

## Re: Opinions on approach, please...

---

*Source:* <http://coding.derkeiler.com/Archive/Cobol/comp.lang.cobol/2008-05/msg00512.html>

---

- *From:* "Pete Dashwood" <[dashwood@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:dashwood@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Mon, 26 May 2008 10:55:57 +1200
- 

"Frederico Fonseca" <[real-email-in-msg-spam@xxxxxxxxxx](mailto:real-email-in-msg-spam@xxxxxxxxxx)> wrote in message [news:n71j34tmrobg3jias4akjsg50pjl0es1g@xxxxxxxxxx](mailto:news:n71j34tmrobg3jias4akjsg50pjl0es1g@xxxxxxxxxx)

On Sun, 25 May 2008 14:41:09 +1200, "Pete Dashwood" <[dashwood@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:dashwood@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx)> wrote:

I'm currently looking at automated code conversion for migrating COBOL applications that use Indexed files to RDB.

(Data conversion is pretty easy and has been established for some time now.)

Code conversion is much more tricky.

Conceptually, I want to add a Data Access layer to the existing COBOL, which has no such layer. The code has indexed I-O scattered right through it as was usually the case for COBOL.

I plan to generate from a template, a module (actually a Class...) that will do all of the possible I-O for a given indexed file, except it will do it against a table set on a RDB. (A "table set" in this context is defined as a BASE table with any attached tables. Any COBOL OCCURS clauses in the original record definition cause an attached table to be generated, in line with 2NF, when the data is converted.)

The maintenance "module" will be tailored for each table set, and it will be an OO Class. This allows multiple users to access simultaneously, using their own instance of it. I call it MOST (Maintenance Object Server Template). This code is being generated as a COM server because it will be used from a COBOL and (later) from a C# environment (where it runs as unmanaged code).

Re: Opinions on approach, please...

I have a current Migration Toolset which is being updated to automate the task of replacing the ISAM access in the COBOL source with INVOKES of equivalent MOST object methods.

Converting the existing programs then comes down to mainly replacing indexed access with the invocation of a MOST object method, through an interface.

Here's the current definition of the interface block:

```
001711 01 MOST-interface-block.
001712 12 MOST-reserved. *> Do NOT change these fields
001713 *> once this block has been
001714 *> returned by MOST
001715 15 MOST-SQLSTATE pic x(5).
001716 15 MOST-SQLMsg pic x(512).
001717 15 MOST-file-status pic xx.
001718 15 MOST-access-mode pic x.
001719 88 MOST-sequential value '0'.
001720 88 MOST-random value '1'.
001721 88 MOST-dynamic value '2'.
001722 15 MOST-open-mode pic x.
001723 88 MOST-input value '0'.
001724 88 MOST-output value '1'.
001725 88 MOST-I-O value '2'.
001726 88 MOST-extend value '3'.
001727 15 MOST-action pic x.
001728 88 MOST-read value '0'.
001729 88 MOST-write value '1'.
001730 88 MOST-rewrite value '2'.
001731 88 MOST-start value '3'.
001732 88 MOST-delete value '4'.
001733 15 MOST-start-condition.
001734 88 MOST-relation pic XX.
001735 88 MOST-rel-data pic x(100).
001736 15 MOST-instance-stamp pic x(8). *> Do not pass this
block
```

snip...

Pete.

What you are doing is perfectly achievable. I have it done on the current application I am working with, although not using OO COBOL.

Good! Thanks for the encouragement, Frederico :-)

Re: Opinions on approach, please...

Main point I highlight immediately is that by using SQL you will now start using COMMIT/ROLLBACK to finalize your transactions. Although it seems "easy" this has an immediate effect on where you code your commits/rollback.

Yes, that's right. I plan to issue a COMMIT for each INSERT, UPDATE, and DELETE. The processing will be transactional and each update will be committed as it is made.

This will be done within the MOST object.

It shouldn't be a problem.

With traditional COBOL (there is one exception with ACUCOBOL), once the records are (re)written to the indexed file, there is no way to reverse it. An no log is kept of it. With RDBMS every update to a table is logged, and this has a HUGE impact in performance. In some engines it will be impossible for you to update more than 32k records without a commit in-between.

