EuroForth 2013

Forth Query Language (FQL) - Implementation and Experience

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
4-5 Great Western Court
Ross-on-Wye
Herefordshire
HR9 7XP
UK
Tel. +44 1989 768080
Email njn@micross.co.uk

Abstract
We will demonstrate how a hard problem in SQL becomes easy when combining SQL and Forth using FQL - "Forth Query Language".

1. Introduction

Databases are at the root of many computer applications, and almost all databases are programmed using SQL. In a previous paper (EuroForth 2006) I introduced the concept of a highly efficient method of switching between Forth and SQL, so as to achieve the maximum benefits from both. We have now had several years experience of FQL and can report on its success.

2. Features of SQL

SQL stands for "Structured Query Language" and this is possibly the most deceptive name ever devised by a computer scientist. SQL is very difficult to structure, can do other things besides queries, and is not even a language in the sense that a Forth programmer would understand. It is a "declarative" language, in that one describes the output that one needs, but not the method used for obtaining that output. It cannot be used by itself, and always needs some program written in another language for an interface.
However, for the tasks that it was originally designed for, it is extremely simple and effective. It is only when one tries to stretch it beyond its natural capabilities, that it becomes hard to manage.

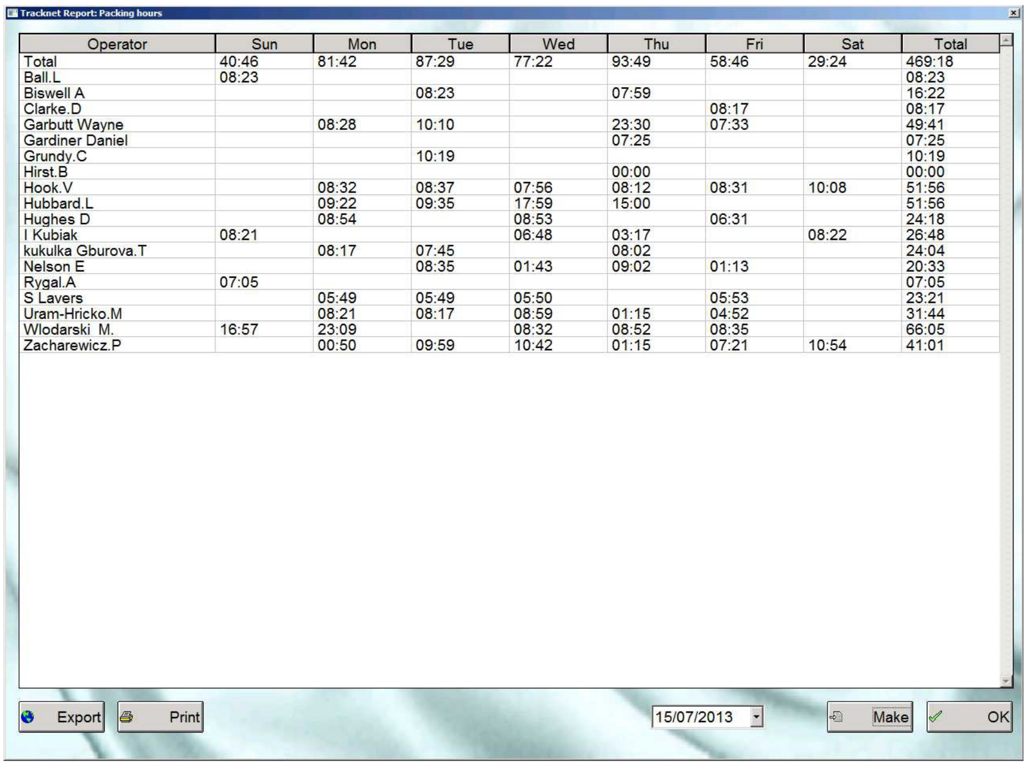
3. Features of Forth

Forth, as we all know, is the outstanding language for creating highly structured and maintainable large programs. However, it is not particularly good at creating large and complex strings, which is what is required to generate SQL queries.

4. A very hard problem in SQL

Although SQL has functional extensions (if for example IF... statements have been added), almost any problem which requires a partly functional solution, is difficult to code in tidy SQL.
Here is a example which is simple to describe, but not so simple to code.
Suppose we have a table which records the time when operators start and stop working at a particular work zone. We want to produce a report like this:

We are looking at the total hours spent at this workzone, over a period of a week, and an analysis by day, and by operator.



