Developer Tips: Pagination – Data

By Andrew Hinkle

A common feature of most ecommerce sites is the standard product search. A typical implementation consists of three parts: data, user interface, and the communication between the two. For the data we'll use an Azure SQL database. Microsoft has simplified the process of creating a test database, so I'll just give some guidelines below. We'll create a stored procedure given some search parameters will retrieve the data and pagination information with performance and flexibility in mind. The user interface and the communication parts will be discussed in future articles.

Pagination is the concept of breaking up content across multiple pages with an easy to use navigation to traverse the pages. In this article are goal is to create the stored procedure that will return all of the information required to properly create that user interface. This includes the current page, entries per page, total entries, and total pages. Optionally and for diagnostics each row of the search results will include the current row, current page, and current row on page.

# Requirements to run examples

The following are required to run this script to create a stored procedure and run the sample calls under the testing section.

1. [SQL Server Management Studio (SSMS) 2017](https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017)
2. Azure Free Account or better
   1. <https://portal.azure.com>
3. [Create an Azure SQL database in the Azure portal](https://docs.microsoft.com/en-us/azure/sql-database/sql-database-get-started-portal) with the AdventureWorksLT
   1. Select source: Sample (AdventureWorksLT)
   2. Database name: AdventureWorksLT
   3. Pricing Tier: Basic
4. [ProductSearch stored procedure with examples](https://github.com/penblade/Tips/tree/master/Tips.Pagination/Scripts)
   1. This is the actual script this article references

# What are we paginating?

The AdventureWorksLT database has some product tables with some basic information and relationships such as product name, number, parent/child categories, and description. The description table adds some complication in that it stores a product description by six different cultural languages. Example: 'en' for English, 'fr' for French, etc. We're going to allow users to search by the cultural language (all or exact match), product name (like), product number (exact match), and product description (like).

We'll also allow them to request the current page and the number of entries per page. The current page must be passed in. The number of entries per page is optional as you could decide to default or hard code it. Some frontend developers prefer to get all the product search results and perform the pagination filtering, caching, and calculations in code. While I don't think it's very efficient, sometimes you don't have the technology or the abilities to implement it. To support this feature, if the user sends us 0 for the current page and entries per page we'll ignore pagination and return all of the results.

# Parameter Sniffing

I defer to Brent Ozar's great team on explaining "[the elephant and the mouse or parameter sniffing in sql server](https://www.brentozar.com/archive/2013/06/the-elephant-and-the-mouse-or-parameter-sniffing-in-sql-server/)". The short version is that SQL server will cache query plans based on the parameter. However, an optimized query plan for a "mouse" an estimated small result set may not be optimized for an "elephant" an estimated large result set. Parameter sniffing tries to save time by reusing query plans, but that is not always beneficial. Kendra Little explains the concept very well in her roughly 20 minute video in the same link.

By default you should not jump the gun and assume parameter sniffing is a problem and try to fix it. There are exceptions to the rule. If the stored procedure comes up in your top 10 of slow queries, analyze the query plan and try the more conventional troubleshooting first before trying to determine if you have a parameter sniffing issue. You may also want to try the following if you already have a very good idea that you'll have a performance issue such as with product searches that could return vastly different sizes in result sets.

From most invasive to least invasive:

1. WITH RECOMPILE;
   1. Force recompile every time. CPU intensive. Nuclear option. Could take longer than using the slower query plan.
2. Where clause with OPTION (RECOMPILE)
   1. Only when the where clause condition changes. Similar to RECOMPILE every time, however, if the value that is changing changes often, it can be just as nuclear.
3. OPTION (OPTIMIZE FOR ({@parameter} = value))
   1. If the data changes over time, this could cause issues in the future and may need updated to get more optimized plans. May involve monitoring and revisiting. Potentially higher maintenance.
4. OPTION (OPTIMIZE FOR UNKNOWN)
   1. Optimizes for any value. More generic query plan. Not best but usually very reasonable.
5. Assign parameters that you don't want sniffed to local variables. (Read the comments in the referenced article)
   1. Optimizes for any value. More generic query plan. Not best but usually very reasonable.
6. Do Nothing
   1. The Parameter Sniffing feature was added for a reason. Let it do its job until you have or will have an issue.

The small data sample in our test database would typically not warrant any parameter sniffing optimization, however, I want you to assume the database is filled with vast amounts of products. Given this is a product search I decided to use local variables. I've had great success using local variables in the past for optimizing generic search queries, so I usually pull the trigger and implement it in these scenarios.