That's why I dn't plan on doing that... :-)

This is something you will need to consider in your application design.

All fine so far, but lets examine the following situation.

```
Program reads file A in a loop.  
open input-output file_a  
open input-output file_b  
open input-output file_c
```

This will involve 3 interfaces and 3 MOST objects, one for each file.

```
for each record in file_A  
update file_B
```

That would be a REWRITE, or WRITE in the current program. It becomes an UPDATE or INSERT in the MOST object for file B.

```
update file_C
```

Re: Opinions on approach, please...

Re: Opinions on approach, please...

That would be a REWRITE, or WRITE in the current program. It becomes an UPDATE or INSERT in the MOST object for file C.

```
update file_A (mark as processed)
```

That would be a REWRITE, or WRITE in the current program. It becomes an UPDATE or INSERT in the MOST object for file A.

EACH of these updates would be committed to the database as it occurs, exactly as happens in the current programs using indexed files.

I understand the transactional nature of your "mark as processed" comment, but that ISN'T what happens with their current system.

If you can't do a transactional COMMIT with ISAM/VSAM then you won't be able to with the system running against a DB. That seems fair to me.

```
next for
```

```
close file_a, file_b, file_c
```

Once you start adding commits into your program, a logical place to put a commit would be after the update of A.

Yes, but now we are letting applicational requirements affect the development of the software. I simply can't do that. It has to work for ANY application. A general solution.

Translating the above to SQL it would be (approximately) as follows.  
This assuming pure COBOL without stored procedures.

Yes, I haven't actually considered using stored procedures, although it could be a possibility. I'd need to generate and apply them "on the fly". It is not an attractive proposition :-). Obviously, it would be a last resort.

working storage.

(Only requirement is that it is before any reference to the cursor on the code, I advise you to do cursor definitions on working storage).

The cursors will be generated into the MOST module and have to be general purpose. I don't want cursor code placed into the application programs being converted. In fact, I don't want to see ANY SQL code in them. All we should see is the invocation of an interface. In that way the actual data

Re: Opinions on approach, please...

Re: Opinions on approach, please...

access using SQL is abstracted into a separate layer (data access tier). This has major implications further down the line when the app. code is refactored to C# and they move off COBOL for good.

```
exec sql
declare cursor file-a
for
select ...
from db_filea
where where_conditions
for update
end-exec

procedure ..
...
exec sql open file-a end-exec.
check sqlcode
...
file-a-cursor-read

exec sql
fetch file-a into ....
end-exec.
if sqlcode = 0
exec sql
update db_file_b set ... where (where_conditions)
end exec
check sqlcode
exec sql
update db_file_c set ... where (where_conditions)
end exec
check sqlcode
exec sql
update db_file_a set ... where current of file_a
end exec
check sqlcode
if sqlcode = 0
exec sql commit end-exec
end-if
end-if

exec sql close file-a end-exec.
```

Yes, I take your point. This is application processing, though.

Main differences from pre-post sql changes.

Re: Opinions on approach, please...

Re: Opinions on approach, please...

- 1– file b and file c are no longer opened within program.
- 2– depending on how it is coded, they are not read on program either, only updated.

All fine if it wasnt for a particular problem.

The way I defined the cursor when the commit is done, the cursor will be closed.

There are 2 ways to solve this issue. a bad (very bad), and the good way.

Bad way – after the commit open the cursor again, and loop through the cursor until you are positioned on the correct record (if required).

Correct way.  
define your cursor in a way that it will not be affected by the commit/rollback.

in the case of DB2 cursor would be defined as

```
declare cursor file—a WITH HOLD
for
select ...
from db_filea
where where_conditions
for update
end-exec
```

Yes, I encountered this problem years ago and solved it using WITH HOLD. DB2 is an excellent RDBMS. (But most of them are pretty good...)

Regarding

Have I missed an entirely better way of handling skip-sequential processing?  
Can I do it without a cursor?

Answer is no. it has to be a cursor in 99% of situations.

Thank you. I couldn't see another way, but it is good to have someone else confirm it.

Re: Opinions on approach, please...

