

SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 7: Index Fragmentation

Paul S. Randal
Paul@SQLskills.com



Overview

- Data access methods
- What is fragmentation?
- How does fragmentation happen?
- Detecting, mitigating, removing fragmentation

- 2010 PASS presentation on GUIDs
 - **Tales from the Trenches: GUIDs - Use, Abuse and How to Move Forward**
<mms://passmedia.sqlpass.org/SummitOnDemand/2010/Top10/AD372S.wmv>

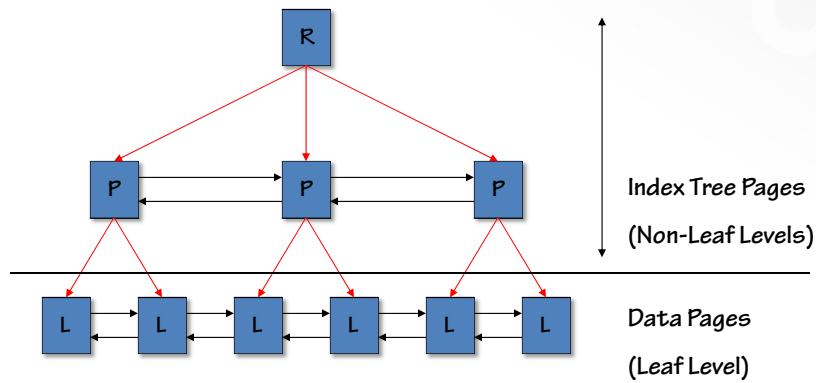


2

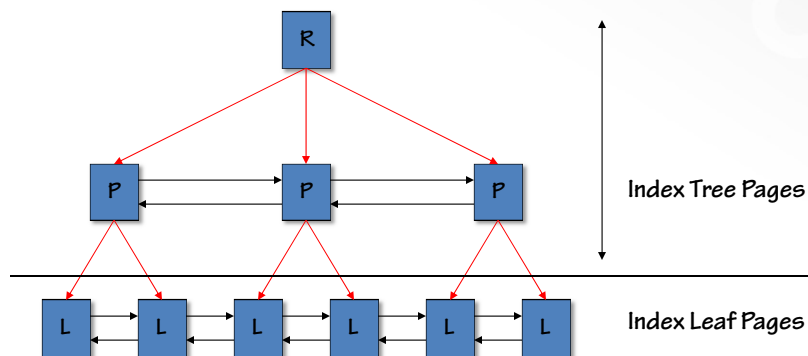
© SQLskills. All rights reserved.
<http://www.SQLskills.com>



Clustered Index Structure



Nonclustered Index Structure



Singleton Lookup (Index Seek/Clustered Index Seek)

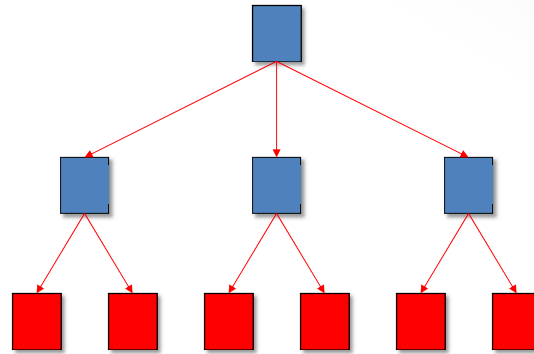
The diagram illustrates a B-tree structure. At the top is the root node, a blue square with a red horizontal bar. A red arrow points down to the root. The root has three children, all blue squares. The left child has a red horizontal bar and two children of its own. The middle child has two children. The right child has two children. The leftmost leaf node (under the left child) has a red horizontal bar. A red arrow points from the root to this leaf. A black arrow points from the text 'Matching record' to the red bar in this leaf node.

