Index Black Ops

How sys.dm_db_index_operational_stats collects and returns information

Jason Strate
www.twitter.com\stratesql
www.jasonstrate.com

Contents

Introduction

This document is a compilation of six blog posts written by Jason Strate. The post describes and investigates the DMV sys.dm_db_index_operational_stats. The content contained in this document is copyrighted by Jason Strate and may not be reused or redistributed without permission and attribution.

For more information on this and other clients, check out my blog at www.jasonstrate.com.

Biography

Jason Strate, Digineer Inc, is a database architect and administrator with over fifteen years of experience. He is a recipient of the Microsoft Most Valuable Professional (MVP) for SQL Server since July, 2009. His experience includes design and implementation of both OLTP and OLAP solutions as well as assessment and implementation of SQL Server environments for best practices, performance, and high availability solutions. Jason is a SQL Server MCITP and participated in the development of Microsoft Certification exams for SQL Server 2008.

Jason enjoys helping others in the SQL Server community and does this by presenting at technical conferences and user group meetings. Most recently, Jason has presented at the SSWUG Virtual Conferences, TechFuse, numerous SQL Saturdays, and at PASSMN user group meetings. Jason is also a contributing author for the Microsoft white paper “Empowering Enterprise Solutions with SQL Server 2008 Enterprise Edition.” Jason is an active blogger with a focus on SQL Server and related technologies.

Index Black Ops Part 1 - Locks and Blocking

As I mentioned in my TSQL2sDay index summary post, the next few posts will be on sys.dm_db_index_operational_stats and the information that the DMV contains. In this post, we are going to look at the locking and blocking columns.

Base Columns

Before we jump over to the meat and potatoes, let's first take a look at four columns that we'll be using to make sense of the data in the DMV.

- **database_id (smallint)** - The ID of the database. This can be translated with DB_NAME() and by querying sys.databases.
- **object_id (int)** - ID of the table or view This can be translated using sys.all_objects, sys.tables, or with OBJECT_NAME().
- **index_id (int)** - ID of the index or heap. Used in conjunction with object_id this can determine the view that is being referenced in sys.indexes.
- **partition_number (int)** - 1-based partition number within the index or heap. Every index has at least a single partition. Even if you aren't partitioning the index, the partition is there for you non-partitioned data.

Meat and Potato Columns

I said meat and potato columns, so here we are. These are the columns that you can

- **row_lock_count (bigint)** - Cumulative number of row locks requested.
- **row_lock_wait_count (bigint)** - Cumulative number of times the Database Engine waited on a row lock.
- **row_lock_wait_in_ms (bigint)** - Total number of milliseconds the Database Engine waited on a row lock.
- **page_lock_count (bigint)** - Cumulative number of page locks requested.
- **page_lock_wait_count (bigint)** - Cumulative number of times the Database Engine waited on a page lock.
- **page_lock_wait_in_ms (bigint)** - Total number of milliseconds the Database Engine waited on a page lock.

These are the columns that will provide the details at an individual index level on the blocking that is occurring. The locks report on the volume of activity on the index. The lock wait counts provide details on the rate in which the locks are being blocked. Finally the lock wait in ms will help establish the degree of severity that the locking is in regards to.

Get Your Query On

Before going much further let's build a couple queries that we can use to investigate locks and blocking. One query to get page locks and blocking percentages:

```
SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
        ,i.name AS index_name
        ,page_lock_count
        ,page_lock_wait_count
        ,CAST(100. * page_lock_wait_count / NULLIF(page_lock_count,0) AS
DECIMAL(6,
                                                                          2)) AS
page_block_pct
        ,page_lock_wait_in_ms
        ,CAST(1. * page_lock_wait_in_ms / NULLIF(page_lock_wait_count,0) AS
DECIMAL(12,
                                                                          2)) AS
page_avg_lock_wait_ms
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY row_lock_wait_count + page_lock_wait_count DESC
        ,row_lock_count + page_lock_count DESC
```

And another to get row locks and blocking percentages:

```
SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
        ,i.name AS index_name
        ,row_lock_count
        ,row_lock_wait_count
        ,CAST(100. * row_lock_wait_count / NULLIF(row_lock_count,0) AS
DECIMAL(6,
                                                                          2)) AS
row_block_pct
        ,row_lock_wait_in_ms
        ,CAST(1. * row_lock_wait_in_ms / NULLIF(row_lock_wait_count,0) AS
DECIMAL(12,
                                                                          2)) AS
row_avg_lock_wait_ms
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY row_lock_wait_count + page_lock_wait_count DESC
        ,row_lock_count + page_lock_count DESC
```

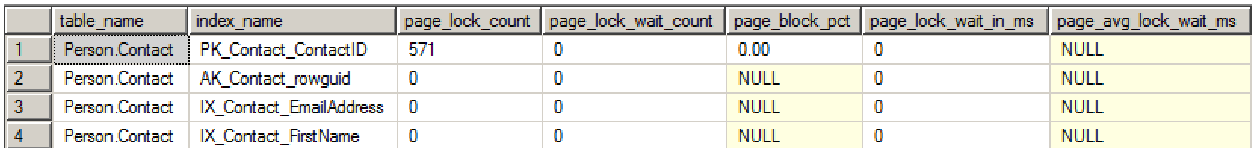
What Does It Mean?

At this point you are sitting there staring across the screen at me as though I am the one building the Devil's Tower out of mashed potatoes. I wouldn't have written this post if this didn't mean something. And it does… no really.

Let's look at how these locks are tabulated. First run a query that will return all of the results for a table. I'll be using AdventureWorks – because that's how I roll.

```
USE AdventureWorks
GO

SELECT   *
FROM     Person.Contact
```

Run the page lock query and the following results will be returned:



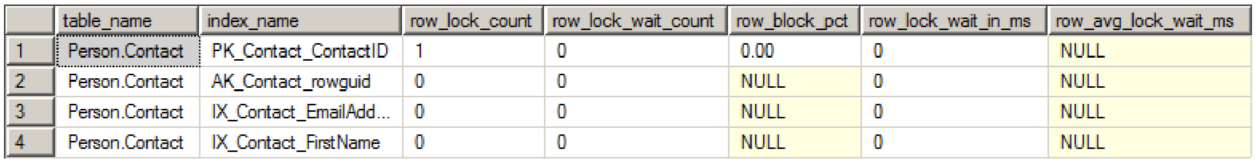
	table_name	index_name	page_lock_count	page_lock_wait_count	page_block_pct	page_lock_wait_in_ms	page_avg_lock_wait_ms
1	Person.Contact	PK_Contact_ContactID	571	0	0.00	0	NULL
2	Person.Contact	AK_Contact_rowguid	0	0	NULL	0	NULL
3	Person.Contact	IX_Contact_EmailAddress	0	0	NULL	0	NULL
4	Person.Contact	IX_Contact_FirstName	0	0	NULL	0	NULL

As you can see, returning all rows resulted in page locks as the query placed a lock on each page to return the data.

Change the query a bit to only return a single row.

```
USE AdventureWorks
GO

SELECT   *
FROM     Person.Contact
WHERE    ContactID = 1
```

Run the row lock query and the following results will be returned:



	table_name	index_name	row_lock_count	row_lock_wait_count	row_block_pct	row_lock_wait_in_ms	row_avg_lock_wait_ms
1	Person.Contact	PK_Contact_ContactID	1	0	0.00	0	NULL
2	Person.Contact	AK_Contact_rowguid	0	0	NULL	0	NULL
3	Person.Contact	IX_Contact_EmailAdd...	0	0	NULL	0	NULL
4	Person.Contact	IX_Contact_FirstName	0	0	NULL	0	NULL

This time the single row returned resulted in a row lock on a single row.

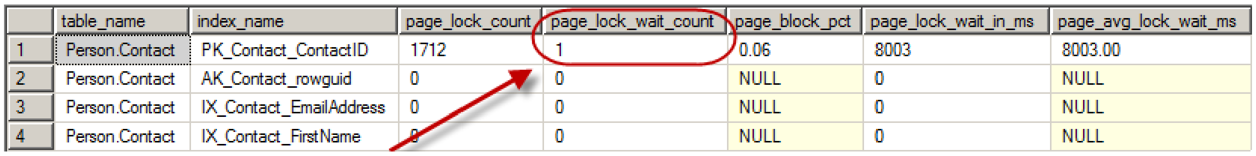
What About Blocking?

Copy the following query text into another query window and execute it.

```
BEGIN TRAN
UPDATE   Person.Contact WITH (PAGLOCK)
```

```
SET     NameStyle = NameStyle

WAITFOR DELAY '00:00:10'
COMMIT TRAN
```

Go back to the original query window and execute the SELECT query without the WHERE clause above. When it finishes execute the page lock query and the following results will be returned:



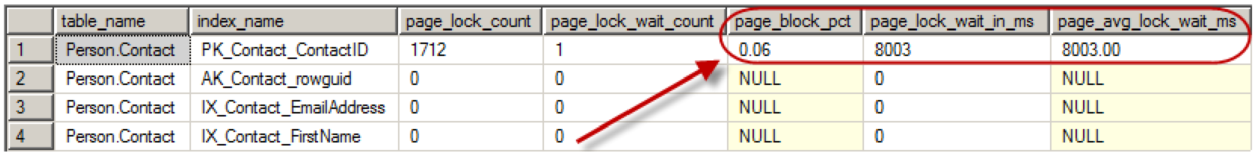
	table_name	index_name	page_lock_count	page_lock_wait_count	page_block_pct	page_lock_wait_in_ms	page_avg_lock_wait_ms
1	Person.Contact	PK_Contact_ContactID	1712	1	0.06	8003	8003.00
2	Person.Contact	AK_Contact_rowguid	0	0	NULL	0	NULL
3	Person.Contact	IX_Contact_EmailAddress	0	0	NULL	0	NULL
4	Person.Contact	IX_Contact_FirstName	0	0	NULL	0	NULL

Now this time we have some blocking. The query had a 10 second wait (and the 3 seconds that I took to get the other query started) – which resulted in about 7 seconds of waits on the index PK_Contact_ContactID. So for that who time, nobody could access or edit any of the rows in the that were locked by the UPDATE statement.

