Dynamic SQL

brentozar.com/sql/dynamic

/*

Dynamic SQL - When, Why, and How to Use It

v1.8 - August 2019

You're comfortable writing T-SQL, and you've built a lot of stored procedures

that have a bunch of parameters. For example, you have that "product search"

stored proc with parameters for product category, name, price range, sort

order, etc, and you have to accept any of 'em.

So how do we make those go fast? And how can we get 'em to use indexes?

In one all-demo hour, performance tuner Brent Ozar will show you several ways

that fail comically. You'll learn how to write dynamic SQL that's easy to

tune, manage, and troubleshoot.

Download the latest version free: https://www.BrentOzar.com/go/dynamicsql

Open source with the Creative Commons License. For details, see the end of this file.

Requirements:

* Any Stack Overflow database: https://BrentOzar.com/go/querystack

* Any supported SQL Server or Azure SQL DB

This first RAISERROR is just to make sure you don't accidentally hit F5 and

run the entire script. You don't need to run this:

*/

RAISERROR(N'Oops! No, don''t just hit F5. Run these demos one at a time.', 20, 1) WITH LOG;

GO

/*

Make sure we don't have extra nonclustered indexes that would interfere with

the expected demo results.

If you don't have this proc: https://www.brentozar.com/go/dropindexes

*/

Use StackOverflow2013;

GO

EXEC DropIndexes;

GO

/* Turn on our tuning options and Actual Execution Plans: */

SET STATISTICS IO ON;

GO

/* Meet the Users table: */

SELECT TOP 100 * FROM dbo.Users;

```
GO

/*

How big is the users table? Look at how many rows are returned from this query,

and on the Messages tab, check the STATISTICS IO results to see how many

logical reads we did. That's how many 8KB pages the table has.

*/

SELECT COUNT(*) FROM dbo.Users;

GO

/*

We'll be searching by DisplayName, Location, and/or Reputation, so create

indexes on those fields:

*/

CREATE INDEX IX_DisplayName ON dbo.Users(DisplayName);

CREATE INDEX IX_Location ON dbo.Users(Location);

CREATE INDEX IX_Reputation ON dbo.Users(Reputation);

GO

/*

Folks want a single stored procedure that can search on any of these 3 fields.

You should be able to pass in 1, 2, or all 3, and find just the users that

match ALL of your parameters.

They want a stored proc that works like this:

*/

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO
```

```
EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchLocation = 'San Diego%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%', @SearchReputation = 107548;

GO

/*

Here's one way to do it. It's a bad way - it doesn't really work - but let's

look at how it performs:

*/

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

/* OrderBy isn't implemented yet in this version - I swear I'll do that later. Love, The Last Guy
*/

IF @SearchDisplayName IS NOT NULL

SELECT *

  FROM dbo.Users

  WHERE DisplayName LIKE @SearchDisplayName

  ORDER BY CreationDate;
```

```
ELSE IF @SearchLocation IS NOT NULL
SELECT *
  FROM dbo.Users
  WHERE Location LIKE @SearchLocation
  ORDER BY CreationDate;
ELSE IF @SearchReputation IS NOT NULL
SELECT *
  FROM dbo.Users
  WHERE Reputation = @SearchReputation
  ORDER BY CreationDate;
END
GO
/* Will that work? Is there a bug in that logic? */
/* If we pass in multiple fields, like this, it only searches for one of them: */
EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchLocation = 'San Diego%';
GO
/*
But forget accuracy. Does it perform? Turn on actual plans and run: */
EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';
GO
EXEC usp_SearchUsers @SearchReputation = 2;
GO
EXEC usp_SearchUsers @SearchLocation = 'Indiana%';
GO
```

```
/*

When I'm looking at search code performance, I'll often ask:

* Is the optimizer willing to use indexes?

* Are the row estimates right?

In this case, all 3 of the branches were willing to use indexes - too willing -

and the estimates weren't right - because SQL Server built the entire execution

plan for all 3 branches when we ran the very first query.

Notice some of the query plans have yellow bang warnings on their sort operator

because SQL Server underestimated how many rows those queries would bring back.

It's that pesky ORDER BY CreationDate.

Does it get better if you clear the cache, and try a different branch first?

*/

DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

/*

Sometimes - for some branches - but not always. That's the devil of parameter

sniffing - the whole plan gets optimized for the first set of parameters that

get used, even if some branches rely on parameters that didn't get passed in.
```

They can get optimized for NULL, which can give you really low estimates.