Matching record

Range Scan (Index Scan/Clustered Index Scan)

Matching records (in **red**)

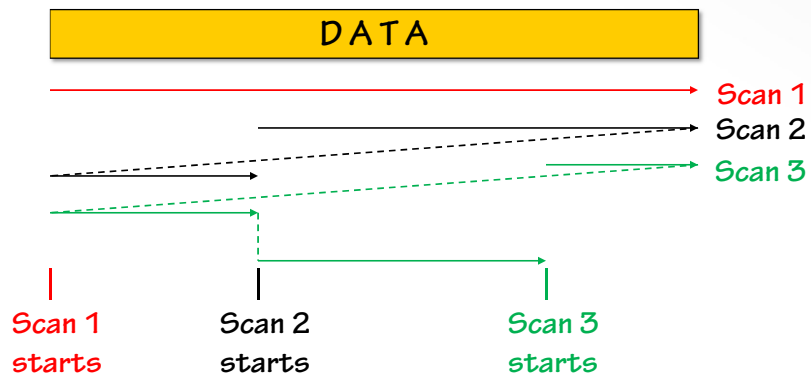
Allocation Order Scan (Table scan or unordered Clustered/Index scan)



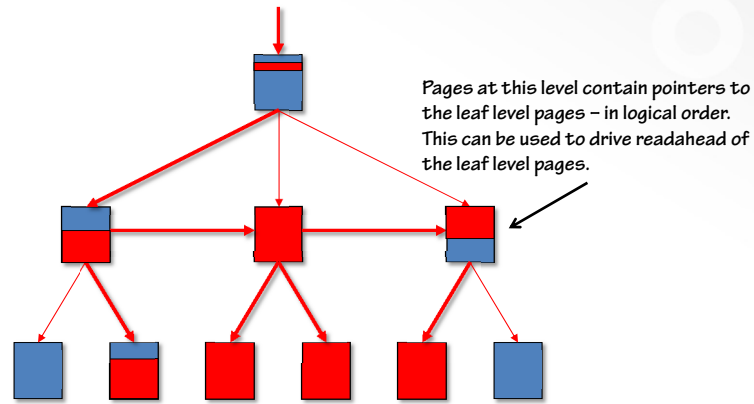
Matching records (in **red**)

Side Note: Merry-Go-Round Scans

- One more reason to not get index order with 'SELECT *'?
- Enterprise only, no control or monitoring possible



Range-Scan: Readahead (1)



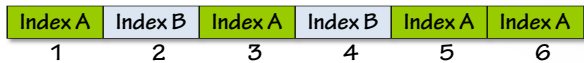
Range-Scan: Readahead (2)

- **Why use readahead?**
 - Keep the CPUs busy, maximize throughput, avoid I/O waits
 - More efficient to issue 1 x 8-page I/O than 8 x 1-page I/Os
- **Feedback mechanism to determine how far ahead of the scan point to read**
- **Driven from parent level during range scans**
 - Parent level pages contain logically-ordered links to the leaf level
- **Issues 1, 8, 32, 64-page I/Os, and up to 8 parallel I/Os (i.e. 512 pages)**
 - 8, 32+ page I/Os only possible with contiguous pages
 - Better contiguity = bigger I/Os = better performance
- **Problem: fragmentation causes lower-performing range scans**

Logical Fragmentation Defined

- (Sometimes called “external” fragmentation)
- Occurs when the next logical page is not the next physical page
- Prevents optimal readahead
 - Reduces range scan performance
- Does not affect pages that are already in cache
 - Smaller indexes cause less of a performance hit (e.g. 1-5000 pages or less)
- Reported as `avg_fragmentation_in_percent` for indexes in the `sys.dm_db_index_physical_stats` DMV

Extent Fragmentation Defined

- Occurs when the extents in an index are not contiguous
- 
- Also affects readahead performance but not as much
 - When writing the DMV for 2005, we decided to remove it to avoid confusion from too many measures of ‘fragmentation’
 - Reported as `avg_fragmentation_in_percent` in the `sys.dm_db_index_physical_stats` DMV for heaps ONLY
 - (2000: extent fragmentation algorithm in DBCC SHOWCONTIG is documented as not working for multiple files)

