

Don't Use Cursors!

– or –

Why You Maybe Should Use a Cursor After All

Erland Sommarskog
Data Platform MVP



Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

<http://www.sommarskog.se>

esquel@sommarskog.se



GOLD SPONSORS



Volvo Group IT

SILVER SPONSORS



BRONZE SPONSOR



STRATEGIC PARTNER



Agenda

- Why loops are slow.
- Why set-based statements generally are faster.
- Why we sometimes need loops and how to write them.

Looking at a Loop

[01_poor-mans-cursor.sql](#)

- Make sure that your iteration column is indexed.
- Generally in a loop, you want seeks to reduce overhead as much as possible.
- Even if the table is relatively small – many small scans add up!
- Lee Tudor's `sp_sqltrace` is a good tool for finding bottlenecks in loops.
 - <http://www.sommarskog.se/sqlutil/sqltrace.html>

Advanced:

Reduce Transaction Overhead

[02_using_transactions.sql](#)

- With auto-commit, SQL Server has to wait after every UPDATE for the transaction log to be hardend.
- This can be mitigated by adding a transaction and committing after every 1000th iteration or so.
- If the entire loop is a single transaction, execution time per iteration may start to grow.

Reduce Transaction Overhead, Cont'd

- Ultimately, business constraints define the transaction scope.
 - If all or nothing => Single transaction.
 - If single failure must not affect the rest => No transaction.
- In practice, this technique is of limited use.

Set-Based Logic

[03_set-based.sql](#)

- These solutions are faster!
- In this particular case, the indexes on the temp table and Discounts are not critical.
- Since it is a single UPDATE statement, we get the reduced transaction overhead for free.
- Does not have to be one monolithic statement – it may be easier and better to use intermediate temp tables.

Set-Based – General Observations

- A set-based solution is generally faster than the most well-written loop.
- You tell the optimizer what result you want (declarative), not how to compute it (imperative).
- Code is more concise.
- Loops are prone to errors:
 - Failing to initiate loop variables on each iteration.
 - Incorrect termination conditions.
- Writing set-based code may be outside your current comfort zone – you need to extend it!

[04_loopvarinit.sql](#)

Set-Based Magic?

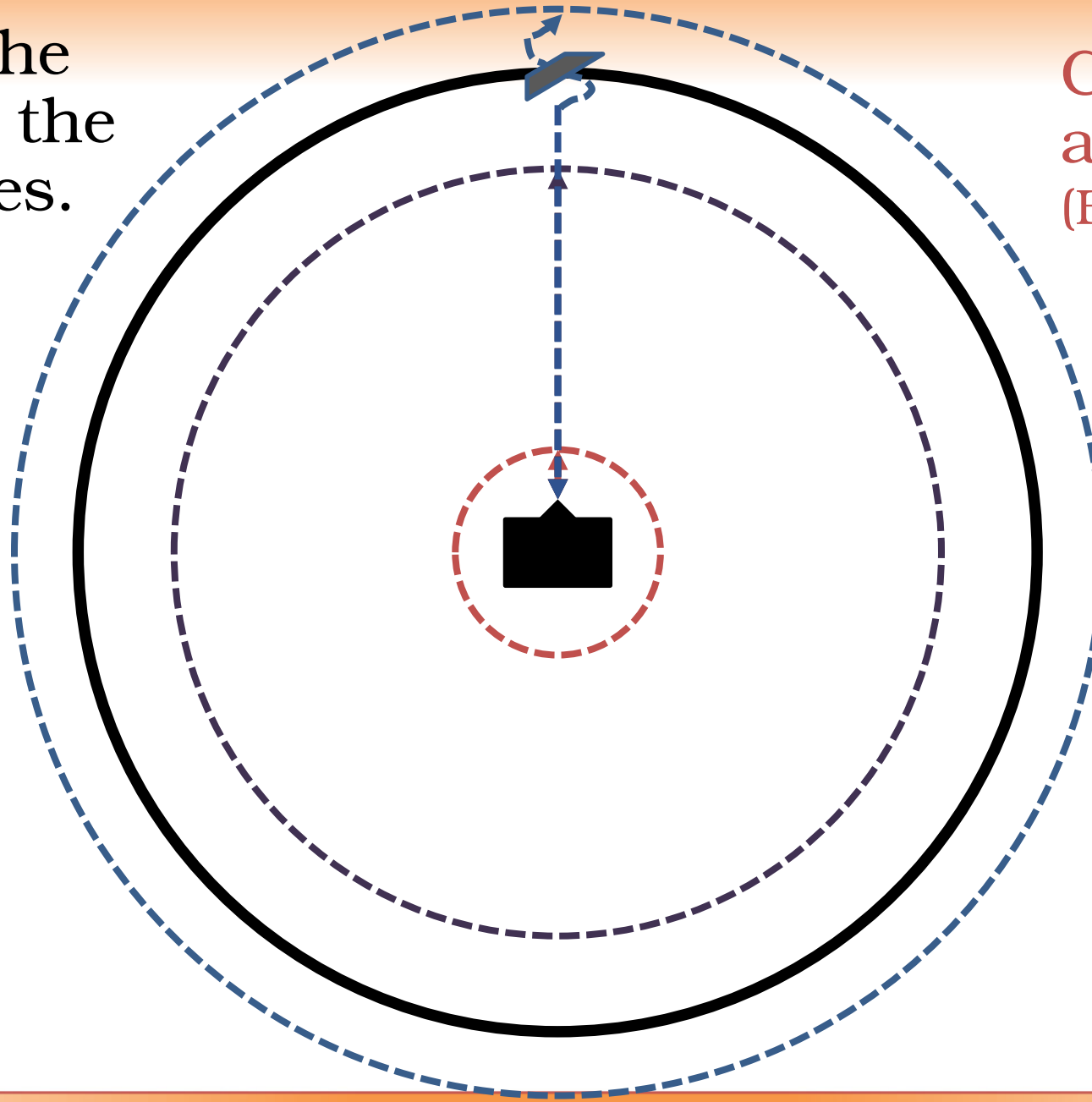
- There is no magic in computers. :-)
- Set-based is a *logical* concept – meaning all or nothing.
- No (normal) computer can update 30 000 values in one bang.
- Inside the UPDATE, it is all – loops.
- So why then not write your own loop and be in control?
- Because it matters where the loop is performed.