Sometimes that's good - but not usually.

The IF branch design has a few drawbacks:

* It's very susceptible to parameter sniffing

* Multi-parameter results don't match what the users asked for

* Getting multi-parameter results to work right would require a lot of code

```
IF @SearchDisplayName IS NOT NULL AND @SearchLocation IS NOT NULL

   ...

IF @SearchDisplayName IS NOT NULL AND @SearchLocation IS NOT NULL AND
@SearchReputation....

   ...
```

Yeah, that's not gonna scale. So let's try a different design:

*/

```
CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

SELECT *

FROM dbo.Users

WHERE (DisplayName LIKE @SearchDisplayName OR @SearchDisplayName IS NULL)

  AND (Location LIKE @SearchLocation OR @SearchLocation IS NULL)

  AND (Reputation = @SearchReputation OR @SearchReputation IS NULL)

ORDER BY CreationDate;
```

```
END

GO

/*

Good news! This will give the right results most of the time. (Handling null
parameters that are specifically looking for nulls only is another story.)

Let's try to run it and see what indexes it chooses to use:

*/

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

/*

Look at the plans, and...

HAHAHAHAHAHAHAHAHAHAHAHA

(breathes)

HAHAHAHAHAHAHAHAHAHAHAHAHAHAHAHAHAHAHA

Then look at logical reads!

(passes out)

That, ladies and gentlemen, is a really bad query plan. SQL Server uses the
first set of incoming parameters to build the plan - in our case, it builds a
plan that uses the DisplayName index.

And when DisplayName isn't passed in, SQL SERVER STILL USES THAT INDEX, doing
```

```sql
a key lookup of every single row. It does millions of logical reads - even
though the whole table only has about 80k pages.
Alright, screw that.
How about ISNULL?
*/
CREATE OR ALTER PROC dbo.usp_SearchUsers
@SearchDisplayName NVARCHAR(100) = NULL,
@SearchLocation NVARCHAR(100) = NULL,
@SearchReputation INT = NULL,
@OrderBy NVARCHAR(100) = 'CreationDate' AS
BEGIN
SELECT *
FROM dbo.Users
WHERE DisplayName LIKE ISNULL(@SearchDisplayName, DisplayName)
  AND Location LIKE ISNULL(@SearchLocation, Location)
  AND Reputation = ISNULL(@SearchReputation, Reputation)
ORDER BY CreationDate;
END
GO
EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';
GO
EXEC usp_SearchUsers @SearchLocation = 'Indiana%';
GO
EXEC usp_SearchUsers @SearchReputation = 2;
```

```
GO

/* How about COALESCE? */

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

SELECT *

FROM dbo.Users

WHERE DisplayName LIKE COALESCE(@SearchDisplayName, DisplayName)

  AND Location LIKE COALESCE(@SearchLocation, Location)

  AND Reputation = COALESCE(@SearchReputation, Reputation)

ORDER BY CreationDate;

END

GO

/* Run 'em and check to see if they use the indexes: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

/*
```

Depending on your version & cardinality estimator, you're probably going to get

either clustered index scans, or some very unusual index intersection plans

that don't perform worth a dang.

Now you see why people have to resort to dynamic SQL.