Operator	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Total
Total	40:46	81:42	87:29	77:22	93:49	58:46	29:24	469:18
Ball.L	08:23							08:23
Biswell A			08:23		07:59			16:22
Clarke.D						08:17		08:17
Garbutt Wayne		08:28	10:10		23:30	07:33		49:41
Gardiner Daniel					07:25			07:25
Grundy.C			10:19					10:19
Hirst.B					00:00			00:00
Hook.V		08:32	08:37	07:56	08:12	08:31	10:08	51:56
Hubbard.L		09:22	09:35	17:59	15:00			51:56
Hughes D		08:54		08:53		06:31		24:18
I Kubiak	08:21			06:48	03:17		08:22	26:48
kukulka Gburova.T		08:17	07:45		08:02			24:04
Nelson E			08:35	01:43	09:02	01:13		20:33
Rygal.A	07:05							07:05
S Lavers		05:49	05:49	05:50		05:53		23:21
Uram-Hricko.M		08:21	08:17	08:59	01:15	04:52		31:44
Wlodarski M.	16:57	23:09		08:32	08:52	08:35		66:05
Zacharewicz.P		00:50	09:59	10:42	01:15	07:21	10:54	41:01

This appears to be deceptively simple. You just subtract each operator's log on time from his log off time, and add them all up.

Unfortunately, some people work night shifts, and also move to other workzones from time to time. Even so, the algorithm is fairly easy to understand. If we just extract all log on and log off times for each day, we can pair them. Any unpaired times can then be paired with the start and end of the day, as appropriate. Then we can calculate the sum of differences.

It will be seen that while the data extraction, and the sum of differences, are very easy in a declarative language such as SQL, the pairing and above all the treatment of unpaired times, are extremely hard. On the other hand, a functional language such as Forth can easily deal with the pairing.

5. Structuring and factoring a query using FQL

If we could easily switch between SQL and Forth, we could choose which part of a problem to allocate to which language. This is what FQL achieves.

In the above example, we first extract from the table a list of times:

```
: PH-RAWQUERY { st -- parray } \ Get raw data into array
  SQL¦
  SELECT opref, opaction, TIME_TO_SEC(TIME(logsystime)),
  DAYOFWEEK(logsystime), DATEDIFF(logsystime,NOW())
  FROM operlog
  WHERE ¦    CH-LOGONOFF  ¦             \ Machine log on or off
  AND   ¦ st CH-WEEK      ¦             \ Monday to Sunday
  AND   ¦    PH-WORKZONES ¦             \ Sorting workzones
  ORDER BY opref,logsystime
  ¦SQL> CH-INSERTARRAY                  \ Insert results into array
;
```

The word SQL¦ starts to create an SQL query, until the next ¦ is encountered, when we switch back to Forth. We then stay in Forth until another ¦ is encountered, when we switch back to SQL. Finally, ¦SQL> executes the query leaving a pointer to the resulting array. That result is then placed in a temporary table in memory, ready for the pairing operations.

Note the other big advantage of FQL - we have introduced Forth structuring into SQL code, making it much easier to read. Each part of the "WHERE" clause is easy to distinguish. For example, "CH-WEEK" calculates, in Forth, the dates of the previous Sunday and the following Saturday, given any specified date, and inserts these dates into the SQL code as a "BETWEEN" clause.

6. Subquery reuse in FQL

Having carried out our pairing and sums of differences, we can then feed the result into a temporary database table. The final SQL query is still fairly complex:

```
: PH-HOURS { st -- } \ SQL query that lists operator hours at packing area by week
  ¦
  SELECT IFNULL(operator.name,'Total') AS 'Operator name',
  TIME_FORMAT((SEC_TO_TIME(sunday.minutes   *60)),('%H:%i')) AS Sun,
  TIME_FORMAT((SEC_TO_TIME(monday.minutes   *60)),('%H:%i')) AS Mon,
  TIME_FORMAT((SEC_TO_TIME(tuesday.minutes  *60)),('%H:%i')) AS Tue,
  TIME_FORMAT((SEC_TO_TIME(wednesday.minutes*60)),('%H:%i')) AS Wed,
  TIME_FORMAT((SEC_TO_TIME(thursday.minutes *60)),('%H:%i')) AS Thu,
  TIME_FORMAT((SEC_TO_TIME(friday.minutes   *60)),('%H:%i')) AS Fri,
  TIME_FORMAT((SEC_TO_TIME(saturday.minutes *60)),('%H:%i')) AS Sat,
  TIME_FORMAT((SEC_TO_TIME(total.minutes    *60)),('%H:%i')) AS Tot
  FROM      ¦ st PH-OPERATORS ¦ AS employee
  LEFT JOIN ¦  0 CH-MINUTES   ¦ AS sunday    ON employee.opref = sunday.opref
  LEFT JOIN ¦  1 CH-MINUTES   ¦ AS monday    ON employee.opref = monday.opref
  LEFT JOIN ¦  2 CH-MINUTES   ¦ AS tuesday   ON employee.opref = tuesday.opref
  LEFT JOIN ¦  3 CH-MINUTES   ¦ AS wednesday ON employee.opref = wednesday.opref
  LEFT JOIN ¦  4 CH-MINUTES   ¦ AS thursday  ON employee.opref = thursday.opref
  LEFT JOIN ¦  5 CH-MINUTES   ¦ AS friday    ON employee.opref = friday.opref
  LEFT JOIN ¦  6 CH-MINUTES   ¦ AS saturday  ON employee.opref = saturday.opref
  LEFT JOIN ¦ -1 CH-MINUTES   ¦ AS total     ON employee.opref = total.opref
  LEFT JOIN operator
  ON employee.opref = operator.opref
  ORDER BY operator.name
  ¦
;
```

This illustrates a further useful feature of FQL. "CH-MINUTES" generates an SQL sub-query which is re-used multiple times. The complete query is in fact rather long and it is not easy to see the structure when written out in full. Expressing it in Forth makes the structure clear.