# Pagination – Notes

Page number 0 returns all records to support applications that implement caching or alternate pagination. If you are getting all records, then the techniques used here are not as efficient.

The CurrentRow, CurrentPage, and CurrentRowOnPage are for diagnostics and may be commented out to further increase performance by storing and returning less information. The CurrentPage must still be maintained in the temp table.

The main concept used for pagination is to use the minimal amount of information to determine the search results storing only the most unique identifier(s) in the temp table. Given these unique ids join on all the tables to gather the information you need to include in the results.

The introduction of different product descriptions by culture added a challenge. I could have assumed that the culture would always be 'en' for English. However, when I was selecting all records I was getting odd CurrentRow results until I realized that I needed to store the culture in the temp table as well.

# Pagination – Order By Considerations

One more note before I move on. If you run one of the following test execute statements to include the actual execution plan you'll see a third of the time is dedicated to the sort. Further efforts could be made to add an index for the order by combination. Perhaps there is no business need to ever return result sets for more than one cultural language, so the order by may be simplified to just the product id. There are further considerations to optimize the cached query plan. I took liberties here to show a less efficient plan to demonstrate a slightly more complicated scenario with two Order By fields.

# Pagination – Test Execute Statements

I add the test execute statements at the top of the stored procedure comments, so other developers will know how to implement the stored procedure. They also serve the benefit of defining your Acceptance Criteria, verifying that you accomplished all of your goals.

/\*

TESTING:

-- Test: SET FMTONLY ON returns results sets.

-- If the SET FMTONLY OFF statement is removed

-- and you run the following statements, then

-- you'll get the error:

-- Invalid object name '#ProductSearchResults'.

SET FMTONLY ON

EXEC [dbo].[ProductSearch]

@CurrentPage = NULL

, @EntriesPerPage = NULL

, @Culture = NULL

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

SET FMTONLY OFF

-- Test: Defaults to all records

EXEC [dbo].[ProductSearch]

@CurrentPage = NULL

, @EntriesPerPage = NULL

, @Culture = NULL

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: English records only

EXEC [dbo].[ProductSearch]

@CurrentPage = NULL

, @EntriesPerPage = NULL

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage = 1 retrieves records 1 - 10

EXEC [dbo].[ProductSearch]

@CurrentPage = 1

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage = 2 retrieves records 11 - 20

EXEC [dbo].[ProductSearch]

@CurrentPage = 2

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage = 30 retrieves records 291 - 294

EXEC [dbo].[ProductSearch]

@CurrentPage = 30

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage = 31 retrieves no records,

-- because the page is greater than total pages

EXEC [dbo].[ProductSearch]

@CurrentPage = 31

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage 1; ProductName is like 'BIKE'

EXEC [dbo].[ProductSearch]

@CurrentPage = 1

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = 'BIKE'

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage 5; ProductName is like 'mountain'

EXEC [dbo].[ProductSearch]

@CurrentPage = 5

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = 'mountain'

, @ProductNumber = NULL

, @ProductDescription = NULL

-- Test: CurrentPage 2; ProductName is like 'mountain'

-- Product Description is like 'bike'

EXEC [dbo].[ProductSearch]

@CurrentPage = 2

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = 'mountain'

, @ProductNumber = NULL

, @ProductDescription = 'bike'

-- Test: CurrentPage 1; ProductNumber is exactly 'SO-B909-L'

EXEC [dbo].[ProductSearch]

@CurrentPage = 1

, @EntriesPerPage = 10

, @Culture = 'en'

, @ProductName = NULL

, @ProductNumber = 'SO-B909-L'

, @ProductDescription = NULL

\*/

# Pagination – Create stored procedure and assign to local variables

CREATE PROCEDURE [dbo].[ProductSearch] (

@CurrentPage INT

, @EntriesPerPage INT

, @Culture VARCHAR(6)

, @ProductName NVARCHAR(50)

, @ProductNumber NVARCHAR(25)

, @ProductDescription NVARCHAR(400)

)

AS

SET NOCOUNT ON

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

-- When "SET FMTONLY ON" is used by extensions like "Entity Framework Reverse POCO Generator"

-- Temp tables cause the error: "Invalid object name '#{temp table name}'"

-- To avoid the error and still return the result type formats, just turn the feature off.

-- The extension then creates all of the appropriate result models, even for multiple result sets.

SET FMTONLY OFF

-- Set parameters to local variables forcing the SQL query plan to

-- cache a generically optimized query plan instead of caching by value.