So let's spend some time at the Dynamic SQL Resort:

```
*/

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

SET @StringToExecute = N'SELECT * FROM dbo.Users WHERE 1 = 1 ';

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ';

IF @SearchLocation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Location LIKE @SearchLocation ';

IF @SearchReputation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Reputation = @SearchReputation ';

SET @StringToExecute = @StringToExecute + N' ORDER BY CreationDate; ';

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @SearchLocation NVARCHAR(100),
@SearchReputation INT',
@SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/* Run these one at a time, and look at the plans. Do they use indexes? */

DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

/* What if we use combinations of fields? */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchLocation = 'San Diego%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%', @SearchReputation = 107548;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchReputation = 12145;

GO

/* Is it bloating the plan cache? Turn OFF actual plans: */

DBCC FREEPROCCACHE

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO
```

```
EXEC usp_SearchUsers @SearchLocation = 'Indiana%';

GO

EXEC usp_SearchUsers @SearchReputation = 2;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchLocation = 'San Diego%';

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%', @SearchReputation = 89836;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchReputation = 12145;

GO

sp_BlitzCache

GO

/*

There's a line in the plan cache for every variation of the dynamic SQL, but

whenever we use different parameters, they'll reuse an existing plan. This

might look like a lot of plans at first glance - but it's not really that many,

especially as opposed to building a line for every single combination of

parameter VALUES, like Brent or Erik or Tara.
It's a little bit of plan cache bloat - but totally worth it since each

combination of parameters can get its own customized plan with good (hopefully)

indexes and memory grants.

*/

/* Pro Tip: in your dynamic SQL, throw in a comment to include the source. */

CREATE OR ALTER PROC dbo.usp_SearchUsers
```

```
@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

SET @StringToExecute = N'/* usp_SearchUsers */ SELECT * FROM dbo.Users WHERE 1 = 1 ';

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ';

IF @SearchLocation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Location LIKE @SearchLocation ';

IF @SearchReputation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Reputation = @SearchReputation ';

SET @StringToExecute = @StringToExecute + N' ORDER BY CreationDate; ';

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @SearchLocation NVARCHAR(100),
@SearchReputation INT',

@SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/*

So that when you're viewing plans, you can see the source. You're nbot going to

see it in the sp_BlitzCache "Query Text" column, but you'll see it in the plan.

*/
```

```
DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

sp_BlitzCache;

GO

/* Pro Tip: add line feeds to make it easier to read. */

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @StringToExecute = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT * FROM
dbo.Users WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ' + @crlf;

IF @SearchLocation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Location LIKE @SearchLocation ' +
@crlf;

IF @SearchReputation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Reputation = @SearchReputation ' +
@crlf;
```

```
SET @StringToExecute = @StringToExecute + N' ORDER BY CreationDate; ';

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @SearchLocation NVARCHAR(100),
@SearchReputation INT',

@SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/*

Now check out the nicely formatted string - after sp_BlitzCache runs, click on

the query plan, right-click on the query, and click Edit Query Text:

*/

DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

sp_BlitzCache;

GO

/* But do NOT put dynamic stuff in the comments, les you end up with

different plans for every dynamic thing:

*/

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate' AS
```

```
BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @StringToExecute = @crlf + N'/* usp_SearchUsers at date/time: ' +
CONVERT(NVARCHAR(100), GETDATE(), 21) + '*/' + @crlf + N' SELECT * FROM dbo.Users
WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ' + @crlf;

IF @SearchLocation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Location LIKE @SearchLocation ' +
@crlf;

IF @SearchReputation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Reputation = @SearchReputation ' +
@crlf;

SET @StringToExecute = @StringToExecute + N' ORDER BY CreationDate; ';

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @SearchLocation NVARCHAR(100),
@SearchReputation INT',

@SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/* Because every string will be unique: */

DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO
```

```
EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%';

GO

sp_BlitzCache;

GO

/*

See how sp_BlitzCache shows four different lines of dynamic SQL now?

Click on each of their plans, and look at the query text to see how the time

text comments meant that each query has different text, so it gets different

entries in the plan cache.

*/

/* Pro Tip: Every proc with dynamic SQL needs a @Debug_PrintQuery parameter to print
strings: */

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate',

@Debug_PrintQuery TINYINT = 0,

   @Debug_ExecuteQuery TINYINT = 1 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);
```

```
SET @StringToExecute = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT * FROM
dbo.Users WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ' + @crlf;

IF @SearchLocation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Location LIKE @SearchLocation ' +
@crlf;

IF @SearchReputation IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND Reputation = @SearchReputation ' +
@crlf;

SET @StringToExecute = @StringToExecute + N' ORDER BY CreationDate; ';

IF @Debug_PrintQuery = 1

      PRINT @StringToExecute;

   IF @Debug_ExecuteQuery = 1

   EXEC sp_executesql @StringToExecute,

   N'@SearchDisplayName NVARCHAR(100), @SearchLocation NVARCHAR(100),
@SearchReputation INT',

   @SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/* So you can see the T-SQL in the Messages tab, complete with line returns: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%',

   @Debug_PrintQuery = 1, @Debug_ExecuteQuery = 1;

GO

/*
```

```
PRINT is limited to 4,000 characters. If you build longer strings, use this
stored procedure, Helper_LongPrint:
https://www.codeproject.com/Articles/18881/SQL-String-Printing
*/
/*
Let's implement that ORDER BY statement.
Just to keep the projector screen readable, I'm going to ignore the filters for
@SearchLocation and @SearchReputation, and only work on the ORDER BY part.
*/
CREATE OR ALTER PROC dbo.usp_SearchUsers
@SearchDisplayName NVARCHAR(100) = NULL,
@OrderBy NVARCHAR(100) = 'CreationDate',
@Debug_PrintQuery TINYINT = 0 AS
BEGIN
DECLARE @StringToExecute NVARCHAR(4000);
DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);
SET @StringToExecute = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT * FROM
dbo.Users WHERE 1 = 1 ' + @crlf;
IF @SearchDisplayName IS NOT NULL
SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ' + @crlf;
IF @OrderBy IS NOT NULL
SET @StringToExecute = @StringToExecute + N' ORDER BY ' + @OrderBy;
IF @Debug_PrintQuery = 1
    PRINT @StringToExecute;
```

```sql
EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @OrderBy NVARCHAR(100)',

@SearchDisplayName, @OrderBy;

END

GO

/*

Let's see what QUERY (not query plan) that it builds. I'm less worried about

performance here, and more just worried about getting the query right first.

*/

SET STATISTICS IO, TIME OFF;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation',
@Debug_PrintQuery = 1;

GO

/* And try descending order: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation DESC',
@Debug_PrintQuery = 1;

GO

/* Pretty cool. And how about this: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation DESC;
SELECT * FROM sys.databases;', @Debug_PrintQuery = 1;

GO

/* Oh. Well, that's not good. That's SQL injection.

Never, ever use the stored procedure's input parameters as part of the dynamic

SQL itself. Never, ever. Obligatory comic strip: https://xkcd.com/327/
```

