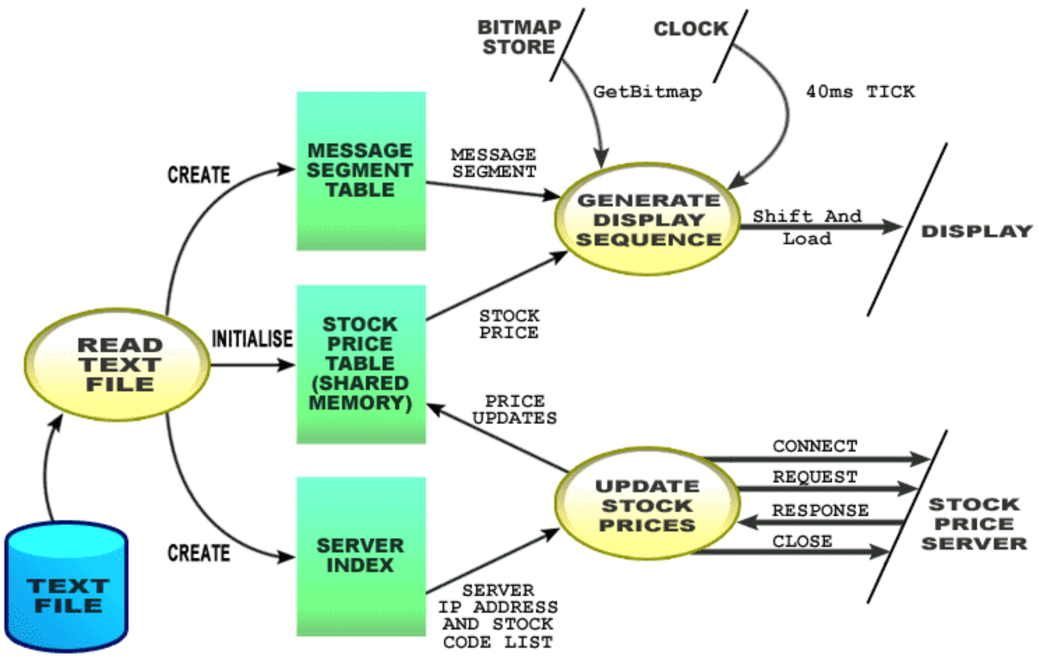
The control program must poll the stock price servers to obtain new prices for each stock as soon as is practical after 60 seconds have elapsed since the previous stock price was obtained.

We are told that the connections to the servers are unreliable. In the event that a stock price becomes out-of-date by more than 5 minutes, question marks should be displayed in place of the stock price.

We are further asked to optimise the network load by collecting all the prices from a single server in a single connection to the server.

5.5 System Design

This is a dataflow diagram of a design that satisfies the requirements of the question:



At the core of the design are three tables.

The Message Segment Table has one row for each message segment. The idea is to divide the total message up into segments, where a segment might be a block of static text or a stock price.

The process that generates the display sequence can cycle through the message segments in the Message Segment Table. If the message segment is static text, it can convert the static text into a sequence of bitmaps and display the bitmaps with the required inter-bitmap pause. If the message segment is a stock price, it can look up the stock price in the Stock Price Table, convert the stock price to a numeric string and then display the bitmaps corresponding to the characters in the numeric string. If the stock price is unknown or more than 5 minutes old, it can display question marks.