Now take the query below and execute it in another query window.

```
BEGIN TRAN
UPDATE  Person.Contact
SET     NameStyle = NameStyle
WHERE   ContactID = 1

WAITFOR DELAY '00:00:10'
COMMIT TRAN
```

As before, go back to the original query window and execute the SELECT query with the WHERE clause. And when it finishes, execute the row lock query for (you guessed it) the results below:



	table_name	index_name	page_lock_count	page_lock_wait_count	page_block_pct	page_lock_wait_in_ms	page_avg_lock_wait_ms
1	Person.Contact	PK_Contact_ContactID	1712	1	0.06	8003	8003.00
2	Person.Contact	AK_Contact_rowguid	0	0	NULL	0	NULL
3	Person.Contact	IX_Contact_EmailAddress	0	0	NULL	0	NULL
4	Person.Contact	IX_Contact_FirstName	0	0	NULL	0	NULL

And as you could have guessed, this will show the wait on the single row and then an accumulation of time on the index as well.

Conclusion

In the queries above, I've demonstrated how the locking and waits on indexes are tabulated. The effect of these were shown in the queries. Hopefully, you've seen through these examples how you can use sys.dm_db_index_operational_stats to identify indexes where locking pressure is occuring.

Relieving locking pressure isn't always the easiest thing to do. But it generally boils down to:

- Reviewing queries utilizing the index to determine if they are performing optimally
- Reviewing indexes to determine if you have the proper indexing in place

Two very broad areas, but by using the information above you can identify which indexes to look at and hone in on issues as they start arising in your index usage patterns.

Index Black Ops Part 2 - Page IO Latch, Page Latch

As I mentioned in my TSQL2sDay index summary post, the I am writing a few posts on the information that is contained in sys.dm_db_index_operational_stats. The posts will be the following:

- Index Black Ops Part 1 – Locking and Blocking
- Index Black Ops Part 2 – Page IO Latch, Page Latch
- Index Black Ops Part 3 – Index Usage
- Index Black Ops Part 4 – Index Overhead and Maintenance
- Index Black Ops Part 5 – Page Splits
- Index Black Ops Part 6 – Fill Factor vs. Page Splits

Today's post is the second in the series and will discuss the page IO and page latch information that is available. We'll cover what this information is. Afterwards we will look at querying the DMV for the data. And finally wrapping up with some idea on how you can use this data.

What Are Latches?

If you aren't familiar with latches, they are very similar to locks within SQL Server. Though they are less intrusive and are design in a manner that prevents locking collisions, such as deadlocks. While locks control and manage the state of data in a SQL Server database, the latches will control and manage the location of the data.

- **Page Latch** – These are waits that occur when a worker needs to wait for a page to become available. This typically occurs on a page is already available in memory.
- **Page IO Latch** – These are waits that occur when a needs to wait for a page due to physical I/O. Such as when a page needs to be made available in the buffer pool for reading or writing and SQL Server needs to retrieve it from disk.

Like locks, latches aren't a bad thing. Latches waiting excessively on other latches though… well too much of anything can be a bad thing.

Indexes with Latches

There are a few columns in sys.dm_db_index_operational_stats that are important when investigating latches on indexes. These columns are:

- **page_latch_wait_count (bigint)** - Cumulative number of times the Database Engine waited, because of latch contention.
- **page_latch_wait_in_ms (bigint) -** Cumulative number of milliseconds the Database Engine waited, because of latch contention.
- **page_io_latch_wait_count (bigint)** - Cumulative number of times the Database Engine waited on an I/O page latch.

- **page_io_latch_wait_in_ms (bigint)** - Cumulative number of milliseconds the Database Engine waited on a page I/O latch.
- **tree_page_latch_wait_count (bigint)** - Subset of page_latch_wait_count that includes only the upper-level B-tree pages. Always 0 for a heap.
- **tree_page_latch_wait_in_ms (bigint)** - Subset of page_latch_wait_in_ms that includes only the upper-level B-tree pages. Always 0 for a heap.
- **tree_page_io_latch_wait_count (bigint)** - Subset of page_io_latch_wait_count that includes only the upper-level B-tree pages. Always 0 for a heap.
- **tree_page_io_latch_wait_in_ms (bigint)** - Subset of page_io_latch_wait_in_ms that includes only the upper-level B-tree pages. Always 0 for a heap.

Now we won't be looking at any queries with the tree latches. That's something that can be covered in a more in-depth conversation. But suffice to say, want generally applies at the leaf level also applies at the tree level.

Page IO Latch Demo

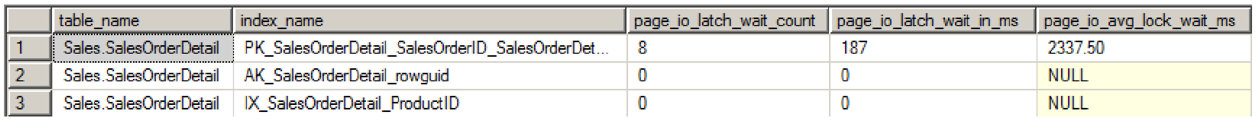
We'll start with a demo that will generate some page IO latch waits. The query I am using to generate page IO latches is below. All it is going to measure is the time that SQL Server spent waiting for the pages it needs to be read from disk. If the disk on the server you are using is great compared to my laptop drive you may see different results.

```
USE AdventureWorks
GO

SELECT  *
FROM    Sales.SalesOrderDetail
```

Now we'll run the query below to investigate the effect that the load had on the database:

```
SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,page_io_latch_wait_count
       ,page_io_latch_wait_in_ms
       ,CAST(100. * page_io_latch_wait_in_ms /
NULLIF(page_io_latch_wait_count,
                                                                 0) AS DECIMAL(12,2)) AS
page_io_avg_lock_wait_ms
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY 5 DESC
```

If you don't see the waits appear, try running multiple copies of the query or use a table that is larger. That will do it for you. The results of the query should look similar to the following:



	table_name	index_name	page_io_latch_wait_count	page_io_latch_wait_in_ms	page_io_avg_lock_wait_ms
1	Sales.SalesOrderDetail	PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	8	187	2337.50
2	Sales.SalesOrderDetail	AK_SalesOrderDetail_rowguid	0	0	NULL
3	Sales.SalesOrderDetail	IX_SalesOrderDetail_ProductID	0	0	NULL

As the picture indicates, there were page IO latch waits accumulated on the index **PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID**. These occurred on the concurrent executions of the query above while they wait for the first query to load the data it needed into the buffer pool. Since they were all going after the same data, they needed to wait instead of loading a copy for their use.

Page Latch Demo

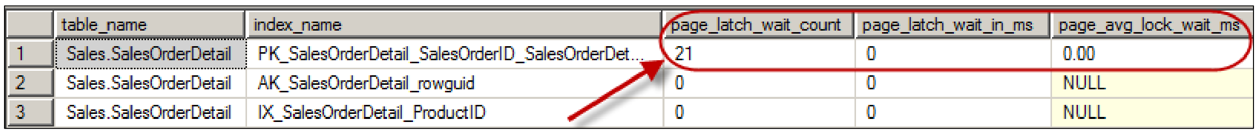
As I mentioned in the introduction, page latches occur when a worker is picking up a page for an operation. To generate some page latches we'll execute an update statement. And as before, the shoddy resources on my laptop will likely generate page latches. You results may vary, but if they do just increase the volume of the query.

```
UPDATE  sod
SET     UnitPrice = UnitPrice * 1.05
FROM    Sales.SalesOrderDetail sod
```

Now we'll run the query below to investigate the effect that the load had on the database:

```
SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,page_latch_wait_count
       ,page_latch_wait_in_ms
       ,CAST(100. * page_latch_wait_in_ms / NULLIF(page_latch_wait_count,0)
AS DECIMAL(12,
                                                               2)) AS
page_avg_lock_wait_ms
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY 5 DESC
```

The results of the query should look similar to the following:



	table_name	index_name	page_latch_wait_count	page_latch_wait_in_ms	page_avg_lock_wait_ms
1	Sales.SalesOrderDetail	PK_SalesOrderDetail_SalesOrderID_SalesOrderDet...	21	0	0.00
2	Sales.SalesOrderDetail	AK_SalesOrderDetail_rowguid	0	0	NULL
3	Sales.SalesOrderDetail	IX_SalesOrderDetail_ProductID	0	0	NULL

Like before, the effect of the page latches are seen above.

Life, The Universe, and Everything

Great! With the scripts above I've demonstrated how you can identify indexes that have high latch occurrences and the amount of time spent on them. It's like I'm a computer in front of large crowd only will to say 42, over and over again. Not exactly useful but we know it means something.