-- On average, this will perform better.

DECLARE

@localCurrentPage INT = @CurrentPage

, @localEntriesPerPage INT = @EntriesPerPage

DECLARE

@localCulture NVARCHAR(6) = @Culture

, @localProductName NVARCHAR(50) = @ProductName

, @localProductNumber NVARCHAR(25) = @ProductNumber

, @localProductDescription NVARCHAR(400) = @ProductDescription

-- Pagination Defaults

IF (@localCurrentPage IS NULL OR @localCurrentPage = '') SET @localCurrentPage = 0

IF (@localEntriesPerPage IS NULL OR @localEntriesPerPage = '') SET @localEntriesPerPage = 0

# Pagination – Create the ProductSearchResults temp table

-- Create a temp table to store the current page and product id of the search results.

IF OBJECT\_ID('tempdb..#ProductSearchResults') IS NOT NULL DROP TABLE #ProductSearchResults

CREATE TABLE dbo.#ProductSearchResults (CurrentRow INT, CurrentPage INT, CurrentRowOnPage INT, ProductID INT, Culture NVARCHAR(6))

-- Get the page number and product id for the search results.

-- Do not gather all of the fields that need returned at this time

-- as that would get way too much information that is never returned.

INSERT INTO #ProductSearchResults

SELECT

-- Get the row number and subtract 1 to make it 0-based to group 1-10 together

-- Divide by @localEntriesPerPage to get the 0-based page number

-- Add 1 to make the page number 1-based.

-- ex. 1 + ( 1 - 1) / 10 = 1 -- Row 1

-- ex. 1 + (10 - 1) / 10 = 1 -- Row 10

-- ex. 1 + (11 - 1) / 10 = 2 -- Row 11

-- Handle the division by 0 case.

-- The current row.

-- \*1\* The ORDER BY must match for all queries.

1 + (ROW\_NUMBER() OVER (ORDER BY p.Name, pmpd.Culture) - 1) AS CurrentRow

-- The current page.

, CASE WHEN @localEntriesPerPage > 0

-- \*1\* The ORDER BY must match for all queries.

THEN 1 + ((ROW\_NUMBER() OVER (ORDER BY p.Name, pmpd.Culture) - 1) / @localEntriesPerPage)

ELSE 0

END AS CurrentPage

-- The current row on the page.

, CASE WHEN @localEntriesPerPage > 0

-- \*1\* The ORDER BY must match for all queries.

THEN 1 + ((ROW\_NUMBER() OVER (ORDER BY p.Name, pmpd.Culture) - 1) % @localEntriesPerPage)

ELSE 0

END AS CurrentRowOnPage

-- While you could include the product name instead of the product id,

-- the NVARCHAR is not as performant as using an INT and a primary key

-- is preferred over a column that may not have been indexed.

, p.ProductID

-- The culture is required to get the distinct description associated

-- with the current row.

-- You could instead enforce that only a single culture may be used at a time,

-- which makes perfect since in this product search. This would allow you

-- to remove it from the temp table and use the local culture variable in

-- the queries. You would need to return an exception at the beginning

-- of the stored procedure to enforce that requirement.

, pmpd.Culture

FROM

SalesLT.Product p

INNER JOIN SalesLT.ProductModel pm ON p.ProductModelID = pm.ProductModelID

INNER JOIN SalesLT.ProductModelProductDescription pmpd ON pm.ProductModelID = pmpd.ProductModelID

INNER JOIN SalesLT.ProductDescription pd ON pmpd.ProductDescriptionID = pd.ProductDescriptionID

WHERE

-- Culture (exact)

(@localCulture IS NULL OR @localCulture = '' OR pmpd.Culture = @localCulture)

-- ProductName (like)

AND (@localProductName IS NULL OR @localProductName = '' OR p.Name LIKE '%' + @localProductName + '%')

-- ProductNumber (exact)

AND (@localProductNumber IS NULL OR @localProductNumber = '' OR p.ProductNumber = @localProductNumber)

-- ProductDescription (like) - This may not be culture compliant except for english.

AND (@localProductDescription IS NULL OR @localProductDescription = '' OR pd.Description LIKE '%' + @localProductDescription + '%')

-- \*1\* The ORDER BY must match for all queries.

ORDER BY p.Name, pmpd.Culture

# Pagination – Get the search results

-- Gather all of the fields to be returned for the requested subset of records.