Think you can work around it via regex and sanitizing? Think again, as Bert

Wagner explains: https://groupby.org/conference-session-abstracts/sql-injection-attacks-is-your-data-secure/

Instead, examine the user's input, and substitute your own:

```
*/

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate',

@Debug_PrintQuery TINYINT = 0 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @StringToExecute = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT * FROM dbo.Users u WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE @SearchDisplayName ' + @crlf;

  IF @OrderBy IS NOT NULL

    BEGIN

    SET @StringToExecute = @StringToExecute + N' ORDER BY ';

    SET @StringToExecute = @StringToExecute +

      CASE WHEN @OrderBy LIKE 'CreationDate%' THEN N' u.CreationDate '

        WHEN @OrderBy LIKE 'DisplayName%' THEN N' u.DisplayName '

        WHEN @OrderBy LIKE 'Location%' THEN N' u.Location '

        WHEN @OrderBy LIKE 'Reputation%' THEN N' u.Reputation '
```

```
            ELSE N' u.Id '   /* Or whatever default ordering you want, ideally to make SQL's life
easier */

         END;

      IF @OrderBy LIKE '% DESC' /* You could also do this with a separate @OrderByDesc bit
parameter if you want */

         SET @StringToExecute = @StringToExecute + N' DESC ';

      SET @StringToExecute = @StringToExecute + N';'; /* Because technically, we're
supposed to */

      END

   IF @Debug_PrintQuery = 1

      PRINT @StringToExecute;

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100), @OrderBy NVARCHAR(100)',

@SearchDisplayName, @OrderBy;

END

GO

/* Test it: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation',
@Debug_PrintQuery = 1;

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation DESC',
@Debug_PrintQuery = 1;

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @OrderBy = 'Reputation DESC;
SELECT * FROM sys.databases;', @Debug_PrintQuery = 1;

GO

/*

Note that the more complex our dynamic SQL gets, like with different possible

filters, joins, and order by clauses, the more we may need specialized indexes.
```

But don't try to guess which ones need indexes! Use sp_BlitzCache to review

your most resource-intensive queries. That'll tell you which ones need the most

help because they're getting called more frequently and/or sucking the most.