Walk around the box and touch the arrow 100 times.

Close loop around the box?
(Batch mode)

Along the walls inside the room?
(Row mode)

Walk outside of the room and in through the door every time?
(Run your own loop)



Which is the fastest way?

It's All About Overhead

- When you run the loop yourself, you open the “door” to the engine many times, and for every time you may repeat all sorts of things:
 - Accidental small scans.
 - Hardening transaction log for updates.
 - Other actions with high start-up cost, for instance linked servers, recompilation, triggers etc.
- With a set-based statement, you only open the “door” once.

A Different Example

Populate Excel sheet from PowerShell (or C#, whatever). Write cell by cell or something better?

[Slow version.](#)

[Fast version.](#)

Both solutions loops the dataset. But the fast version loops in .Net only. The slow version opens the door to Excel on every iteration.

Loading Data from Client to Database

- Sending data in a grid to a table – insert one row at a time, or all at once?
- If you send one row at a time:
 - Hardening the transaction log (unless you have a transaction).
 - Network overhead – particularly noticeable when the database is in the cloud.
- Instead use table-valued parameters:
 - You can send a .Net DataTable or a List<SqlDataRecord>.
 - Or you can stream the data!
- [Using Table-Valued Parameters in SQL Server and .NET](#)

Watch Out for Hidden Doors

[05_hidden-loop.sql](#)

- Scalar user-defined functions with data access are internal “doors” in the SQL Server room.
- Data access means that you have a query within the query that is executed separately for every row.
- The optimizer has no knowledge of what is inside the UDF or how cheap or expensive it is.
- For a scalar function without data access, the performance penalty is usually acceptable.

Reuse Stored Procedures?

```
WHILE EXISTS(SELECT * FROM #tmporders)
BEGIN
    SELECT @OrderID = MIN(OrderID) FROM #tmporders

    EXEC CorrectOrderDiscount @OrderID

    DELETE #tmporders WHERE OrderID = @OrderID
END
```

Good or bad?

Well, it depends...

Reuse Stored Procedures, cont'd

- Code reuse is a virtue in T-SQL too.
- ...but not as big virtue as in O-O languages.
- For a simple stored procedure that just inserts or updates a row – discard and write set-based.
- For a more complex procedure, write new proc with set-based logic and make existing SP a wrapper.
 - [How to Share Data between Stored Procedures](#)
- ...but for a very complex procedure, it may be prohibitly expensive.

Sometimes You Need to Loop

- Reuse complex stored procedure.
- Too difficult (for you) to implement set-based.
- Don't want to fail on single bad row.
- Other reason (for instance performance!)