SELECT

r.CurrentRow

, r.CurrentPage

, r.CurrentRowOnPage

, r.ProductID

, pmpd.Culture

, p.ProductNumber

, p.Name AS ProductName

, pd.Description AS ProductDescription

-- Take advantage of the fact that there will always be a

-- parent/child category relationship and is not further nested.

, pc2.Name AS ParentProductCategory

, pc1.Name AS ChildProductCategory

FROM #ProductSearchResults r

INNER JOIN SalesLT.Product p ON r.ProductID = p.ProductID

INNER JOIN SalesLT.ProductModel pm ON p.ProductModelID = pm.ProductModelID

INNER JOIN SalesLT.ProductModelProductDescription pmpd ON pm.ProductModelID = pmpd.ProductModelID

INNER JOIN SalesLT.ProductDescription pd ON pmpd.ProductDescriptionID = pd.ProductDescriptionID

-- Join on additional tables to get information that wasn't required to perform the search.

INNER JOIN SalesLT.ProductCategory pc1 ON p.ProductCategoryID = pc1.ProductCategoryID

INNER JOIN SalesLT.ProductCategory pc2 ON pc1.ParentProductCategoryID = pc2.ProductCategoryID

WHERE

-- Required if you want all records returned and you'll handle pagination in code.

(r.CurrentPage = @localCurrentPage OR @localCurrentPage = 0)

-- Use the culture from the temp table.

-- It's required to get the distinct product description by current row.

-- You could enforce the culture is provided to this stored procedure.

-- By doing so you could change this to use the local culture variable.

AND (r.Culture = pmpd.Culture)

-- \*1\* The ORDER BY must match for all queries.

ORDER BY p.Name, pmpd.Culture

# Pagination – Get the pagination information

-- Return pagination information.

SELECT

@localCurrentPage AS CurrentPage

, @localEntriesPerPage AS EntriesPerPage

, COUNT(r.ProductId) AS TotalEntries

, MAX(r.CurrentPage) AS TotalPages

FROM #ProductSearchResults r

# Pagination – Delete the temp table and end the stored procedure

IF OBJECT\_ID('tempdb..#ProductSearchResults') IS NOT NULL DROP TABLE #ProductSearchResults

GO

-- Depending on how you plan to implement this stored procedure

-- you should consider restricting access to it to only those

-- in specific roles, such as a company website role.

--GRANT EXECUTE ON [dbo].[ProductSearch] TO [CompanyWebsiteRole]

--GO

# Future Enhancements

You could further enhance this product search to include parameters such as parent or child product category name. Perhaps include products that are less than or greater than a specific list price, weight, sell date, etc.

# Entity Framework Reverse POCO Generator – SET FMTONLY OFF

I created a similar generic search stored procedure recently and used the T4 generator to create EF stored procedure calls and result set models. I ran into issues where the models were not getting generated. After reviewing many forum posts, I found that the extension determines the structure of the result set by executing the stored procedure with SET FMTONLY ON as follows.

SET FMTONLY ON

EXEC [dbo].[ProductSearch]

@CurrentPage = NULL

, @EntriesPerPage = NULL

, @Culture = NULL

, @ProductName = NULL

, @ProductNumber = NULL

, @ProductDescription = NULL

SET FMTONLY OFF

If you use a temp table in your stored procedure and run the statement you'll get the error.

Invalid object name '#ProductSearchResults'.

To avoid the issue, you just need to add the following line at the top of your stored procedure.

SET FMTONLY OFF

Now it works! I may work on some code samples illustrating this in a future article.

# Troubleshooting

While testing, if the CurrentRow, CurrentPage, and CurrentRowOnPage aren't in numerical order as expected for the filter, then you should verify that all of your ORDER BY statements are the same for each of your queries.

You could also be missing a field used in the order by or primary key alternative in the temp table. Ex. product name -> product id; culture was the gotcha for me in creating this stored procedure example.

# Conclusion

A lot of concepts were brought up on creating a flexible and optimized stored procedure that would return the search results only for the page requested and a second result set with the pagination information. Examples were provided on how to execute the procedure for multiple scenarios. Calculation details were included to help grasp how the pagination fields were determined and used. Known issues with Entity Framework Reverse POCO Generator were discussed along with the work around. Finally we ended with a troubleshooting tip.

How do you perform your pagination? Do you prefer using Entity Framework and accessing the tables directly? Do you prefer retrieving all of the search results and let WebForms manage the pagination? Are your pagination stored procedures performant and flexible? Let's discuss in the comments below.