*/

/*

Next up, how about letting users pick what fields they want to see, and from

which tables? We already know we NEVER EVER want to let them send their own

T-SQL in as a parameter because it's a recipe for SQL injection.

Here's one implementation - again, I'm going to remove the search parameters

and the order-by stuff just to keep my presentation screen easier to read, and

only focus on the dynamic join implementation:

*/

```
CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@ShowTotalVotes BIT = 0,

@Debug_PrintQuery TINYINT = 0 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @StringToExecute = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT * FROM
dbo.Users u WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @StringToExecute = @StringToExecute + N' AND DisplayName LIKE
@SearchDisplayName ' + @crlf;

   IF @ShowTotalVotes = 1
```

```
        SET @StringToExecute = @StringToExecute + N' LEFT OUTER JOIN dbo.Votes v ON u.Id =
v.UserId ';

    IF @Debug_PrintQuery = 1

        PRINT @StringToExecute;

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100)',

@SearchDisplayName;

END

GO

/* Let's see how it works: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @ShowTotalVotes = 1,
@Debug_PrintQuery = 1;

GO

/*

Well, that's not gonna work.

Our dynamic SQL building is gonna have to get a lot more intelligent, and we'll

need to build the string in different parts:

* @Select

* @From

* @Where

* @OrderBy

And then at the end of our string-building, assemble them all into one like

some kind of transformer.

*/

CREATE OR ALTER PROC dbo.usp_SearchUsers
```

```
@SearchDisplayName NVARCHAR(100) = NULL,

@ShowTotalVotes BIT = 0,

@Debug_PrintQuery TINYINT = 0 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000), @Select NVARCHAR(4000),

        @From NVARCHAR(4000), @Where NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @Select = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT u.Id, u.DisplayName,
u.Location ' + @crlf;

SET @From = N' FROM dbo.Users u ' + @crlf;

SET @Where = N' WHERE 1 = 1 ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @Where = @Where + N' AND DisplayName LIKE @SearchDisplayName ' + @crlf;

  IF @ShowTotalVotes = 1

    BEGIN

    SET @Select = @Select + N' , VotesCast = (SELECT SUM(1) FROM dbo.Votes v WHERE u.Id
= v.UserId) ' + @crlf;

    END



  /* Autobots, unite! */

  SET @StringToExecute = @Select + @From + @Where;

  IF @Debug_PrintQuery = 1

    PRINT @StringToExecute;

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100)',
@SearchDisplayName;

END

GO

/* Let's see how it works: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @ShowTotalVotes = 0,
@Debug_PrintQuery = 1;

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @ShowTotalVotes = 1,
@Debug_PrintQuery = 1;

GO

/*

The beauty of dynamic SQL here is that you're only joining out to the Votes

table if you really need to show its contents. Otherwise, you keep the query

fast by only looking at the Users table.

You can also leverage views here to put the optional joins and fields in the

view, and SQL Server will (usually) do a good job of eliminating unneccessary

joins, but that also means you can't just SELECT *. You have to be really

careful about only selecting the fields you need.

*/

/*

Debugging this stuff can be AWFUL, especially when you start nesting dynamic

SQL inside of dynamic SQL. It can even be tough figuring out which part of your

query is throwing the error - normal SQL, or dynamic SQL, since the line

numbers are worthless too. Run this, and try to figure out where the error is:

*/

CREATE OR ALTER PROC dbo.usp_SearchUsers
```

```
@SearchDisplayName NVARCHAR(100) = NULL,

@ShowTotalVotes BIT = 0,

