

Dynamic Search Conditions

Erland Sommarskog
SQL Server MVP

SQL Saturday #642,
Sofia

Sponsors





Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

<http://www.sommarskog.se>

esquel@sommarskog.se

Dynamic Search Conditions

Be able to search on many different conditions (order id, date interval etc) with correct result **and** good performance for most or all conditions.

We will work in the database Northgale that has one million orders.

(To install, first run [instnwnd.sql](#) and then [Northgale.sql](#))

Today's Search Task

```
SELECT o.OrderID, o.OrderDate, od.UnitPrice, od.Quantity,  
       c.CustomerID, c.CustomerName, c.Address, c.City,  
       c.Region, c.PostalCode, c.Country, c.Phone, p.ProductID,  
       p.ProductName, p.UnitsInStock, p.UnitsOnOrder,  
       o.EmployeeID  
FROM   Orders o  
JOIN    [Order Details] od ON o.OrderID = od.OrderID  
JOIN    Customers c ON o.CustomerID = c.CustomerID  
JOIN    Products p ON p.ProductID = od.ProductID
```

We want to implement a search in this space of order lines.

SP Interface

```
CREATE PROCEDURE static_search_1
    @orderid      int          = NULL,
    @status       char(1)      = NULL,
    -- N, P, E or C. 99% are C. Filtered index on <> 'C'.
    @fromdate     datetime     = NULL, -- >= @fromdate.
    @todate       datetime     = NULL, -- <= @todate.
    @custid       nchar(5)     = NULL,
    @custname     nvarchar(40) = NULL, -- Starts with.
    @city         nvarchar(25) = NULL,
    @region       nvarchar(15) = NULL,
    @prodid       int          = NULL,
    @prodname     nvarchar(40) = NULL AS -- Starts with.
```

The search result is an intersection of the search conditions.

Static SQL or Dynamic SQL?

One of them is not better than the other. Where one is strong, the other is poor – and vice versa.

Static SQL – preferred for simple problems.

Dynamic SQL – when complexity grows.

We start with static SQL and then go dynamic.

General Pattern for Static SQL

```
WHERE (o.OrderID = @orderid OR @orderid IS NULL)
      AND (o.Status = @status OR @status IS NULL)
      AND (o.OrderDate >= @fromdate OR @fromdate IS NULL)
      AND (o.OrderDate <= @todate OR @todate IS NULL)
      AND (o.CustomerID = @custid OR @custid IS NULL)
      AND (c.CustomerName LIKE @custname + '%' OR @custname IS NULL)
      AND (c.City = @city OR @city IS NULL)
      AND (c.Region = @region OR @region IS NULL)
      AND (od.ProductID = @prodid OR @prodid IS NULL)
      AND (p.ProductName LIKE @prodname + '%' OR @prodname IS NULL)
ORDER BY o.OrderID
```


General Pattern for Static SQL

```
WHERE (o.OrderID = @orderid OR @orderid IS NULL)
AND (o.Status = @status OR @status IS NULL)
AND (o.OrderDate >= @fromdate OR @fromdate IS NULL)
AND (o.OrderDate <= @todate OR @todate IS NULL)
AND (o.CustomerID = @custid OR @custid IS NULL)
AND (c.CustomerName LIKE @custname + '%' OR @custname IS NULL)
AND (c.City = @city OR @city IS NULL)
AND (c.Region = @region OR @region IS NULL)
AND (od.ProductID = @prodid OR @prodid IS NULL)
AND (p.ProductName LIKE @prodname + '%' OR @prodname IS NULL)
ORDER BY o.OrderID
```

Ex: @fromdate = '2017-10-12', @city = 'Sofia'

What About Performance?

Let's test...

[static_search_1](#)

The plan is optimised for the first search ("parameter sniffing") and does not fit other search conditions.

We need different plans for different search conditions.

Side Track

Some people say that this is faster:

```
WHERE o.OrderID = isnull(@orderid, o.OrderID)
      AND o.Status = isnull(@status, o.Status)
      AND ...
```

[static_search_2](#)

This is a trap – does not work with NULL! **Do not use!**

And, no, it does not run faster.

Different Plans for Different Search Conditions

CREATE PROCEDURE ... WITH RECOMPILE AS

Compile the procedure every time.

Any improvement?