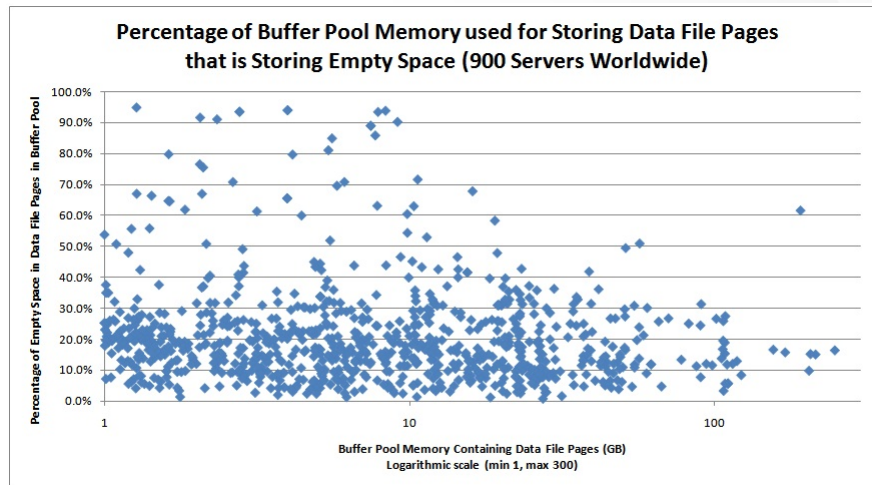
Using -E Startup Option

- <http://support.microsoft.com/kb/329526>
- SQL CAT advises using it, as does Fast Track DW
- During index build/rebuild (and all other operations):
 - 2005: 4 extents allocated before round-robin (256KB)
 - 2008+: 64 extents allocated before round-robin (4MB)
 - NOT single allocations of 4 or 64 extents
- Useful for very large indexes and range scans but NOT useful for OLTP
 - Unless you have a large DW, ignore this option
- Combine with large RAID stripe size
- Consider low MAXDOP during index (re)build for maximum contiguity and best range scan performance on Enterprise Edition

Page Density Defined

- (Sometimes called “physical” or “internal” fragmentation)
- Page fullness is below the optimal level so lots of wasted space
- Effect is:
 - Increased disk space (more pages required to hold the same number of rows)
 - Increased I/Os to read the same amount of data, leading to I/O subsystem pressure and overall performance degradation
 - Greater memory usage if most of the index is memory resident, leading to increased I/Os, and so on...
- This means ‘fragmentation’ can affect your performance even if you don’t do index range scans
- Reported as avg_page_space_used_in_percent in the DMV

Low Buffer Pool Usage

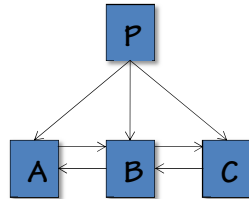


- Source: my blog at <http://bit.ly/10qs55H>

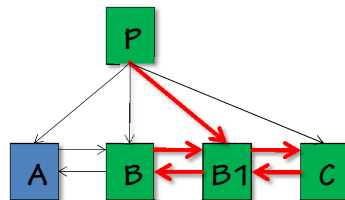
What is a Page Split?

- This is the primary cause of fragmentation, and is itself a performance problem when it occurs
- Occurs when a record must be inserted onto (or expanded on) a specific page in the index and there is not enough space
 - Could be caused by a new record or an updated record that is now longer than it was before
 - Could also be caused by enabling snapshot isolation, which makes updated records 14-bytes longer
- The page has to 'split' to make room
 - Split point is usually as close to 50/50 as possible, but may be skewed if Storage Engine can determine an obvious split point)

Updates During a Page Split



- Page B splits into B and B1
- New page B1 is unlikely to be physically adjacent to B
- Pages P, B, C must be updated to change/create page links
 - Page P might split...
- All changes are fully logged



Page Split Mechanism

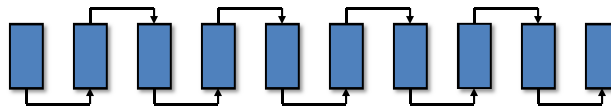
- A new page is allocated to the index
 - Usually not physically contiguous to the splitting page so immediately it is logically fragmented
- All records before the split point stay on the original page
- All records after the split point are moved to the new page
- New record is written to either page, determined by key value
- Both pages (old, and newly allocated) now have low page density
- New record must be inserted into index level above the leaf
 - Could also cause a page split, cascading upwards to the root page
- All of this is fully logged regardless of recovery model
- Summary: page splits are very expensive and are the main cause of fragmentation