These situations are rare – but they do occur.

So How Do You Loop?

Answer: In most cases...

Tada! You write a cursor!

Not because cursors are faster as such, but because the risk for hiccups are less than when you roll your own.

Provided that you set up the cursor the right way...

DECLARE CURSOR Syntax

ISO Syntax

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR  
FOR select_statement  
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
```

Transact-SQL Extended Syntax

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]  
[ FORWARD_ONLY | SCROLL ]  
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]  
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
[ TYPE_WARNING ]  
FOR select_statement  
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```


STATIC vs the Rest

- STATIC – query is evaluated once, result is stored in a worktable from which the cursor is served.
- ~~• Watch out, default is DYNAMIC, query is evaluated on every FETCH.
 - Slow.
 - Rows may come back if you update, causing infinite loop.~~
- ~~• FAST_FORWARD, like DYNAMIC with “optimisations”.~~
- ~~• KEYSET – Only keys are stored in worktable.~~

LOCAL vs GLOBAL

- LOCAL – Scope of cursor is the stored procedure (or function, trigger etc).
 - When scope exits, cursor goes away.
- GLOBAL – Cursor is process-global.
 - If procedure aborts on error and is called again on the same connection, cursor exists already => nasty error.
 - (But you need GLOBAL if you define the cursor in dynamic SQL.)
- Default is GLOBAL.
 - (Actually it is a DB setting, but default for that setting is GLOBAL.)

Pattern for Writing Cursors

```
DECLARE mycur STATIC LOCAL FOR
    SELECT col1, col2, col3, ...
    FROM ...
    ORDER BY ...

OPEN mycur
WHILE 1 = 1
BEGIN
    FETCH mycur INTO @var1, @var2, @var3, ...
    IF @@fetch_status <> 0
        BREAK
    -- Do stuff    (don't forget to initiate loop variables)
END
DEALLOCATE mycur
```

Using Cursor Variables

```
DECLARE @mycur CURSOR
SET @mycur = CURSOR STATIC FOR
    SELECT col1, col2, col3, ...
    FROM ...
    ORDER BY ...

OPEN @mycur
WHILE 1 = 1
BEGIN
    FETCH @mycur INTO @var1, @var2, @var3, ...
    IF @@fetch_status <> 0
        BREAK
    -- Do stuff (don't forget to initiate loop variables)
END
```

- Fat-fingering cursor name is caught at compile time.
- Don't need LOCAL or DEALLOCATE.

Common Anti-Pattern

```
DECLARE mycur STATIC LOCAL FOR
    SELECT col1, col2, col3, ...
    FROM   tbl
    ORDER BY ....
OPEN mycur
FETCH mycur INTO @var1, @var2, @var3, ...
WHILE @@fetch_status <> 0
BEGIN
    -- Do stuff
    FETCH mycur INTO @var1, @var2, @var3, ...
END
DEALLOCATE mycur
```

When changing SELECT, you may forget second FETCH.

When a Static Cursor Does Not Fit

- You actually want to see changes in the data that occur after you open the cursor.
- Cursor qualifies many rows, but you only need to look at a handful.
- In these cases, do the loop as roll-your-own. (And make sure you have indexes to support the loop!)
- *Never* use any of the other cursor types.

[06_cursordemo.sql](#)

Summing Up

- “Don’t use cursors” means “Write set-based code”.
- The “set-based” moniker really means “work with all data at once”.
 - Avoid repeating scans from source tables.
 - Don’t write one row at a time when you can write many.
- Avoid crossing boundaries repeatedly.
 - Recall the Excel example.
 - Access across a network.

Summing Up II

- Watch out for hidden boundaries.
 - Scalar functions with data access – a query within the query.
 - In client code: accessing an innocently looking class in a loop
 - but every access incurs a database call, more or less efficient.
- Keep in mind that when you work with data, you may be working with lots of data, GB, TB...

Summing Up III

- There are occasional situations when a loop is the right thing.
 - In T-SQL – rarely from client to server.
- When looping, use a static cursor – avoid “poor man’s cursor”.
 - Never use any of the other cursor types.
- Use Lee Tudor’s **sp_sqltrace** to find bottlenecks in loops.
 - <http://www.sommarskog.se/sqlutil/sqltrace.html>

End of the Loop

Erland Sommarskog

esquel@sommarskog.se

Slides and scripts on

<http://www.sommarskog.se/present>

(To create the database, first run [instnwnd.sql](#) and then [Northgale.sql](#))