

Dynamic Search Conditions

Erland Sommarskog
SQL Server MVP

Pinellas User Group
March 16th 2021



Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

<http://www.sommarskog.se>

esquel@sommarskog.se

Slides and scripts on <http://www.sommarskog.se/present>

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.

A production database may have 100 million orders...

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.

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.

Search Parameters

@orderid	int	Search on a specific order.
@status	char(1)	N(ew), P(rocessing), E(rror), C(ompleted). 99% are C. Filtered index on <> 'C'.
@fromdate	date	>= @fromdate.
@todate	date	<= @todate.
@custid	nchar(5)	Search on specific customer.
@custname	nvarchar(40)	Starts with. 'Bla' returns both Black and Blake.
@city	nvarchar(25)	City customer comes from.
@region	nvarchar(15)	Region for customer.
@prodid	int	Specific product.
@prodname	nvarchar(40)	Starts with.

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 = '2021-03-14', @city = 'Tampa'

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.

A Trap to Watch Out For

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](#)

Rows with NULL in a nullable column are filtered out when they should not be. **Do not use!**

And, no, it does not run faster. It is still a static plan.

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 — scans instead of seeks.

Input parameters are sniffed, but the optimizer must consider that they may change before query runs.

OPTION (RECOMPILE)

Query hint that 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 (almost) always need this hint.

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
```

```
...
```

```
WHERE o.OrderID = @orderid  
      AND (o.Status = @status OR @status IS NULL)
```

```
-- OPTION (RECOMPILE)
```

Cached plan

```
END  
ELSE  
BEGIN
```

```
...
```

```
WHERE (o.Status = @status OR @status IS NULL)  
      AND (o.OrderDate >= @fromdate OR @fromdate IS NULL)
```

```
...
```

```
OPTION (RECOMPILE)
```

Compilation every time

```
END
```


Different Branches

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
    ...
    @employeecsv nvarchar(MAX)= NULL AS
    ...
    AND (o.EmployeeID IN
        (SELECT n FROM dbo.intlist_to_tbl (@employeecsv))
        OR @employeecsv IS NULL)
```

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

Table-Valued Parameters

Need a variable to hold whether the table has data.

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

DECLARE @hastable bit =
    IIF(EXISTS (SELECT * FROM @employeeetbl), 1, 0)

...
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!

Multi-Valued, Performance CSV

For a CSV – this gets complicated. :-)

With a multi-statement T-SQL UDF:

- In SQL 2012 and earlier: optimizer makes a blind guess of one row.
- In SQL 2014 and SQL 2016: blind guess of 100 rows.
- In SQL 2017 and later: optimizer runs UDF before compiling the rest and gets the correct cardinality. Thus, in par with a TVP.
 - But only for pure SELECT, not INSERT, UPDATE etc.

With the built-in function `string_split`, blind guess of 50 rows!

Other methods? Typically also blind guesses.

Multi-Valued, Ultimate Solution

To handle skew (or poor estimates for CSVs):

Bounce data over a temp table to get correct cardinality and distribution statistics.

Use `UPDATE STATISTICS` to avoid running with statistics from previous execution.

Alternate Tables

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
```

Works — but code is more complex. With many tables, it can become unwieldy.

Control Sort Order

CASE to the rescue – be aware of data types!

```
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
```

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

Searching on Alternate Keys

We want to look up customer on one of customer ID, tax number or name. Index on all three columns.

What do you think of:

```
WHERE (CustomerID = @custid AND @custid IS NOT NULL)
      OR (VATno = @vatno AND @vatno IS NOT NULL)
      OR (CustomerName LIKE @custname + '%' AND @custname IS NOT NULL)
```

Note: No OPTION (RECOMPILE)!

Start-up Filter

The plan has searches on all three indexes with a start-up filter
=> only one index is used at a time.

The condition @val IS NOT NULL must be there!

Rarely (if ever) works if search columns are in different tables.

Always test that the plan is what you intended.

Be aware of that parameter sniffing may trip you.

[cust_lookup](#)

Other Examples with Start-up Filters

```
WHERE (@suppl_city IS NULL OR
      EXISTS (SELECT *
              FROM Suppliers s
              WHERE s.SupplierID = p.SupplierID
                   AND s.City      = @suppl_city))
```

Note: this is not operator shortcutting!

Start-up Filters, cont'd

We have seen this one before.

```
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
```

Again: Always test to verify that start-up filters are used!

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 [*Packaging Permissions in Stored Procedures*](#). I'm also presenting this topic [April 13th](#).]

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 procedure in cache.

The procedure 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),
        @n1           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' + @n1
```

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 @employeetbl)
  SELECT @sql += ' AND o.EmployeeID IN
    (SELECT val FROM @employeetbl)' + @nl
```

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

What about?

Don't even think about it!

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

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

[dynamic_search_bad](#)

Opens for SQL injection!

Cache and Compilation I

```
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

Cache and Compilation II

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.

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 understanding of the data.

Less granular than OPTION(RECOMPILE) but less compiling.

Altering the Query Text, Variation

```
DECLARE @days int = datediff(DAY, @fromdate, @todate) + 1
SELECT @sql += ' -- ' +
    CASE WHEN @days < 7
        THEN convert(varchar, @days) + ' days.'
    WHEN @days < 35
        THEN convert(varchar, @days / 7) + ' weeks.'
    ELSE convert(varchar, @days / 30) + ' months.'
    END + @nl
```

More cache entries of which some have the same plan.
But less risk for poor plain choices.
The power of dynamic SQL!

Inlining Certain 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
```

Columns with a few possible values that are skewed:

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

Don't inline on a general basis — you will litter the cache!

Enable Filtered Indexes

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
```


Select Sort Order

Could it be this simple?

```
@sql += ' ORDER BY ' + @sortcol
```

How neat — @sortcol could be a list of columns!

No! Risk for SQL injection. Use quotename!

```
@sql += ' ORDER BY ' + quotename(@sortcol)
```

Now lists will not work, but you need @sortcol1, @sortcol2 etc. Or decode @sortcol.

But client still has a dependency on column names.

Select Sort Order, cont'd

Thus, this is likely to be better after all:

```
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 ELSE with a default to avoid NULL in @sql!

Alternate Tables

Don't send in table names, but translate to dbo.tblname etc.

Implementing @ishistoric:

```
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
```

GROUP BY, Aggregation, Select Columns etc

There is a limit for what you can do with dynamic SQL in an stored procedure 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.

Sometimes you can rely on start-up filters, in which case you don't need `OPTION(RECOMPILE)`. (But test carefully!)

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.

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