When will you know that you need to take a look at latches? What situations will happen when knowing which index that latches are occurring on will these scripts assist you.

There are a number of reasons you may want to look into these. If you have a high volume of page IO latches on an index you may wonder why queries need to go to disk so often to to get data for the index. Is this a memory issue where data is being pushed out? Is there a report problem where a ad-hoc report is flooding the buffer pool with data that the OLTP system doesn't need for its regular processing?

As I work through these kinds of issues with client, I plan to write more about them to share what I learn. In the meantime, I'd recommend reading SQL Server 2005 Waits and Queues and Troubleshooting Performance Problems in SQL Server 2008. These are a couple excellent white papers that can get anyone going when it comes knowing when to look into latch issues.

Index Black Ops Part 3 - Index Usage

Last week, I mentioned in my TSQL2sDay index summary post, that I'd be writing a few posts on the information that is contained in sys.dm_db_index_operational_stats. The posts will be the following:

- Index Black Ops Part 1 – Locking and Blocking
- Index Black Ops Part 2 – Page IO Latch, Page Latch
- Index Black Ops Part 3 – Index Usage
- Index Black Ops Part 4 – Index Overhead and Maintenance
- Index Black Ops Part 5 – Page Splits
- Index Black Ops Part 6 – Fill Factor vs. Page Splits

We're up to part three now and according to the schedule, this should have been published last Thursday. Those that were at SQL Saturday #50 heard about the laptop issues that I had and I'm going to just blame that for the delay. But now that I am back in action, on to part 3. For the current post, we are going to look at the index usage data that is contained in sys.dm_db_index_operational_stats.

Index Usage

The goal today is to find out how much use an index is providing to your database through DMVs. This information can be used to provide valuable insight into the value of the index. It's an easy calculation, if the index is used a lot more than other indexes on the table. Then it's probably a "better" index to have on the system.

There are a number of columns that we'll look at when it comes to reviewing index usage:

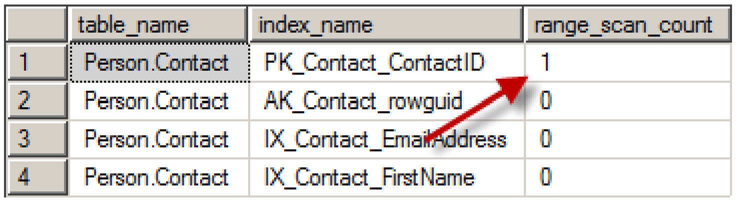
- **range_scan_count (bigint)** - Cumulative count of range and table scans started on the index or heap.
- **singleton_lookup_count (bigint)** - Cumulative count of single row retrievals from the index or heap.
- **leaf_insert_count (bigint)** - Cumulative count of leaf-level inserts.
- **leaf_delete_count (bigint)** - Cumulative count of leaf-level deletes.
- **leaf_update_count (bigint)** - Cumulative count of leaf-level updates.
- **leaf_ghost_count (bigint)** - Cumulative count of leaf-level rows that are marked as deleted, but not yet removed. These rows are removed by a cleanup thread at set intervals.
- **nonleaf_insert_count (bigint)** - Cumulative count of inserts above the leaf level.
- **nonleaf_delete_count (bigint)** - Cumulative count of deletes above the leaf level.
- **nonleaf_update_count (bigint)** - Cumulative count of updates above the leaf level.

Range Scan

The first column that we'll look at is range_scan_count. As noted above, this column collects a count of seek and scan operations that occur against an index. To illustrate execute the following query:

```
USE AdventureWorks
GO

SELECT TOP 100
        *
FROM    Person.Contact
GO

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,ios.range_scan_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY ios.range_scan_count DESC
```

The results for the second query should look like the following:



	table_name	index_name	range_scan_count
1	Person.Contact	PK_Contact_ContactID	1
2	Person.Contact	AK_Contact_rowguid	0
3	Person.Contact	IX_Contact_EmailAddress	0
4	Person.Contact	IX_Contact_FirstName	0

As you can see, for the 100 rows returned from query against Person.Contact there is an increment for 1 for the index that was used in sys.dm_db_index_operational_stats.

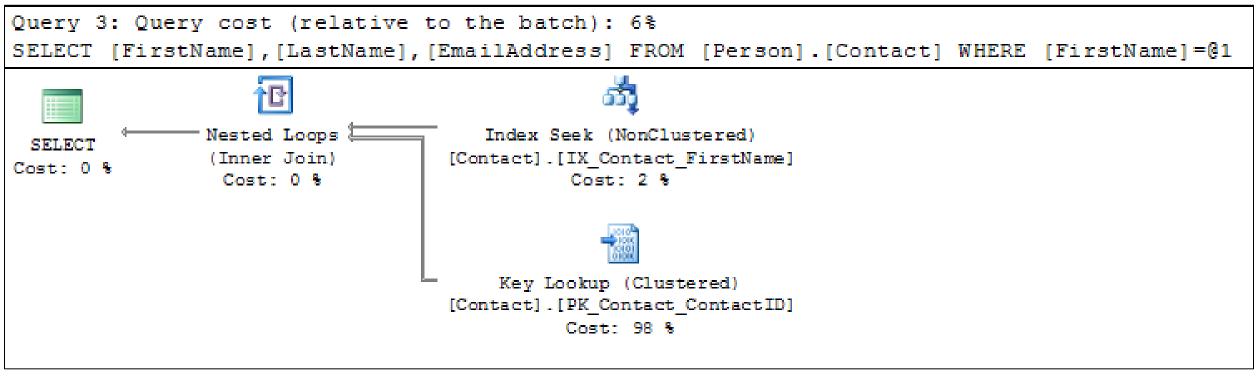
Singleton Lookup

The next column is to look at is the singleton_lookup_count. This column provides a count of the number of rows retrieved by a RowID and Key lookups. To demonstrate how this is accumulated, create the index below and run the accompanying queries. Turn on the Actual Execution Plan option for the query to view the execution included below as well.

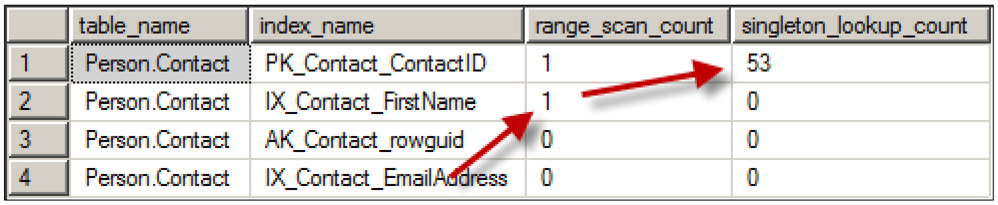
```
USE AdventureWorks
GO

IF EXISTS ( SELECT  *
            FROM    sys.indexes
            WHERE   name = 'IX_Contact_FirstName' )
    DROP INDEX IX_Contact_FirstName ON Person.Contact
GO

CREATE INDEX IX_Contact_FirstName ON Person.Contact(FirstName)
GO

SELECT  FirstName
       ,LastName
       ,EmailAddress
```

```
FROM     Person.Contact
WHERE    FirstName = 'Adam'
GO

SELECT   OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
        ,i.name AS index_name
        ,ios.range_scan_count
        ,ios.singleton_lookup_count
FROM     sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
         INNER JOIN sys.indexes i
             ON i.object_id = ios.object_id
                AND i.index_id = ios.index_id
WHERE    OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY ios.range_scan_count DESC
GO
```

If your Adventureworks database is similar to mine, then the execution plan for the SELECT statement on Person.Contact will look like the following:


```
Query 3: Query cost (relative to the batch): 6%
SELECT [FirstName],[LastName],[EmailAddress] FROM [Person].[Contact] WHERE [FirstName]=@1
```
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[Contact].[IX_Contact_FirstName]
Cost: 2 %
Key Lookup (Clustered)
[Contact].[PK_Contact_ContactID]
Cost: 98 %

The results of the SELECT will return 53 rows where first name is "Adam", not shown. The second result set for the DMV query will return results as below:



	table_name	index_name	range_scan_count	singleton_lookup_count
1	Person.Contact	PK_Contact_ContactID	1	53
2	Person.Contact	IX_Contact_FirstName	1	0
3	Person.Contact	AK_Contact_rowguid	0	0
4	Person.Contact	IX_Contact_EmailAddress	0	0

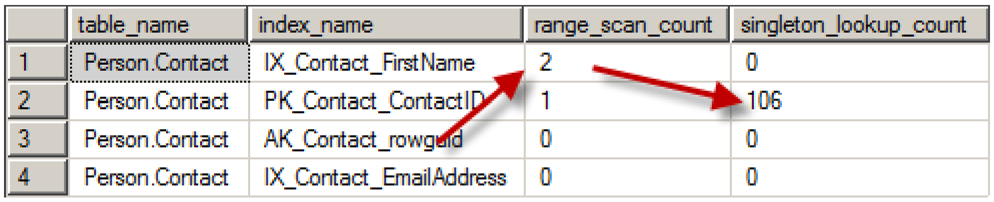
In this case, the range_scan_count incremented by 1 for the index seek operation on IX_Contact_FirstName. The count for singleton_lookup_count increased by 53 for the lookups against PK_Contact_ContactID.

While this appears to be a count of rows returned from by the Key Lookup operation, it is actually a count of the number of operations that occurred against the index.