Re: Opinions on approach, please...

Exceptions

is if your program is doing a read next, followed by an update of same record. On this cases the code should be replaced by the corresponding SQL statement to do a SET based processing. Will imply code changes for sure.

Hmmm... I hope not :-)

I see it working as:

Current code (application) + MOST action.

```
=====
START ... + DECLARE,OPEN,FETCH Cursor
READ NEXT + return held record, FETCH
REWRITE + UPDATE using key of record
+ (This is separate and independent
of the
+ cursor...)
..... +
EOF action + CLOSE cursor
=====
```

(I expect the text only nature of posts here will scramble the table above, but hopefully, you'll see what is happening...)

For cases where the cursor is the only way in order to improve performance, you should try and add all the conditions that cause a record to be include/excluded from further processing on the SQL statement.

This would require you either to use dynamic SQL (which may not be advisable depending in SEVERAL factors), or to code SQL statements within each program.

I think I can do it WITHOUT using dynamic SQL to PREPARE the cursor, but I don't really know yet.

Here's my reasoning:

1. A relational condition (as we might encounter in a START, for instance) has a known format.
2. It can be parsed into operand1...relation...operand2 (parsing can deal with compound conditions)
3. Operand1 is a field in the record and will be either a primary or alternate key.
4. MOST has metadata that tells it which fields in the record are key fields, so it can find operand1 OK.
5. The relation (<, >, <=,=>,=) can be passed through the interface.
6. The data in operand 2 can be passed through the interface.

Re: Opinions on approach, please...

Re: Opinions on approach, please...

It would be possible for MOST to contain a cursor structure that handles each of the possible relation conditions and then simply "slot in" the data passed through the interface. However, that's pretty clumsy and also inflexible. I guess it will come down to dynamic SQL, built by MOST at run time.

My very strong advice to you/your employer/client is to go ahead with replacing the index access with your IO modules, but to proceed with an extra exercise afterwards whereby you replace it with ESQL on the extract/report programs (normally known in other environments as BATCH programs without updates).

Reporting has been removed from the equation entirely. We are using .NET and Simulsoft with C#. This means reports are designed using a visual interface. Given an existing report sample, you can rebuild it in Simulsoft in an hour, without any need for COBOL, counting fillers in report lines. handling totalling, control breaks, page numbers, or even Master/Detail reporting. This is a very impressive package. It works as an extension to Visual Studio and generates C# code. I was blown away by the capabilities of it.

The only code to be converted now is true application code. Some of this is in PowerCOBOL and I'm looking forward to that...:-) (Another story...)

This new change would create all required cursors with ALL the joins required by all the files extracted within the program. DO NOT, I repeat, DO NOT replace each individual call to your IO module with its individual cursor. Performance is extremely poor if you do not do this. These cursors would obviously only retrieve the columns required by the program, which your IO modules can not do if you wish them to replace all the update programs you have at the moment.

Thank you. That is very sound advice and I appreciate it. Fortunately, Stimulsoft obviates the need for this processing.

Regarding START file ...

Main thing to consider here is that many programs use a START INVALID KEY to determine if AT LEAST ONE record meet the condition. While this can be done in SQL, it is not normally advisable as there is a performance hit on it.

Re: Opinions on approach, please...

Re: Opinions on approach, please...

At this stage, I'm more concerned with automating code conversion than I am with performance. However, performance is certainly a consideration once we have the system running against the RDB.

You can code the start as a single select, retrieving only 1 row. How you code this depends on the RMDB used. SQL Server has TOP 1, DB2 uses FOR FETCH FIRST 1 ROW ONLY. Others have different ways. Problem with this is that depending on your table design, your key access and other aspects of it, the sql may need to process ALL matching rows before returning you a single row. On such cases this should be left to the cursor itself.

I plan to use: DECLARE, OPEN, FETCH as described in the table above. The issue at the moment is whether the DECLARE needs to be dynamic or not.

Just a few thoughts. If you post more target questions, or if I remember any more important points I will post here again.

Thank you, Frederico.

You raised some very good points and all of it gets me thinking.

Cheers,

Pete.

—

"I used to write COBOL...now I can do anything."

.