[static_search_3](#)

Better performance, but not optimal – index scan where we would like a seek.

Input parameters are sniffed, but the optimizer must consider that they may change before query runs – no flow analysis.

OPTION (RECOMPILE)

Query hint which you put after the SQL statement. It forces a recompile of that statement on every execution.

[static_search_4](#)

Since only the SQL statement is recompiled, all variables can be handled as **constants**.

To get good performance with searches with static SQL, you always need this hint.

Note: Requires SQL 2008 (latest service packs) or later. Works differently on SQL 2005.

Compiling Always is Expensive?

It depends...

A few searches per minute and 50 ms in compile time – no problem. Not the least if the total execution time goes from 5 seconds to 200 ms.

Searching on a simple condition like order ID 100 times a second – that hurts.

Specific Branches for Frequent Searches

```
IF @orderid IS NOT NULL
BEGIN
    SELECT ...
    WHERE  o.OrderID = @orderid
           AND  (o.Status = @status OR @status IS NULL)
           AND  ...
    -- OPTION (RECOMPILE)
```

Cached plan

```
END
ELSE
BEGIN
    SELECT ...
    WHERE  (o.Status = @status OR @status IS NULL)
           AND  ...
    OPTION (RECOMPILE)
END
```

Compilation every time

Different Branches, cont'd

Code is more difficult to maintain.

Two or three branches, but hardly more.

Can however be a viable alternative if at most three search conditions are indexed.

Dynamic SQL is better with high call frequency.

Multi-valued Search Conditions

Comma-separated list (CSV)

```
CREATE PROCEDURE static_search_5
    ...
    @employeeestr nvarchar(MAX) = NULL AS
    ...
    AND (o.EmployeeID IN
        (SELECT n FROM intlist_to_tbl (@employeeestr))
        OR @employeeestr IS NULL)
```

(For functions to unpack a CSV into a table see my article [Arrays and Lists in SQL Server.](#))

Table-Valued Parameters

Must use a variable to hold whether there is data in the table.

```
CREATE PROCEDURE static_search_5
    ...
    @employeeetbl intlist_tbltype READONLY AS

DECLARE @hastable bit =
    CASE WHEN EXISTS (SELECT * FROM @employeeetbl)
        THEN 1
        ELSE 0
    END

...
AND (o.EmployeeID IN (SELECT val FROM @employeeetbl) OR
    @hastable = 0)
```

Multi-Valued, Performance

Optimizer has less information.

For a TVP it knows the number of rows, but not more.

Good enough – if distribution is even. **Not if there is a skew.**

For a CSV the optimizer has no clue and makes a blind guess.

Bounce data over a temp table which has distribution statistics.

Use UPDATE STATISTICS to avoid stale stats.

Searches with Dynamic SQL

Higher level of difficulty.

Requires more programmer discipline.

More difficult to maintain if poorly written.

Requires more testing.

Permissions – users must have direct permissions to the tables.

(Can be addressed with certificate signing or EXECUTE AS. See my article [Granting Permissions through Stored Procedures.](#))

But you get a lot more flexibility.

sp_executesql

The core in all searches with dynamic SQL.

[sp_executesql](#)

Creates a nameless SP that is saved in the cache and executes it.
Next time it looks up the SP in cache.

The SP is identified by a hash of the query text as-is – no normalising of spacing, upper/lower etc.

First parameter: @sqlstring. **nvarchar!**

Second parameter: @paramlist. **nvarchar!**

Subsequent parameters: Parameter values for @paramlist.

dynamic_search_1

```
DECLARE @sql          nvarchar(MAX),
        @paramlist    nvarchar(4000),
        @nl            char(2) = char(13) + char(10)

SELECT @sql = 'SELECT o.OrderID, o.OrderDate, ...
              FROM   dbo.Orders o
              JOIN   dbo.[Order Details] od ON o.OrderID = od.OrderID
              JOIN   dbo.Customers c ON o.CustomerID = c.CustomerID
              JOIN   dbo.Products p  ON p.ProductID = od.ProductID
              WHERE  1 = 1' + @nl
```

Users may have different default schemas.

Makes it easier to add conditions.

Improves readability of the generated SQL.