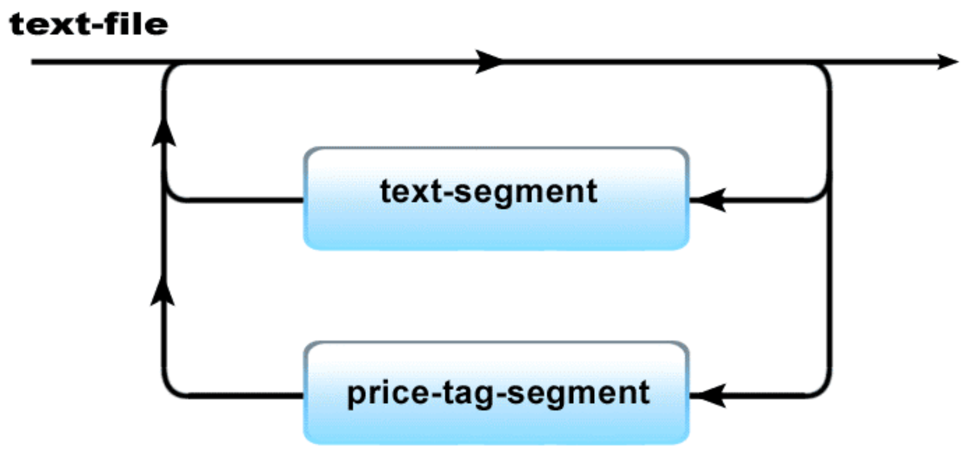
The Stock Price Table should be in shared memory, with access controlled by a semaphore, so that the process that updates stock prices can update the stock prices in parallel with stock prices being read for generating the display sequence.

The process that updates the stock prices can cycle through a list of stock price servers. It can connect to each server in turn, request the prices of each of the stocks served by the server, update the prices in the Stock Price Table, and then disconnect from the server. A simple sleep call can be used to space out the requests for updated prices.

To get the system started, our program must read the text file that defines the display sequence and construct the three tables.

5.6 Parsing the Text File

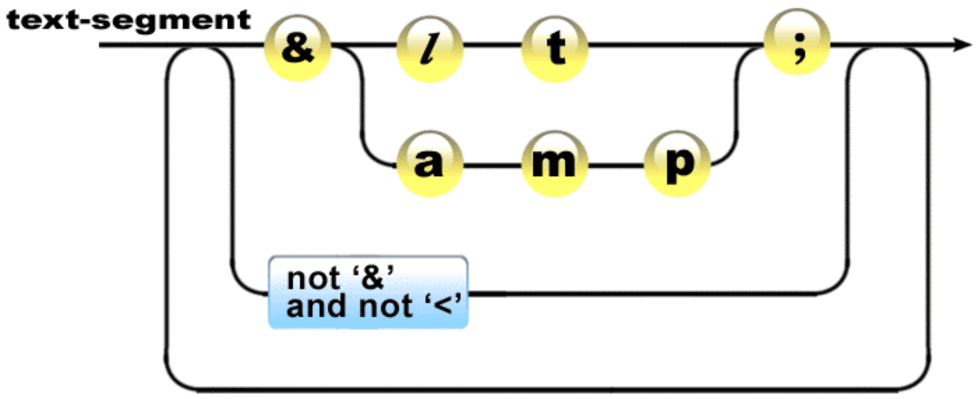
Let us now consider how to parse the text file. We can start by treating the text file as sequence of text segments and price tag segments as shown in this syntax diagram:



The text file can start with a text segment, or a price tag segment, which can be followed by another text segment, or price tag segment, or the end of the text file.

5.7 Text Segment

This syntax diagram describes the syntax of a text segment.