To help show this, run the following set of queries:

```
USE AdventureWorks
GO

WITH    cContact
          AS (
              SELECT    FirstName
                       ,LastName
                       ,EmailAddress
              FROM      Person.Contact
              WHERE     FirstName = 'Adam')
    SELECT  FirstName
           ,LastName
           ,EmailAddress
    FROM    cContact
    WHERE   LastName = 'Reynolds'

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,ios.range_scan_count
       ,ios.singleton_lookup_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY ios.range_scan_count DESC
GO
```

Through the use of the CTE, the SELECT results are reduced from 53 rows to a single row being returned. Operationally, though, the lookup executed 53 times before the final results were filtered to where last name equals “Reynolds”.



	table_name	index_name	range_scan_count	singleton_lookup_count
1	Person.Contact	IX_Contact_FirstName	2	0
2	Person.Contact	PK_Contact_ContactID	1	106
3	Person.Contact	AK_Contact_rowguid	0	0
4	Person.Contact	IX_Contact_EmailAddress	0	0

After this execution, the singleton_lookup_count increased again by 53 instead of by 1 for the number of lookups that were returned by the query. This means that the results of the query against sys.dm_db_index_operational_stats are for the number of operations, not the number of rows returned by the final result set.

Index Modifications

Up to this point, we’ve investigate the columns that return results based on the number of operations that have occurred due to SELECT statements. Now the focus will be shifted to DML operations. For these operations, we’ll look at the columns leaf_insert_count, leaf_update_count, leaf_delete_count, and leaf_ghost_count.

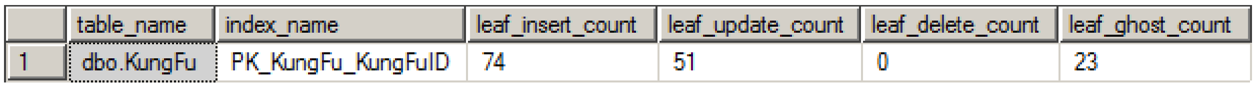
The following script will help demonstrate how DML operations will accumulate in the DMV:

```
USE AdventureWorks
GO

IF OBJECT_ID('dbo.KungFu') IS NOT NULL
    DROP TABLE dbo.KungFu
GO

CREATE TABLE dbo.KungFu
(
 KungFuID INT
,Hustle BIT
,CONSTRAINT PK_KungFu_KungFuID PRIMARY KEY CLUSTERED (KungFuID)
)
GO

INSERT  INTO dbo.KungFu
        SELECT  ROW_NUMBER() OVER (ORDER BY t.object_id)
               ,t.object_id % 2
        FROM    sys.tables t
GO

DELETE  FROM dbo.KungFu
WHERE   Hustle = 0
GO

UPDATE  dbo.KungFu
SET     Hustle = 0
WHERE   Hustle = 1
GO

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,ios.leaf_insert_count
       ,ios.leaf_update_count
       ,ios.leaf_delete_count
       ,ios.leaf_ghost_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),NULL,NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id
WHERE   OBJECTPROPERTY(ios.object_id,'IsUserTable') = 1
ORDER BY ios.range_scan_count DESC
```

Here's a basic walk through the script:

1. Create a table name KungFu
2. Add 73 rows to KungFu
3. Remove 23 rows from KungFu
4. Update 50 rows in KungFu

The results of the sys.dm_db_index_operational_stats query should look like the following:



	table_name	index_name	leaf_insert_count	leaf_update_count	leaf_delete_count	leaf_ghost_count
1	dbo.KungFu	PK_KungFu_KungFuID	74	51	0	23

A couple obvious items in the results are the inserts and updates against the index. Leaf_insert_count and leaf_update_count contain this information, respectively.

The next couple columns contain the delete row information. The first column contains the leaf pages that have been deleted and those that are scheduled for deletion but are still part of the index tree.

The non-leaf columns listed at the beginning haven't been discussed in this post. The columns tabulate information almost exactly as the columns for the leaf columns except that they are for the non-leaf pages of the index.

What About sys.dm_db_index_usage_stats?

The first thing that may have crossed your mind with the topic of index usage may have been the other DMV that is often used – namely sys.dm_db_index_usage_stats. This DMV is all well and good but it doesn't provide the full picture of the impact that plans are having on an index.

As you may know, sys.dm_db_index_usage_stats returns results based on the number of plans that have been executed that will utilize an index. The stark difference with sys.dm_db_index_operational_stats is that it returns the number of operations that have actually occurred in those same plans.

Summary

Hopefully this information has been helpful in providing some insight into how this DMV works. Please feel free to leave comments on this DMV below.

Index Black Ops Part 4 - Index Overhead and Maintenance

In my TSQL2sDay index summary post, that I'd be writing a few posts on the information that is contained in sys.dm_db_index_operational_stats. The posts will be the following:

- Index Black Ops Part 1 – Locking and Blocking
- Index Black Ops Part 2 – Page IO Latch, Page Latch
- Index Black Ops Part 3 – Index Usage
- Index Black Ops Part 4 – Index Overhead and Maintenance
- Index Black Ops Part 5 – Page Splits
- Index Black Ops Part 6 – Fill Factor vs. Page Splits

I'm a bit behind – so without further ado on to part four – index overhead and maintenance. For the current post, we are going to look at the data that is contained in sys.dm_db_index_operational_stats that can be used to measure index overhead and maintenance.

Index Overhead

To measure overhead on an index we'll be looking at the writes at the leaf and non-leaf levels for an index. These are measured with some of the same columns that were used in the last post, Index Black Ops Part 3 – Index Usage. These columns are:

- **leaf_insert_count (bigint)** – Cumulative count of leaf-level inserts.
- **leaf_delete_count (bigint)** – Cumulative count of leaf-level deletes.
- **leaf_update_count (bigint)** – Cumulative count of leaf-level updates.
- **leaf_ghost_count (bigint)** – Cumulative count of leaf-level rows that are marked as deleted, but not yet removed. These rows are removed by a cleanup thread at set intervals.
- **nonleaf_insert_count (bigint)** – Cumulative count of inserts above the leaf level.
- **nonleaf_delete_count (bigint)** – Cumulative count of deletes above the leaf level.
- **nonleaf_update_count (bigint)** - Cumulative count of updates above the leaf level.

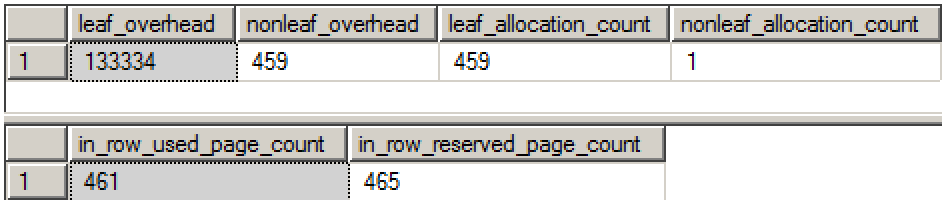
For overhead, we want to consider the amount of writes that occur on an index to identify those that SQL Server is spending more time writing to. A high number of writes does not indicate a negative amount of overhead. It only informs on where the focus of write activity is occurring. To demonstrate these columns in action we'll run the script listed below. This script will execute the following activity:

1. Create a table named dbo.IndexMaintenance
2. Insert 100,000 rows into dbo.IndexMaintenance
3. Update 33,334 rows in dbo.IndexMaintenance
4. Query sys.dm_db_index_operational_stats to demonstrate the number of write operations
5. Query sys.dm_db_partition_stats to demonstrate the number of pages allocated to the index.

Here is the needed script:

```
USE tempdb ;
GO

IF OBJECT_ID('dbo.IndexMaintenance') IS NOT NULL
    DROP TABLE dbo.IndexMaintenance ;

CREATE TABLE dbo.IndexMaintenance
(
 ID INT
,Value UNIQUEIDENTIFIER
,CreateDate DATETIME
,CONSTRAINT PK_IndexMaintenance PRIMARY KEY CLUSTERED (ID)
) ;

WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.IndexMaintenance
SELECT TOP (100000)
        n
       ,NEWID()
       ,GETDATE()
FROM    nums
ORDER BY n ;

UPDATE  dbo.IndexMaintenance
SET     Value = NEWID()
WHERE   ID % 3 = 1 ;

SELECT  leaf_insert_count + leaf_delete_count + leaf_update_count
        + leaf_ghost_count AS leaf_overhead
       ,nonleaf_insert_count + nonleaf_delete_count + nonleaf_update_count AS
nonleaf_overhead
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.IndexMaintenance'),
                                          NULL,NULL)

SELECT  in_row_used_page_count
       ,in_row_reserved_page_count
FROM    sys.dm_db_partition_stats
WHERE   object_id = OBJECT_ID('dbo.IndexMaintenance')
```

If the script ran properly, you should have received the following output:



	leaf_overhead	nonleaf_overhead	leaf_allocation_count	nonleaf_allocation_count
1	133334	459	459	1

	in_row_used_page_count	in_row_reserved_page_count
1	461	465

As the output shows, there were 133,334 writes at the leaf level on the index for dbo.IndexMaintenance. These writes were 100,000 inserts and 33,334 updates. At the non-leaf level where were 459 writes. These non-leaf writes corelate to the 459 leaf pages that were allocated for the index. Take those pages and add in the first page of the index and then the non-leaf page allocated for the index. With that you get the number of pages allocated in sys.dm_db_partition_stats.