"CH-MINUTES" takes a parameter which generates a slightly different sub-query each time.

```
: CH-MINUTES { pday -- } \ Inserts the time subquery
  ¦
  ( SELECT IFNULL(opref,0) as opref, SUM(pmins) AS minutes
    FROM  sortlog
    WHERE ¦ pday CH-DAY ¦
    GROUP BY opref WITH ROLLUP
  )
  ¦
;
```

Within this word, "CH-DAY" either introduces a "WHERE" clause for each day, or a "WHERE 1=1" type clause, for the totals.

```
: CH-DAY { pday -- } \ Inserts day clause, 0=select all
  ¦ (pday= ¦ pday -1 <> IF
      pday ZFORMAT
    ELSE
      Z"" pday"                  \ Select all gives pday=pday
    THEN >SQL ¦ ) ¦
;
```

Note that we could have introduced two Forth DO...LOOPs for the days of week, which would have made the code more compact. However, I think the readability is probably better without the loops.

7. Making FQL thread safe

Because SQL queries often take an appreciable time to execute, they are usually carried out in separate threads of execution. Furthermore, in an automation application, such as our flagship program "Tracknet", there may be many database operations taking place simultaneously, in different threads. Making FQL thread safe is simply a matter of assigning a private SQL connection handle and query string pad, for each thread.

```
STRUCT _TASKPRIVS
  PADSIZE           FIELD  .OWNPAD2      \ PAD2 and PAD3 for that thread
  PADSIZE           FIELD  .OWNPAD3
  MAXQUERYSIZE      FIELD  .OWNSQLPAD    \ SQL pad for thread
  CELL              FIELD  .OWNHSQL      \ Handle of SQL connection
  MAX_PATH          FIELD  .OWNCURRPATH  \ Current directory for that thread
  CELL              FIELD  .VAHDTASK     \ AHD task
  CELL              FIELD  .VAHDPROG     \ AHD progress
END-STRUCT
```

```
: SQLPAD ( ---addr ) \ SQL pad owned by a thread
  GET-TASK-PRIVATE .OWNSQLPAD
;

: HTHREADSQL ( ---addr ) \ Handle to SQL connection owned by a thread
  GET-TASK-PRIVATE .OWNHSQL
;
```

8. Comparison of code size

We noted the code lengths, when we converted a particular query from an older technique (see Swialowski, EuroForth 2006) to FQL. The old code required 136 lines, despite the lines being longer and difficult to read. The FQL solution required 89 lines, despite the lines being deliberately shortened to improve reliability. Typically, the old solution required approximately 50% more code than FQL.

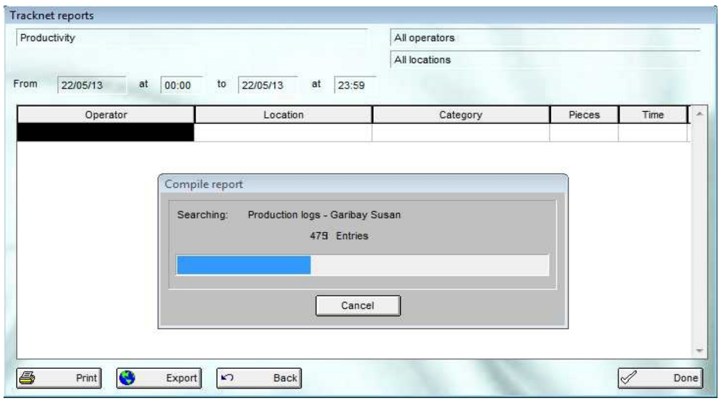
9. Comparison of performance

Very significant performance improvements can sometimes be achieved by using FQL instead of pure SQL. The most dramatic improvements are observed when the SQL code contained extensive use of procedural "enhancements" for which SQL was never originally designed.

Extreme example: Operator efficiency query.

Pure SQL	86s
FQL	4.5s

A further advantage is that, because the SQL is broken up into lots of smaller queries, it is possible to support a plausible "progress" bar and cancel button, for time consuming operations.



Tracknet reports

Productivity

All operators

All locations

From 22/05/13 at 00:00 to 22/05/13 at 23:59

Operator	Location	Category	Pieces	Time

Compile report

Searching: Production logs - Garibay Susan

479 Entries

10. Conclusion

By combining the best features of both SQL and Forth, FQL creates code that is faster, more compact, more readable and easier to maintain.

NJN
September 2013

References:

1. The Nearly Invisible Database or ForthQL
    N.J. Nelson, EuroForth 2006
2. Database access for illiterate programmers
    K.B.Swiatlowski, EuroForth 2006