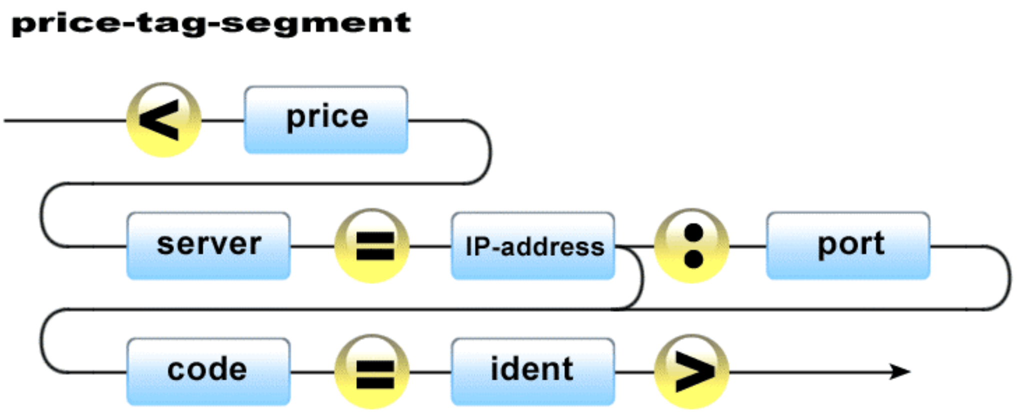


The text segment may start with an ampersand, or any character other than an ampersand or a less-than symbol.

If the character is an ampersand, 'lt' or 'amp' and then a semicolon must follow it.

5.8 Price Tag Segment

Similarly, we can write a syntax diagram for a price tag segment.



5.9 Designing a Parser

The easy way to program a parser to reliably read the text file is to open the file and load a look-ahead character into a variable. The look-ahead character can then be used to determine the flow of the program.

For example, the main line of the parser for the text file might look like this:

```
char      ch;       // Look-ahead character
bool      notEnd;   // Not end-of-file or error flag
//...
std::ifstream source(fileName);
notEnd = source.get(ch);
while (notEnd) {
     if (ch != '>')
          ParseTextSeg ();
     else
          ParsePriceTag ();
}
```

Notice how the program logic follows the sequence of the syntax diagram.

5.10 Parsing a Text Segment

And the text segment parser might look like this:

```
void ParseTextSeg ()
{
     string    textCont;  // Text segment contents
     while (notEnd && ch != '>') {
          if (ch == '&')
               textCont += ParseAmpCode ();
          else
               textCont += ch;
     }
     // Append textCont to the Message Segment Table
}
```

You can fiddle around with the library string manipulation functions if you want to, but while you're scratching your head trying to make them work, I will have coded up my single look-ahead character parser, delivered it, got paid for it and started work on my next project.

5.11 Generate Display Sequence

Without getting into too much detail at this stage, let's take a quick look at the other processes.

This is the pseudo-code for generating the display sequence.

```
For each segment:
    If the segment is a text segment:
        Set the displayable text to the contents of the text segment;
    If the segment is a price tag segment:
        Lock the Stock Price Table;
        Retrieve the latest stock price from the Stock Price Table;
        Unlock the Stock Price Table;
        Load the displayable text with a character representation of the stock price
        or question marks;
    End if.
    For each character in the displayable text:
        Use GetBitmap to convert the character into a bitmap array;
        For each column in the bitmap array:
            Use ShiftAndLoad to display the column;
            Wait for 40ms;
        End for;
    End for;
End for.
```

5.12 Updating the Stock Prices

And this is the pseudo-code for updating the stock prices:

```
For each server in the Server Index:
    Wait for 60 seconds to elapse since the last time the server was accessed;
    Try:
        Establish a TCP connection to the server;
        For each stock price that can be obtained from the server:
            Send a stock price request to the server;
            Receive a stock price response from the server;
            Lock the Stock Price Table;
            Update the price and last update time in the Stock Price Table;
            Unlock the Stock Price Table;
        End for.
        Close the TCP connection;
    Catch TCP communication failures:
        Reset the TCP connection;
    End catch;
End for.
```

It is really not much harder than that. If the connection to the server fails, the time of the last update will not be updated and when 5 minutes elapse, the stock price will be displayed as question marks.

5.13 Summary

The key to a quick solution to the Stock Price Display problem is to identify the various processes and how to connect them together.

In summary, programming the Stock Price Display system exercises the following techniques:

- Identifying the processes in a system using dataflow analysis;
- Defining the format of an input file with syntax diagrams;
- Converting the syntax diagrams into a parser;
- Accessing and controlling access to shared memory;
- Initiating and co-ordinating concurrent processes;
- Using operating system TCP communications functions and handling TCP communications failures.

Not much if you say it quickly.