The question is, of course, how does one use this information. The trouble here is that it depends. The index overhead is going to help highlight indexes that have a lot of write activity against them. Being that the operations happen at the row level, you can translate these operations to the number of rows affected.

We know from above that there were 133,334 writes happened on the index in the demonstration script. Is this bad? Or is this good? This is where the use of the index and the purpose of the database will be important information. When one index is updated more frequently than others it will be important to understand why. Is the index built on a volatile column that maybe should be indexed and referenced differently? Or is there a column included in the index that maybe should be removed to reduce write activity?

These and other questions will be at the purview of your database. The benefit of looking at index overhead is that these spot of high overhead can be deduced from the system.

Index Maintenance

Now that we've looked into overhead, we should also look at the maintenace that occurs on an index. There are times when gaps open up inside indexes during delete operations and whole pages of data are no longer populated with data. This type of activity is tracked through two more columns:

- **leaf_page_merge_count (bigint)** - Cumulative count of page merges at the leaf level.
- **nonleaf_page_merge_count (bigint)** - Cumulative count of page merges above the leaf level.

Let's continue the demonstration above and execute the following statements against the table already created:

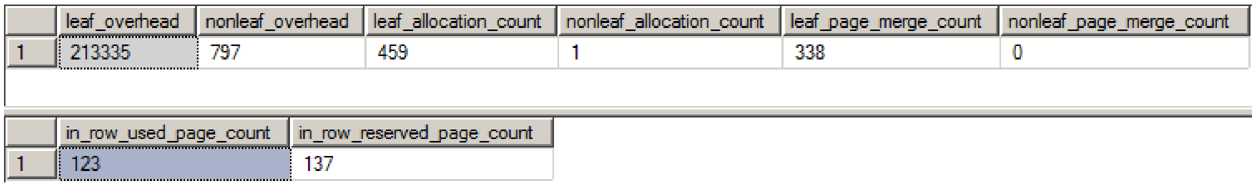
```
USE tempdb ;
GO

DELETE   FROM dbo.IndexMaintenance
WHERE    ID BETWEEN 10000 AND 90000

SELECT   leaf_insert_count + leaf_delete_count + leaf_update_count
         + leaf_ghost_count AS leaf_overhead
```

```
        ,nonleaf_insert_count + nonleaf_delete_count + nonleaf_update_count AS
nonleaf_overhead
        ,leaf_allocation_count
        ,nonleaf_allocation_count
        ,leaf_page_merge_count
        ,nonleaf_page_merge_count
FROM     sys.dm_db_index_operational_stats(DB_ID(),
                                           OBJECT_ID('dbo.IndexMaintenance'),
                                           NULL,NULL)

SELECT   in_row_used_page_count
        ,in_row_reserved_page_count
FROM     sys.dm_db_partition_stats
WHERE    object_id = OBJECT_ID('dbo.IndexMaintenance')
```

This script added executed the following actions:

1. Deleted 80,000 rows from dbo.IndexMaintenance
2. Query sys.dm_db_index_operational_stats to demonstrate the number of pages deallocated from the index.
3. Query sys.dm_db_partition_stats to demonstrate the number of pages allocated to the index.

As a result of the script, the following query results should be displayed:



	leaf_overhead	nonleaf_overhead	leaf_allocation_count	nonleaf_allocation_count	leaf_page_merge_count	nonleaf_page_merge_count
1	213335	797	459	1	338	0

	in_row_used_page_count	in_row_reserved_page_count
1	123	137

As you can see in the leaf_page_merge_count column there were 338 leaf pages in the index merged into other pages. This number does represent the difference in the number of pages allocated to the index but rather as I stated the operations that occurred merging the pages.

This information has been useful with my client in finding indexes where large numbers of rows are being removed from an index leading to deallocation of pages. After the deallocation, these pages may be reallocated as new rows for this or another index – and which will contribute to index fragmentation.

Index Read Overhead

In this last section, I would want to provide some secret sauce on a way to count all of the reads that have occurred on an index. Unfortunately at this time, SQL Server doesn't have any DMVs, that I know of, that provide this information.

The benefit of this would be that with counts on both reads and writes on an index the true overhead of an index can be determined. For instance, if the database is writing to substantially more rows than are being read - the performance hit on the writes may outweigh the benefit of the performance improvement the provided to the reads.

This information would be extremely beneficial and so much so, that if you agree I'd like to have you vote it up on Microsoft Connect.

Index Black Ops Part 5 - Page Splits

In my TSQL2sDay index summary post, that I'd be writing a few posts on the information that is contained in sys.dm_db_index_operational_stats. The posts are the following:

- Index Black Ops Part 1 – Locking and Blocking
- Index Black Ops Part 2 – Page IO Latch, Page Latch
- Index Black Ops Part 3 – Index Usage
- Index Black Ops Part 4 – Index Overhead and Maintenance
- Index Black Ops Part 5 – Page Splits
- Index Black Ops Part 6 – Fill Factor vs. Page Splits

For the current post, we are going to look at page splits. A page split occurs with a data page needs to be updated but the amount of data being placed back on the page exceeds the amount of room. To make room for the increased amount of data, SQL Server will add a new page to the index and move a portion of the data from the existing page to the new page.

Unfortunately, when page splits occur they create writes and locks that can sometimes affect the concurrency of the database. SQL Server has to write to new pages and lock the existing pages which can lead to performance issues in some systems.

Tracking Page Splits

For the current post, though, I'm not going to be discussing the how and why of page splits. The goal is to show where page splits cause values in sys.dm_db_index_operational_stats to increase.

For this we are going to look at a couple columns that were included in the last post in this series. Those columns are:

- **leaf_allocation_count (bigint)** - Cumulative count of leaf-level page allocations in the index or heap. For an index, a page allocation corresponds to a page split.
- **nonleaf_allocation_count (bigint)** - Cumulative count of page allocations caused by page splits above the leaf level.

As the definition states, these columns accumulate the the lead and non-leaf allocations that have been allocated to an index. When inserts and updates increase the size of the index the number of pages added will be tallied in these columns And when rows are updated and the updates cause page splits the allocation count will increase. The definition also states that these columns accumulate data that corresponds to page allocations.

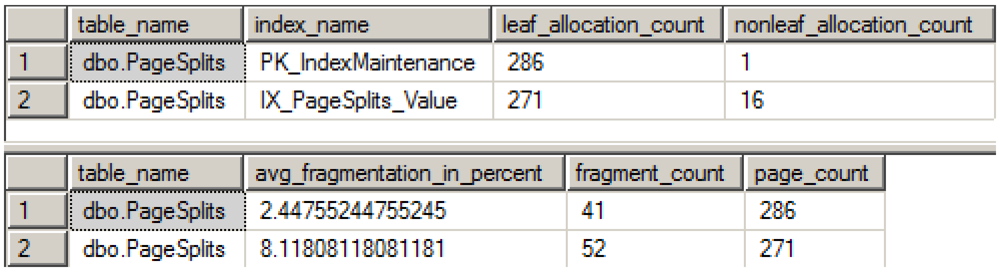
Increment Allocation

Let's investigate how this data is incremented. To start with let's take a look at how some typical activity on a table can increment the columns leaf_allocation_count and non_leaf_allocation_count. To do this execute the following script:

```
USE tempdb ;
GO
IF OBJECT_ID('dbo.PageSplits') IS NOT NULL
    DROP TABLE dbo.PageSplits ;
CREATE TABLE dbo.PageSplits
(
 ID INT
,Value VARCHAR(900)
,CreateDate DATETIME
,CONSTRAINT PK_IndexMaintenance PRIMARY KEY (ID)
) ;
CREATE INDEX IX_PageSplits_Value ON dbo.PageSplits (Value) ;

WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.PageSplits
SELECT TOP (10000)
        n
       ,REPLICATE('X',200)
       ,GETDATE()
FROM    nums
ORDER BY 2 ;

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM
sys.dm_db_index_operational_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                       NULL,NULL,'LIMITED') ips
```

In this script a table was created and the 10,000 rows were added to the table that has clustered and non-clustered indexes. The next query reports the data accumulated in sys.dm_db_index_operational_stats for the leaf_allocation_count and nonleaf_allocation_count columns. With the last query reporting the fragmentation and the number of pages allocated to the table.

The results should look like this:



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	286	1
2	dbo.PageSplits	IX_PageSplits_Value	271	16

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	2.44755244755245	41	286
2	dbo.PageSplits	8.11808118081181	52	271

We can see that 286 leaf pages for the clustered index and 271 leaf pages for the non-clustered index have been allocated to the indexes. From sys.dm_db_index_physical_stats we see that identical numbers of pages are associated with the indexes. Also, there is some fragmentation listed but not an amount that is of importance.