Tracking Page Splits

- There are 'good' and 'nasty' page splits...
 - 'Good' split is when a page is allocated as part of an append-only insert pattern
 - 'Nasty' split is when a real page split occurs
- Unfortunately, all documented methods of tracking page splits prior to SQL Server 2012 do not allow differentiation between 'good' and 'nasty' page splits
 - Perfmon counter
 - sys.dm_db_index_operational_stats
 - Extended event (possibly with post-processing)
- Either use log/log backup scanning or 2012+ Extended Events
 - See my blog post at <http://bit.ly/UG1SyG>

Page Splits in Action (1)

Index leaf level of newly built index

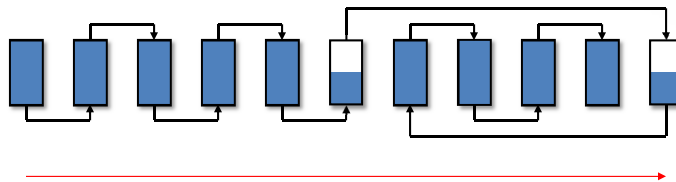


Long arrow is the allocation order

Small arrows are following the logical order

Page Splits in Action (2)

Newly built index leaf after a single page split

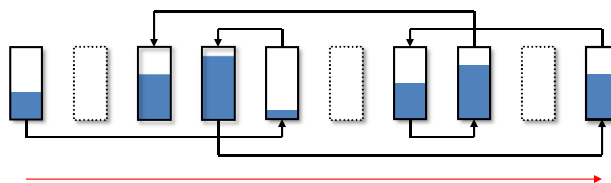


Long arrow is the allocation order

Small arrows are following the logical order

Page Splits in Action (3)

Index leaf level after random inserts/deletes



Long arrow is the allocation order

Small arrows are following the logical order

Demo

Increased logging during page splits

What Causes Fragmentation?

- **Schemas/workloads that cause page splits on full pages**
 - GUID as high-order key (or any other random key)
 - Can even affect nonclustered indexes
 - Updates to variable-length columns
 - Badly configured FILLFACTOR (more in a few slides)
- **Clustered index is likely the only one you can make the key not cause fragmentation by picking an ascending order key (e.g. bigint identity)**
- **Wide schemas that only fit a few records per page**
 - E.g. a fixed-size 5000 byte row = 3000 bytes lost per page!
- **Real-world example:**
 - Social networking site that has a homepage comments table with the member ID as the high-order key
 - Patient check-in company using GUID as clustering key

Can DML Cause Fragmentation?

- **Yes, data modifications can lead to fragmentation**
- **INSERT**
 - YES – if key value is not ever increasing/decreasing (e.g. GUID)
 - NO – if key is ever increasing/decreasing (e.g. INT IDENTITY)
- **UPDATE**
 - YES – if updates make variable-length columns wider on full pages
 - NO – if columns are fixed width or columns have 'place holder' values (i.e. DEFAULT values) to minimize row expansion on update
- **DELETE**
 - YES – if deletes are singleton deletes (Swiss-cheese problem – page density issues)
 - NO – if deletes are range deletes for archival purposes

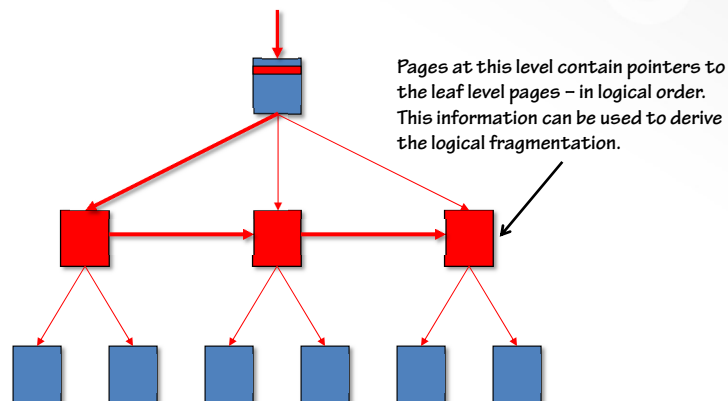
Symptoms of Fragmentation

- **Poor/degrading query performance over time**
 - Longer run-times
 - More disk activity
 - SET STATISTICS IO ON
 - More frequent checkpoints occurring
 - Increased logging (from page split activity)
 - Depending on the average record length and the split point, a page split could log up to 50 times more than a regular insert!
 - Increased buffer pool usage
- **Worsening results from the sys.dm_db_index_physical_stats DMV**
 - Keys to success are knowing which indexes to look at and how to interpret the results

sys.dm_db_index_physical_stats (1)