Add Conditions

```
IF @orderid IS NOT NULL
```

```
    SET @sql += ' AND o.OrderID = @orderid' + @nl
```

```
IF @fromdate IS NOT NULL
```

```
    SET @sql += ' AND o.OrderDate >= @fromdate' + @nl
```

```
IF @custname IS NOT NULL
```

```
    SET @sql += ' AND c.CustomerName LIKE @custname + ''' + @nl
```

+= handy to make code a little shorter.

Always a space after the opening quote.

Append @nl to make string more readable.

Need to double single quotes in the string.

Multi-Valued Parameters

```
IF EXISTS (SELECT * FROM @employeeetbl)
    SELECT @sql += ' AND o.EmployeeID IN
        (SELECT val FROM @employeeetbl) ' + @nl
```

```
IF @employeeestr IS NOT NULL
    SELECT @sql += ' AND o.EmployeeID IN
        (SELECT n FROM dbo.intlist_to_tbl(@employeeestr)) ' + @nl
```

What about?

Don't even think about it!

```
IF @employeeestr IS NOT NULL
    SELECT @sql += ' AND o.EmployeeID IN (' + @employeeestr + ') ' + @nl
```

We will come back to this in a few slides.

The Debug Parameter

This line should always be there when you work with dynamic SQL.

```
@debug bit = 0 AS  
  
...  
IF @debug = 1  
    PRINT @sql
```

ALWAYS!

dynamic_search_1 – Making the Call

```
SELECT @paramlist =  
    '@orderid      int,  
    @status       char(1),  
    ...  
    @employeeestr varchar(MAX),  
    @employeeetbl intlist_tbltype READONLY'  
  
EXEC sp_executesql @sql, @paramlist,  
    @orderid, @status, @fromdate, @todate,  
    @custid, @custname, @city, @region,  
    @prodid, @prodname, @employeeestr, @employeeetbl
```

[dynamic_search_1](#)

All parameters to the SP are included in the parameter list.

Cache and Compilation

OPTION (RECOMPILE)

Compilation every time.

More compilation than needed.

The plan always fits the current parameters.

Dynamic SQL with Parameters

Cached plan per combination of parameters.

Much less compiling.

“Parameter sniffing” can be a problem.

A Bad Example

Some people concatenate the values into the SQL string:

```
IF @orderid IS NOT NULL
    SELECT @sql += ' AND o.OrderID = ' +
                  convert(varchar(10), @orderid) + @nl
...
IF @city IS NOT NULL
    SELECT @sql += ' AND c.City = ''' + @city + '''' + @nl
...
IF @employeeestr IS NOT NULL
    SELECT @sql += ' AND o.EmployeeID IN (' + @employeeestr + ') '
```

[dynamic_search_bad](#)

Opens for SQL injection!

Cache and Compilation II

```
EXEC static_search_4 11000  
EXEC static_search_4 11001  
EXEC static_search_4 11002
```

3 compilations
1 (unused) cache entry

```
EXEC dynamic_search_1 11000  
EXEC dynamic_search_1 11001  
EXEC dynamic_search_1 11002
```

1 compilation
1 cache entry

```
EXEC dynamic_search_bad 11000  
EXEC dynamic_search_bad 11001  
EXEC dynamic_search_bad 11002
```

3 compilations
3 cache entries

Problems for Parameterised SQL

Cached plan is good – but not always.

[compare_1](#)

```
EXEC dynamic_search_1 @custid = 'ERNTC',  
    @fromdate = '19980218', @todate = '19980218'  
EXEC dynamic_search_1 @custid = 'BOLSR',  
    @fromdate = '19980101', @todate = '19981231'
```

Same parameters passed, but which is the best index depends on the parameter values.

```
EXEC dynamic_search_1 @status = 'E'
```

Optimizer cannot use filtered indexes.

Dealing with Parameter Sniffing

OPTION (RECOMPILE)

```
SELECT @sql += ' ORDER BY o.OrderID' + @nl
IF @custid IS NOT NULL AND (@fromdate IS NOT NULL OR
                             @todate IS NOT NULL)
    SELECT @sql += ' OPTION (RECOMPILE) ' + @nl
```

Good solution as long as call frequency is not a concern.