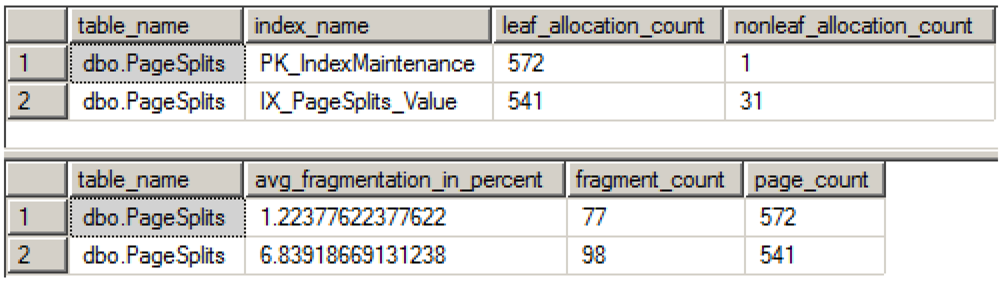
More Rows, More of the Same

The section title is likely a giveaway, but let's show the effect of adding more rows to the table. Execute the next script:

```
WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.PageSplits
SELECT TOP (10000)
        n + 10000
       ,REPLICATE('X',200)
       ,GETDATE()
FROM    nums
ORDER BY 2 ;

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM
sys.dm_db_index_operational_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
```

```
FROM    sys.dm_db_index_physical_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                                   NULL,NULL,'LIMITED') ips
```

This script has a similar output as the first script which will look like the following:



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	572	1
2	dbo.PageSplits	IX_PageSplits_Value	541	31

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	1.22377622377622	77	572
2	dbo.PageSplits	6.83918669131238	98	541

And as you would expect, the number of leaf pages allocated for both indexes increases along with the page counts. Since the new rows are after the first set of rows inserted there isn't any additional fragmentation encountered.

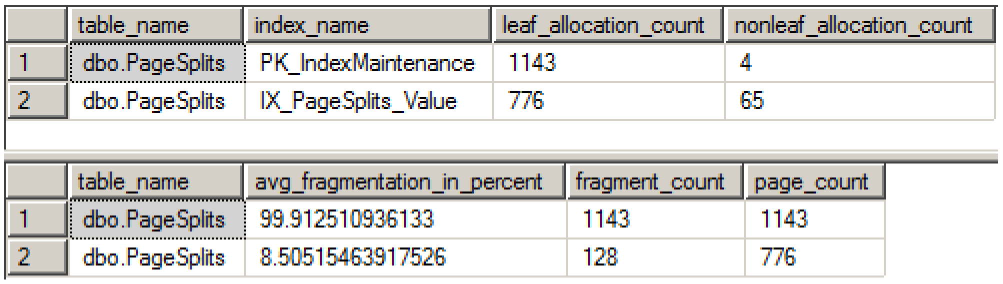
Let's Page Split

Now let's get to the juice. We want to see how the leaf allocation columns can be used to monitor page split activity. To cause some page splits to occur, we'll increase the data in value from 200 to 450 characters.

Run the next script to cause some page splits:

```
UPDATE  dbo.PageSplits
SET     Value = REPLICATE('X',450)
WHERE   ID % 5 = 1

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM
sys.dm_db_index_operational_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                                   NULL,NULL,'LIMITED') ips
```

As you can see in the results below, a number of pages were allocated to both indexes. This allocation coincides with an increase in fragmentation. This fragmentation was a result of the page splits that occurred on the indexes. Thus as pages were needed by the index to accommodate for the page splits, the page allocations were tracked by sys.dm_db_index_operational_stats.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	1143	4
2	dbo.PageSplits	IX_PageSplits_Value	776	65

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	99.912510936133	1143	1143
2	dbo.PageSplits	8.50515463917526	128	776

Fragmentation, Cha, Cha, Cha

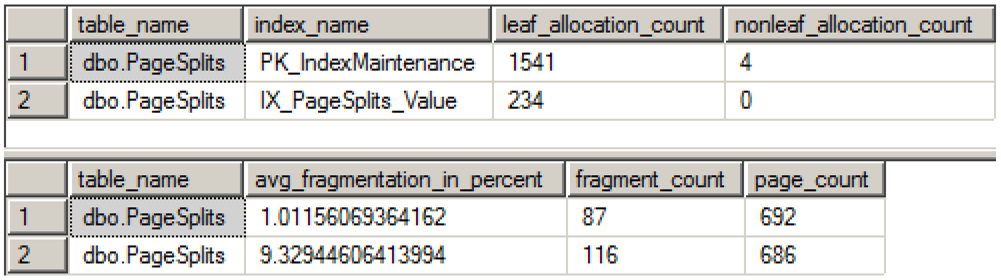
Typically when there is a large amount of fragmentation on an index, we'll be good DBAs and defragment the index to remove the fragmentation. The fragmentation on the clustered index is over 99.9% so there is definitely a need for defragmentation.

Let's do that now with the following script:

```
ALTER INDEX PK_IndexMaintenance ON dbo.PageSplits REORGANIZE ;

ALTER INDEX IX_PageSplits_Value ON dbo.PageSplits REORGANIZE ;

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
        ,i.name AS index_name
        ,leaf_allocation_count
        ,nonleaf_allocation_count
FROM
sys.dm_db_index_operational_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
        ,ips.avg_fragmentation_in_percent
        ,ips.fragment_count
        ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                       NULL,NULL,'LIMITED') ips
```

Viola! The fragmentation is reduced in the index and number of pages for each index reported by the second query is reduced. The same amount of data on fewer pages.

Looking at the first set of results, though, the number of pages allocated to the index has increased again. Not only does the allocation rise when pages are allocated for page splits, but same as when pages are reorganized on the index during defragmentation.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	1541	4
2	dbo.PageSplits	IX_PageSplits_Value	234	0

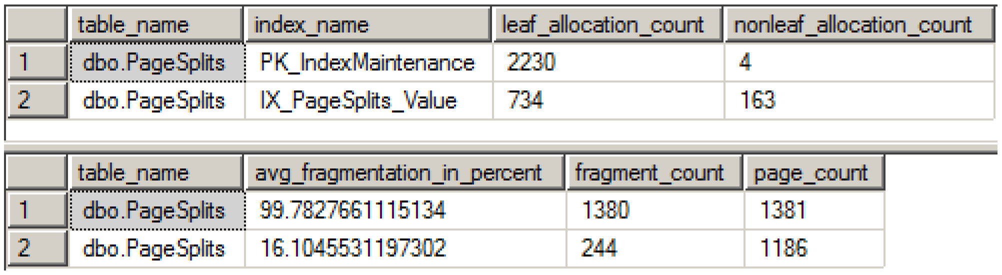
	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	1.01156069364162	87	692
2	dbo.PageSplits	9.32944606413994	116	686

Moar Page Splits and Fragmentation

Let's now cause some more fragmentation for one last item to show about these columns. Run the following script:

```
UPDATE  dbo.PageSplits
SET     Value = REPLICATE('X',900)
WHERE   ID % 5 = 1

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM
sys.dm_db_index_operational_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),OBJECT_ID('dbo.PageSplits'),
                                       NULL,NULL,'LIMITED') ips
```

As the results show below, there were some page splits as evidenced by the fragmentation and a large number of pages were allocated to the indexes. Not really exciting because it was expected.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	2230	4
2	dbo.PageSplits	IX_PageSplits_Value	734	163

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	99.7827661115134	1380	1381
2	dbo.PageSplits	16.1045531197302	244	1186

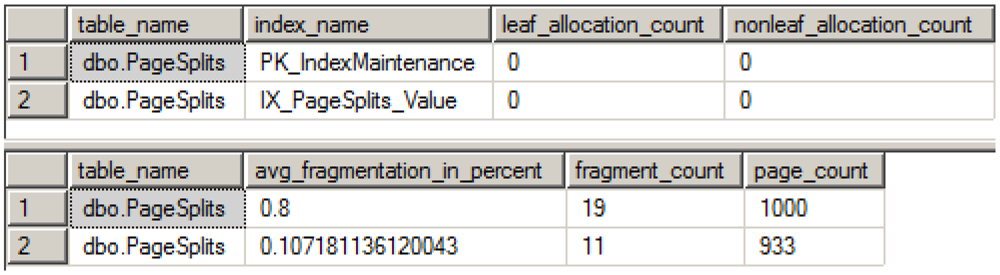
Need to Rebuild

In the previous script to remove fragmentation from the indexes, the REORGANIZE option was used. It isn't always best to reorganize an index but it might be better to rebuild the index at times. Most maintenance scripts examine thresholds to determine when it's best to REORGANIZE an index or when to REBUILD it.

For the final script we'll rebuild the two indexes:

```
ALTER INDEX PK_IndexMaintenance ON dbo.PageSplits REBUILD;

ALTER INDEX IX_PageSplits_Value ON dbo.PageSplits REBUILD;

SELECT
OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id) as
table_name
,i.name as index_name
,leaf_allocation_count
,nonleaf_allocation_count
FROM sys.dm_db_index_operational_stats(DB_ID(),
OBJECT_ID('dbo.PageSplits'),NULL, NULL) ios
INNER JOIN sys.indexes i ON i.object_id = ios.object_id AND i.index_id =
ios.index_id

SELECT
OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id) as
table_name
,ips.avg_fragmentation_in_percent
,ips.fragment_count
,page_count
FROM sys.dm_db_index_physical_stats(DB_ID(),
OBJECT_ID('dbo.PageSplits'),NULL, NULL, 'LIMITED') ips
```

And the results will now show exactly what we probably weren't expecting. Instead of showing the number of pages that were allocated to the index before and after the rebuild, the number of leaf allocations has been reduce to 0.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.PageSplits	PK_IndexMaintenance	0	0
2	dbo.PageSplits	IX_PageSplits_Value	0	0

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.PageSplits	0.8	19	1000
2	dbo.PageSplits	0.107181136120043	11	933

For all intents and purposes the statistics for these columns were wiped out. With the REBUILD operation, the index was recreated and replaced the existing index. That statistics are not carried over from the first to the second physical structures.