@Debug_PrintQuery TINYINT = 0 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000), @Select NVARCHAR(4000),

        @From NVARCHAR(4000), @Where NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @Select = @crlf + N'/* usp_SearchUsers */' + @crlf + N' SELECT u.Id, u.DisplayName,
u.Location ' + @crlf;

SET @From = N' FROM dbo.Users u ' + @crlf;

SET @Where = N' WHERE ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @Where = @Where + N' AND DisplayName LIKE @SearchDisplayName ' + @crlf;

  IF @ShowTotalVotes = 1

    BEGIN

    SET @Select = @Select + N' , VotesCast = (SELECT SUM(1) FROM dbo.Votes v WHERE u.Id
= v.UserId) ' + @crlf;

    END



  /* Autobots, unite! */

  SET @StringToExecute = @Select + @From + @Where;

  IF @Debug_PrintQuery = 1

    PRINT @StringToExecute;

EXEC sp_executesql @StringToExecute,

N'@SearchDisplayName NVARCHAR(100)',
@SearchDisplayName;

END

GO

/* Test it: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @ShowTotalVotes = 0,
@Debug_PrintQuery = 1;

GO

/* OMG THIS IS TERRIBLE

And that line number is worthless.

Is that "AND" inside our regular query, or the dynamic SQL?

So when you build dynamic SQL, it can help to use different casing for your

keywords and variable names inside dynamic SQL, like this:

Regular query:

*/

DECLARE @MyVariable INT = 1;

SELECT DisplayName FROM dbo.Users WHERE Id = @MyVariable AND 1 = 1;

GO

/* Inside my dynamic SQL:  */

declare @myvariable INT = 1;

select DisplayName from dbo.Users where Id = @myvariable and 1 = 1;

GO

/* Let's change the casing on our stored procedure: */

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@ShowTotalVotes BIT = 0,
@Debug_PrintQuery TINYINT = 0 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000), @Select NVARCHAR(4000),

            @From NVARCHAR(4000), @Where NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @Select = @crlf + N'/* usp_SearchUsers */' + @crlf + N' select U.Id, U.DisplayName,
U.Location ' + @crlf;

SET @From = N' from dbo.Users U ' + @crlf;

SET @Where = N' where ' + @crlf;

IF @SearchDisplayName IS NOT NULL

SET @Where = @Where + N' and U.DisplayName like @searchdisplayname ' + @crlf;

    IF @ShowTotalVotes = 1

        BEGIN

        SET @Select = @Select + N' , VotesCast = (select sum(1) from dbo.Votes V where U.Id =
V.UserId) ' + @crlf;

        END


    /* Autobots, unite! */

    SET @StringToExecute = @Select + @From + @Where;

    IF @Debug_PrintQuery = 1

        PRINT @StringToExecute;

EXEC sp_executesql @StringToExecute,

N'@searchdisplayname nvarchar(100)',

@SearchDisplayName;

END
```

```
GO

/* So now when there's an error, it's easier to know where it's coming from: */

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @ShowTotalVotes = 0,
@Debug_PrintQuery = 1;

GO

/*

You may also need to log the dynamic SQL, and/or the parameters. Erik Darling

wrote a solution for that:

https://www.brentozar.com/archive/2017/04/fun-logging-dynamic-sql/

Basically you create a table like this:

*/

CREATE TABLE dbo.Logger

(

  Id BIGINT IDENTITY(1, 1),

  RunHash UNIQUEIDENTIFIER,

  UserName NVARCHAR(128),

  StartTime DATETIME2,

  EndTime DATETIME2,

  RunTimeSeconds AS DATEDIFF(SECOND, StartTime, EndTime),

  DynamicSQL NVARCHAR(MAX)

);

/* And in your stored procedures, include this code: */

DECLARE @RunHash UNIQUEIDENTIFIER = NEWID()

... /* query-building-goes-here */

INSERT dbo.Logger ( RunHash, UserName, StartTime, EndTime, DynamicSQL )
```

```
        VALUES(@RunHash, SUSER_SNAME(), SYSDATETIME(), NULL, @StringToExecute)

EXEC sys.sp_executesql 'Your lousy query'

UPDATE dbo.Logger

SET EndTime = SYSDATETIME()

WHERE RunHash = @RunHash

GO

/*

To show you just how bad it can get, I'm going to put all the logic in at once:

*/

CREATE OR ALTER PROC dbo.usp_SearchUsers