Altering the Query Text Depending on Values

```
IF @fromdate IS NOT NULL AND @todate IS NOT NULL
BEGIN
    SELECT @datediff = datediff(DAY, @fromdate, @todate)
    SELECT @sql += CASE WHEN @datediff = 0 THEN '-- Single day'
                        WHEN @datediff <= 7 THEN '-- A week'
                        WHEN @datediff <= 30 THEN '-- A month'
                        ELSE '-- More than a month'
                    END + @nl
END
```

Requires some understanding of the data.

Less granular than OPTION(RECOMPILE) but less compiling.

Handle Common Values

Say that 60% of all customers are in London.

```
IF @city IS NOT NULL
    SELECT @sql += ' AND c.City = ' +
        CASE @city WHEN N'London' THEN N'N''London'''
                ELSE '@city'
        END + @nl
```

But don't inline names of cities, customers etc on a general basis
– you will litter the cache.

Skewed Columns Such as Status

With few possible values, it may make sense to inline the value:

```
IF @status IS NOT NULL
    SELECT @sql += ' AND o.Status = ' + quotename(@status, ''')
    + @nl
```

Enable a filtered index: add filter condition to query:

```
IF @status IS NOT NULL
    SELECT @sql += ' AND o.Status = @status' +
        CASE WHEN @status <> 'C'
            THEN ' AND o.Status <> ''C'''
            ELSE ''
        END + @nl
```

Alternate Tables, Static SQL

When @ishistoric = 1, read historic order tables.

```
FROM    (SELECT o.OrderID, o.Status, ...
        FROM    Orders o
        JOIN    [Order Details] od ON o.OrderID = od.OrderID
        WHERE   @ishistoric = 0
        UNION   ALL
        SELECT ho.OrderID, ho.Status, ...
        FROM    HistoricOrders ho
        JOIN    HistoricOrderDetails hod
              ON ho.OrderID = hod.OrderID
        WHERE   @ishistoric = 1) AS u

...
OPTION  (RECOMPILE)
```

Alternate Tables, Dynamic SQL

```
FROM    dbo.' + CASE @ishistoric
          WHEN 0 THEN 'Orders'
          WHEN 1 THEN 'HistoricOrders'
        END + ' o
JOIN    dbo.' + CASE @ishistoric
          WHEN 0 THEN '[Order Details]'
          WHEN 1 THEN 'HistoricOrderDetails'
        END + ' od ON o.OrderID = od.OrderID
```


Control Sort Order, Dynamic SQL

Map input from client to column in a CASE:

```
SELECT @sql += ' ORDER BY ' +  
        CASE @sortcol1  
            WHEN 'OrderID'          THEN 'o.OrderID'  
            WHEN 'EmployeeID'       THEN 'o.EmployeeID'  
            WHEN 'ProductID'        THEN 'od.ProductID'  
            WHEN 'CustomerName'     THEN 'c.CustomerName'  
            WHEN 'ProductName'       THEN 'p.ProductName'  
            WHEN 'OrderDate'        THEN 'o.OrderDate'  
            ELSE                     'o.OrderID'  
        END +  
        CASE @isdesc1 WHEN 0 THEN ' ASC' ELSE ' DESC' END
```

Must have an ELSE, to avoid NULL in @sql!

Control Sort Order, Static SQL

Need multiple CASE to keep data types apart:

```
ORDER BY CASE @sortcol WHEN 'OrderID' THEN o.OrderID
                        WHEN 'EmployeeID' THEN o.EmployeeID
                        WHEN 'ProductID' THEN od.ProductID
                        END,
CASE @sortcol WHEN 'CustomerName' THEN c.CustomerName
              WHEN 'ProductName' THEN p.ProductName
              END,
CASE @sortcol WHEN 'OrderDate' THEN o.OrderDate
              END
OPTION (RECOMPILE)
```

If you want to provide multi-column sort, ASC/DESC, it quickly goes out of hand.

GROUP BY, Aggregation, Select Columns etc

There is a limit for dynamic SQL in an SP as well.

Key problem: How express all this in a parameter list?

May be simpler to construct the SQL client-side in a class with an O-O interface.

Never send SQL syntax to a stored procedure!

Conclusion

Static SQL with OPTION (RECOMPILE) – good for the simple cases, but complexity grows fast.

Dynamic SQL – too much hassle for simple cases, but scales better in complexity.

Make a decision from case to case what to use.

That's All Folks!

Erland Sommarskog

esquel@sommarskog.se

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

Slides and all scripts, including scripts to create the database.

Sponsors