Conclusion

Based on the activity we've seen, the leaf and non-leaf allocations in sys.dm_db_index_operational_stats incremented on more than just page split activities. The description from Books On Line regarding page splits is a bit deceptive.

The fact that these columns account for more than just page splits is not a terrible thing. When I consider these columns, I think of them as an accumulation of the how much the index is breathing. This breathing is measured by the page splits (inhales) and the reorganization of pages (exhales). The more it breathes, the more IO that the index requires from day-to-day to support the index. A measurement of this activity is important when considering how to design and maintain a database.

The third point is that the numbers in this DMV have to be taken with a grain of salt. Typical index maintenance will skew the results when comparing one index to another index. Reviews of indexes needs to accommodate for this. Taking snapshots of the index statistics from time to time will help provide meaningful insights into the data this DMV provides.

Last point – if you want a dedicated column to accumulate the page splits on an index then I'd encourage you to <a href="https://connect.microsoft.com/SQL/feedback/ViewFeedback.aspx?FeedbackID=388403">up-vote this connect item.

Index Black Ops Part 6 - Fill Factor vs. Page Splits

In my TSQL2sDay index summary post, that I'd be writing a few posts on the information that is contained in sys.dm_db_index_operational_stats. The posts are the following:

- Index Black Ops Part 1 – Locking and Blocking
- Index Black Ops Part 2 – Page IO Latch, Page Latch
- Index Black Ops Part 3 – Index Usage
- Index Black Ops Part 4 – Index Overhead and Maintenance
- Index Black Ops Part 5 – Page Splits
- Index Black Ops Part 6 – Fill Factor vs. Page Splits

We are now to the last post of this series. In this post we will be investigating the relationship between fill factor and page splits. If you aren't familiar with page splits then take a chance to read the post Index Black Ops Part 5 – Page Splits.

Expanding An Idea

As mentioned in the last post, when data is updated on an index it can lead to page splits. I had originally intended to talk about fill factor with the last post but the length of that post dictated the need for another. So in this post we will use the ideas from that post to see the effect of FILL FACTOR on an index and sys.dm_db_index_operational_stats.

Below I am going to go through three pairs of scripts. Each pair will cover a different fill factor level to see how adjusting it will affect the indexes and page splits that occur. All of the scripts will be using tempdb.

The first script in each pair will perform the following steps:

1. Create dbo.FillFactorTable in tempdb with a clustered and non-clustered index.
2. Insert 10,000 rows into dbo.FillFactorTable.
3. Rebuild the indexes on dbo.FillFactorTable. This will apply the fill factor assigned to the index to the index.
4. Query sys.dm_db_index_operational_stats for leaf and non-leaf allocations.
5. Query sys.dm_db_index_physical_stats for fragmentation and page count.

After that, the second script in each pair will perform the following steps:

1. Update dbo.FillFactorTable to possibly cause page splits. <span style="text-decoration: line-through;">That's called foreshadowing.</span>
2. Query sys.dm_db_index_operational_stats for leaf and non-leaf allocations.
3. Query sys.dm_db_index_physical_stats for fragmentation and page count.

Through these scripts, we should be able to demonstrate the effect of fill factor on the occurrence of page splits.

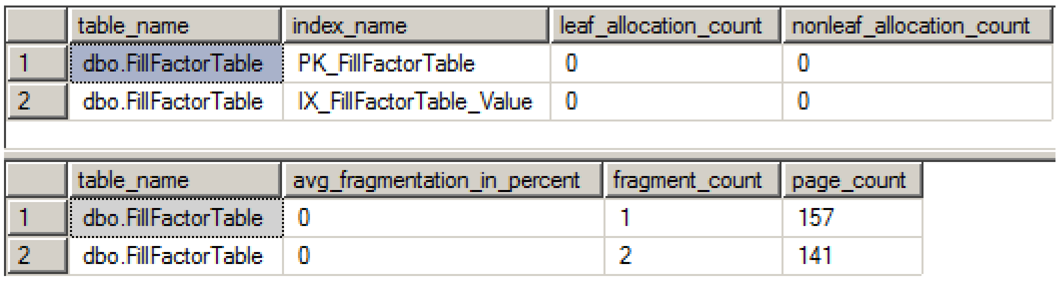
Starting at 0 or 100

Out of the box, SQL Server defaults indexes to a fill factor of 0. This is the same as having a fill factor of 100. To start the demonstrations, run the first script with fill factor 100.

```
USE tempdb ;
GO

IF OBJECT_ID('dbo.FillFactorTable') IS NOT NULL
    DROP TABLE dbo.FillFactorTable ;

CREATE TABLE dbo.FillFactorTable
(
 ID INT
,Value VARCHAR(200)
,CreateDate DATETIME
,CONSTRAINT PK_FillFactorTable PRIMARY KEY (ID) WITH FILLFACTOR = 100
) ;

CREATE INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable (Value)
WITH FILLFACTOR = 100 ;

WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.FillFactorTable
SELECT TOP (10000)
        n
       ,REPLICATE('X',100)
       ,GETDATE()
FROM    nums
ORDER BY 2 ;

ALTER INDEX PK_FillFactorTable ON dbo.FillFactorTable REBUILD ;

ALTER INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable REBUILD ;

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.FillFactorTable'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
```

```
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                                       OBJECT_ID('dbo.FillFactorTable'),NULL,
                                       NULL,'LIMITED') ips
```

The results of the query are below. This shows that there haven't been any page allocations since the index rebuild. And the rebuild pretty much removed all fragmentation from the index. Take note of the page count for the indexes, as we'll see this increase in the later scripts.



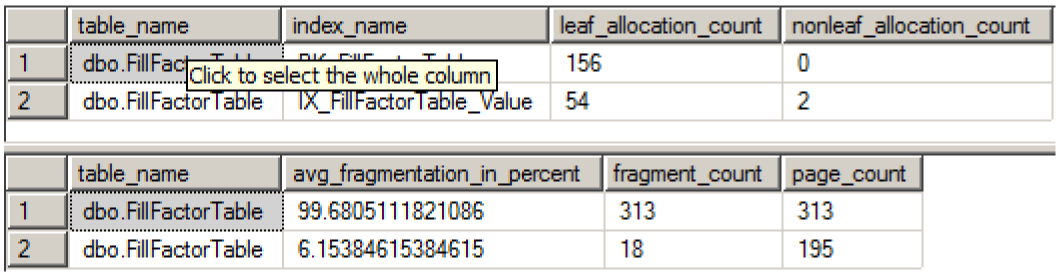
	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFactorTable	PK_FillFactorTable	0	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	0	0

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	0	1	157
2	dbo.FillFactorTable	0	2	141

With that baseline, run the second script listed below:

```
UPDATE  dbo.FillFactorTable
SET     Value = REPLICATE('X',200)
WHERE   ID % 5 = 1

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.FillFactorTable'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                                       OBJECT_ID('dbo.FillFactorTable'),NULL,
                                       NULL,'LIMITED') ips
```

Here are the results from the second script.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFact	Click to select the whole column	156	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	54	2

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	99.6805111821086	313	313
2	dbo.FillFactorTable	6.15384615384615	18	195

As is shown, there is significant fragmentation on the first index. This fragmentation was caused by page splits. Also, both indexes have a number of pages that have been allocated to the index. Based on the previous post, these allocations are related to the page splits that occurred.

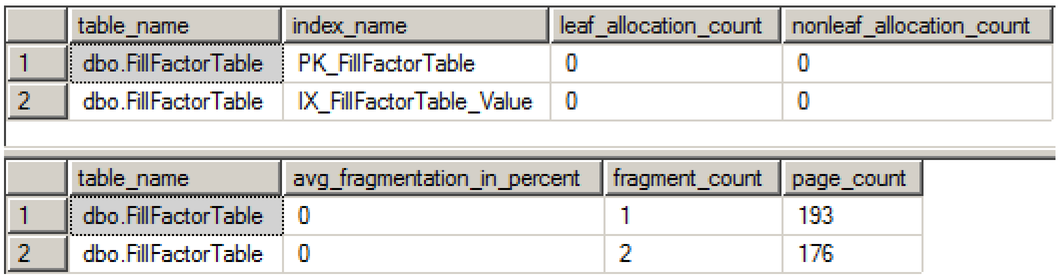
Drop Down to 80

One of the strategies that can be employed to prevent page splits is to reduce the fill factor on an index. Fill factor can be adjust on both clustered and non-clustered indexes. Let's do that now with the table.

```sql
USE tempdb ;
GO

IF OBJECT_ID('dbo.FillFactorTable') IS NOT NULL
    DROP TABLE dbo.FillFactorTable ;

CREATE TABLE dbo.FillFactorTable
(
 ID INT
,Value VARCHAR(200)
,CreateDate DATETIME
,CONSTRAINT PK_FillFactorTable PRIMARY KEY (ID) WITH FILLFACTOR = 80
) ;

CREATE INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable (Value)
WITH FILLFACTOR = 80 ;

WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.FillFactorTable
SELECT TOP (10000)
        n
       ,REPLICATE('X',100)
       ,GETDATE()
FROM    nums
ORDER BY 2 ;

ALTER INDEX PK_FillFactorTable ON dbo.FillFactorTable REBUILD ;
ALTER INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable REBUILD ;
```

```
SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.FillFactorTable'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;
SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                                       OBJECT_ID('dbo.FillFactorTable'),NULL,
                                       NULL,'LIMITED') ips
```

Once the script above executes, the results below will be displayed.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFactorTable	PK_FillFactorTable	0	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	0	0

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	0	1	193
2	dbo.FillFactorTable	0	2	176

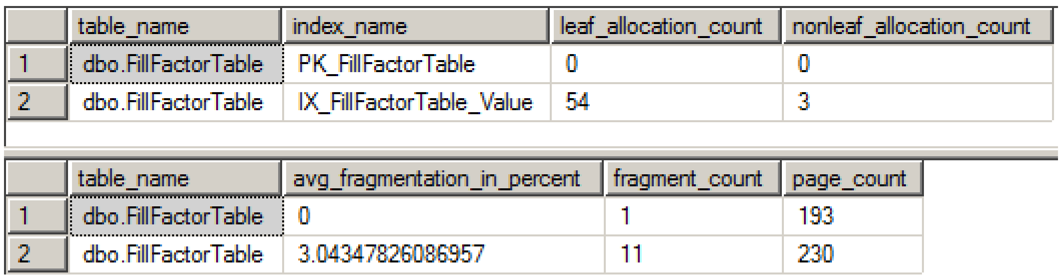
These are similar to the results for fill factor 100. The chief difference in the results is the number of pages allocated to each index. The clustered index increased from 157 to 193 pages and the non-clustered index increased from 141 to 195 pages. The number of pages increase because only 80% of the page was filled with data.