@SearchDisplayName NVARCHAR(100) = NULL,

@SearchLocation NVARCHAR(100) = NULL,

@SearchReputation INT = NULL,

@OrderBy NVARCHAR(100) = 'CreationDate',

@ShowTotalVotes BIT = 0,

@Debug_PrintQuery TINYINT = 0,

  @Debug_ExecuteQuery TINYINT = 1 AS

BEGIN

DECLARE @StringToExecute NVARCHAR(4000), @Select NVARCHAR(4000),

        @From NVARCHAR(4000), @Where NVARCHAR(4000), @Order NVARCHAR(4000);

DECLARE @crlf NVARCHAR(2) = NCHAR(13) + NCHAR(10);

SET @Select = @crlf + N'/* usp_SearchUsers */' + @crlf + N' select U.Id, U.DisplayName,
U.Location ' + @crlf;

SET @From = N' from dbo.Users U ' + @crlf;

SET @Where = N' where 1 = 1 ' + @crlf;
```

```
IF @SearchDisplayName IS NOT NULL

SET @Where = @Where + N' and DisplayName like @searchdisplayname ' + @crlf;

IF @SearchLocation IS NOT NULL

SET @Where = @Where + N' and Location like @searchlocation ' + @crlf;

IF @SearchReputation IS NOT NULL

SET @Where = @Where + N' and Reputation = @searchreputation ' + @crlf;

    IF @ShowTotalVotes = 1

        BEGIN

        SET @Select = @Select + N' , VotesCast = (select sum(1) from dbo.Votes V WHERE U.Id = V.UserId) ' + @crlf;

        END

    IF @OrderBy IS NOT NULL

        BEGIN

        SET @Order = N' order by ';

        SET @Order = @Order +

            CASE WHEN @OrderBy LIKE 'CreationDate%' THEN N' U.CreationDate '

                WHEN @OrderBy LIKE 'DisplayName%' THEN N' U.DisplayName '

                WHEN @OrderBy LIKE 'Location%' THEN N' U.Location '

                WHEN @OrderBy LIKE 'Reputation%' THEN N' U.Reputation '

                ELSE N' U.Id '   /* Or whatever default ordering you want, ideally to make SQL's life easier */

            END;

        IF @OrderBy LIKE '% DESC' /* You could also do this with a separate @OrderByDesc bit parameter if you want */

            SET @Order = @Order + N' desc ';
        END;

    SET @StringToExecute = @Select + @From + @Where + @Order;

    IF @Debug_PrintQuery = 1

        PRINT @StringToExecute;

    IF @Debug_ExecuteQuery = 1

    EXEC sp_executesql @StringToExecute,

    N'@searchdisplayname nvarchar(100), @searchlocation nvarchar(100),
@searchreputation int',

    @SearchDisplayName, @SearchLocation, @SearchReputation;

END

GO

/* Let's see how it works: */

DBCC FREEPROCCACHE;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Brent%', @SearchLocation = 'San Diego%',

                @ShowTotalVotes = 0, @OrderBy = 'DisplayName', @Debug_PrintQuery = 1;

GO

EXEC usp_SearchUsers @SearchLocation = 'Indiana%', @SearchReputation = 2,

                @ShowTotalVotes = 0, @OrderBy = 'CreationDate DESC', @Debug_PrintQuery =
1;

GO

EXEC usp_SearchUsers @SearchDisplayName = 'Lady Gaga%',

                @ShowTotalVotes = 1, @OrderBy = 'Location', @Debug_PrintQuery = 1;

GO

sp_BlitzCache;
```

/*

This here is the point where you wish that your conference water bottle was

filled with margaritas instead, and you start sobbing into your hands.

Dynamic SQL done well is really hard work, but before you complain, think about

a sliding control that you can tune left to right:

```
Fast code ____________________________________________________ Slow code
goes slow                                  goes fast
EF, NHibernate          Stored Procs             Dynamic SQL
```

Start with easy-to-code stuff like ORMs. Later, as you need better performance,

graduate to stored procedures. Over time, as your data grows and you have to

start making tradeoffs between hardware, index size, and budgets, you may have

to graduate to finely tuned dynamic SQL.

Now you're better equipped to start building it in a way that scales.

Wanna learn even more? Check out Erland Sommarskog's epic posts:

http://www.sommarskog.se/dynamic_sql.html

http://www.sommarskog.se/dyn-search.html