We've got extra space in the index now. Is it enough for the updates? Let's run the second script in this pair to see what happens.

```
UPDATE  dbo.FillFactorTable
SET     Value = REPLICATE('X',200)
WHERE   ID % 5 = 1

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.FillFactorTable'),
                                          NULL,NULL) ios
```

```
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                                       OBJECT_ID('dbo.FillFactorTable'),NULL,
                                       NULL,'LIMITED') ips
```

The results for this script are starkly different from the results with a fill factor of 100. In this case, the lower fill factor resulted in no allocations for the clustered index and significantly less for the non-clustered index. And as mentioned these allocations would have been due to page splits.



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFactorTable	PK_FillFactorTable	0	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	54	3

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	0	1	193
2	dbo.FillFactorTable	3.04347826086957	11	230

The other item to note, the page count for the clustered index did not increase. But not only did it not increase, it is also much lower that the post-UPDATE page count for fill factor 100. This important because one of the things that can be an issue with fill factor is the additional amount of space required to store the index.

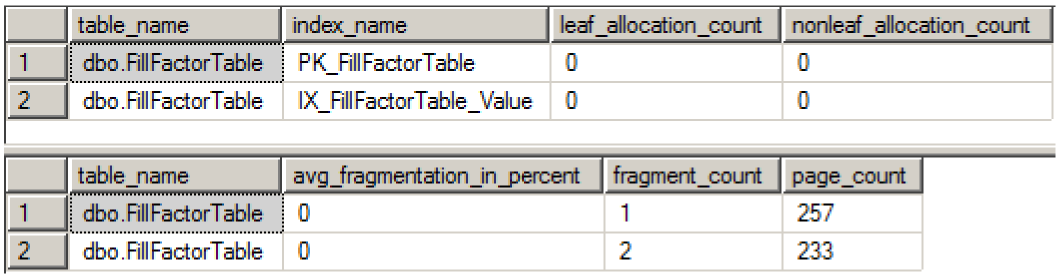
Drop Down to 60

In this last pair of examples, we'll set the fill factor to 60. As you may guess, this fill factor will also prevent page splits. But it will also use an excessive number of pages for each of the indexes.

```
USE tempdb ;
GO

IF OBJECT_ID('dbo.FillFactorTable') IS NOT NULL
    DROP TABLE dbo.FillFactorTable ;

CREATE TABLE dbo.FillFactorTable
(
 ID INT
,Value VARCHAR(200)
,CreateDate DATETIME
,CONSTRAINT PK_FillFactorTable PRIMARY KEY (ID) WITH FILLFACTOR = 60
) ;

CREATE INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable (Value)
WITH FILLFACTOR = 60 ;
```

```
WITH
l0 AS (SELECT 0 AS C UNION ALL SELECT 0),
l1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B),
l2 AS (SELECT 0 AS C FROM l1 AS A CROSS JOIN l1 AS B),
l3 AS (SELECT 0 AS C FROM l2 AS A CROSS JOIN l2 AS B),
l4 AS (SELECT 0 AS C FROM l3 AS A CROSS JOIN l3 AS B),
l5 AS (SELECT 0 AS C FROM l4 AS A CROSS JOIN l4 AS B),
nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS N FROM l5)
INSERT INTO dbo.FillFactorTable
SELECT TOP (10000)
        n
       ,REPLICATE('X',100)
       ,GETDATE()
FROM    nums
ORDER BY 2 ;

ALTER INDEX PK_FillFactorTable ON dbo.FillFactorTable REBUILD ;
ALTER INDEX IX_FillFactorTable_Value ON dbo.FillFactorTable REBUILD ;

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                          OBJECT_ID('dbo.FillFactorTable'),
                                          NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;
SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                                       OBJECT_ID('dbo.FillFactorTable'),NULL,
                                       NULL,'LIMITED') ips
```

After running the script above, the results below will display. These are similar to the previous executions of the these scripts. The exception this that the page counts for both indexes are substantially higher than before. In fact, the non-clustered index with a fill factor of 60 is large than it was post-UPDATE when the fill factor was 100 (233 vs. 195 pages).



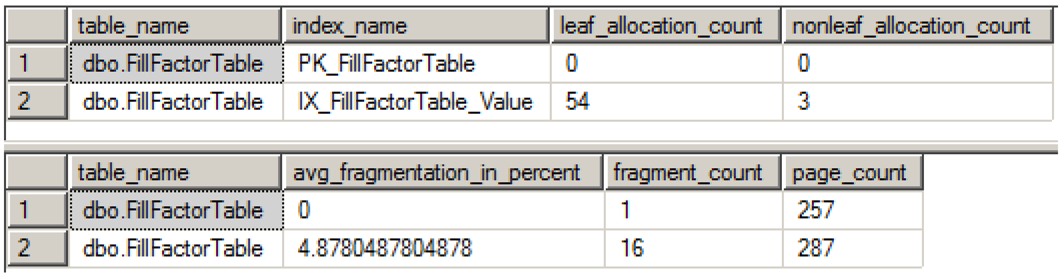
	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFactorTable	PK_FillFactorTable	0	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	0	0

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	0	1	257
2	dbo.FillFactorTable	0	2	233

We've seen the results before the anticipated page splits from the update. Now let's run the update to see what happens.

```
UPDATE  dbo.FillFactorTable
SET     Value = REPLICATE('X',200)
WHERE   ID % 5 = 1

SELECT  OBJECT_SCHEMA_NAME(ios.object_id) + '.' + OBJECT_NAME(ios.object_id)
AS table_name
       ,i.name AS index_name
       ,leaf_allocation_count
       ,nonleaf_allocation_count
FROM    sys.dm_db_index_operational_stats(DB_ID(),
                                  OBJECT_ID('dbo.FillFactorTable'),
                                  NULL,NULL) ios
        INNER JOIN sys.indexes i
            ON i.object_id = ios.object_id
               AND i.index_id = ios.index_id ;

SELECT  OBJECT_SCHEMA_NAME(ips.object_id) + '.' + OBJECT_NAME(ips.object_id)
AS table_name
       ,ips.avg_fragmentation_in_percent
       ,ips.fragment_count
       ,page_count
FROM    sys.dm_db_index_physical_stats(DB_ID(),
                               OBJECT_ID('dbo.FillFactorTable'),NULL,
                               NULL,'LIMITED') ips
```

With the script executed, the results should be:



	table_name	index_name	leaf_allocation_count	nonleaf_allocation_count
1	dbo.FillFactorTable	PK_FillFactorTable	0	0
2	dbo.FillFactorTable	IX_FillFactorTable_Value	54	3

	table_name	avg_fragmentation_in_percent	fragment_count	page_count
1	dbo.FillFactorTable	0	1	257
2	dbo.FillFactorTable	4.8780487804878	16	287

In these script having fill factor at 60 continued to help prevent page splits. On the clustered index there were no page splits or leaf allocations. The non-clustered index had the same number of allocations as when the fill factor was 80.

With the fill factor this low, though, the value of using fill factor has been diminished. It is diminished because for the same data, the index is using up to 25% more tables. In this case the amount of space is not a huge deal. But consider an index that is over a GB in size or larger. This extra space is just sitting in the database and it's not providing any value.

It is important to select a fill factor that isn't too high, such as 100. And it is also important to select a fill factor that isn't too low – as a fill factor of 60 is in this case.

Moar Information

Fill factor and page splits have a nicely tied relationship. One will affect the other and information in sys.dm_db_operational stats can be used to watch this activity. This post didn't exactly expand on the information I've already provided on the leaf and non-lead allocation columns. It did, hopefully, walk-through a case where this information could be used.

One thing we haven't looked at is the effect on performance due to the page splits and increased numbers of pages for the indexes. Both of these will have an effect on performance but much more time and space will be needed to go through that exercise.

For additional information on Fill Factor, I'd recommend reading Paul Randal's (blog | twitter) latest article on SQL Mag. The article covers more on what I covered here.

Hopefully these posts on sys.dm_db_operational stats have proven to be useful and informative. Please leave comments if there are other aspects of this DMV you'd like to see covered.