- **Replacement for DBCC SHOWCONTIG since SQL Server 2005**
 - `select * from sys.dm_db_index_physical_stats (dbid, objectid, indexid, partitionid, samplemode)`
- **No longer need insert/exec to analyze/process DBCC SHOWCONTIG results**
 - DMVs are programmatically “composable”
 - However, this is a DMF, not a true DMV so must do work for results
- **Ability to control how much data is read using sample mode (LIMITED, SAMPLED, DETAILED)**
 - LIMITED (default) does not read the leaf level so is fastest mode
 - SAMPLED reads 1% of the leaf-level pages if the index/partition has more than 10000 pages
 - DETAILED reads everything and is the slowest mode
 - Beware of running against an entire database, could ‘flush’ the buffer pool

sys.dm_db_index_physical_stats (2) LIMITED Scanning Mode



sys.dm_db_index_physical_stats (3)

Output and How to Interpret it

- **Logical fragmentation**
 - avg_fragmentation_in_percent (should be low)
- **Page density**
 - avg_page_space_used_in_percent
 - Should be high for data warehouse
 - Should have some free space for OLTP
- **Other counters exist (e.g. fragments, avg. fragment size) but these were only invented to be more accessible to users – somewhat unsuccessfully**

Demo

Detecting fragmentation using sys.dm_db_index_physical_stats

How to Avoid Fragmentation?

- **Avoid 'random' index keys**
 - Almost impossible to do for nonclustered indexes
 - For clustered indexes, be careful about moving to (BIG)INT IDENTITY as small row size combined with many concurrent inserters could lead to an 'insert hotspot' performance issue
- **Implement index FILLFACTORs and periodically remove fragmentation**
 - See next slides...

FILLFACTOR

- **Makes the Storage Engine leave space on each leaf-level page to allow row inserts/expansions without causing page splits**
- **Specified at index creation or rebuild time**
 - NOT maintained during regular DML
- **Can specify with sp_configure for entire instance**
 - Not recommended – specify it per index
- **Use PAD_INDEX to affect the upper levels of the b-tree (uses the FILLFACTOR value)**
 - Rarely used
- **0 = 100 = default value with special meaning of 'leave no space'**
 - Excellent for data warehouse, but not ideal for OLTP
- **For OLTP, which value to use?**

Configuring FILLFACTOR

- **Balancing act between how often page splits occur and how often you can rebuild/defrag the index**
- **What is going to cause page splits in your schema?**
 - UPDATES to variable-width data types?
 - Random INSERTs?
 - The more volatile ⇒ lower FILLFACTOR
- **How often can you rebuild/defrag?**
 - The more frequent ⇒ higher FILLFACTOR
- **Pick a value, try it, monitor fragmentation, tweak it**
 - Use DMVs to see how fast the fragmentation increases
 - The faster fragmentation occurs ⇒ lower FILLFACTOR or decreased time between rebuilds/defrag
 - 70% is a common first guess

How to Remove Fragmentation?

- **2 realistic choices**
 - Rebuild the index: ALTER INDEX ... REBUILD
 - Create a brand new index structure
 - Reorganize the index: ALTER INDEX ... REORGANIZE
 - Shuffle the existing pages allocated to the index
- **Also CREATE INDEX ... WITH (DROP_EXISTING = ON)**
 - Commonly used to move or (re)partition an index
- **Can also choose not to remove fragmentation**
 - If the index isn't used for range scans, and page density isn't an issue, why spend the resources?
- **Don't just rebuild everything every day**
 - Although for an involuntary DBA with enough resources, this can be better than nothing
- **Synchronous mirroring or AGs may force REORGANIZE to be used**

ALTER INDEX ... REBUILD

- Replaced DBCC DBREINDEX in SQL Server 2005 onward
- **Pros**
 - Atomic operation
 - No knowledge of index schema or constraints required
 - Can use multiple CPUs, and control MAXDOP
 - Rebuilds index statistics (with equivalent of full scan, or sampled if partitioned index)
 - Can rebuild a single partition or all partitions
 - Can be performed online
 - 2012: Indexes with LOB columns (plus clustered index on table with LOB column)
 - 2014: Single partition
 - Can be minimally-logged (but log backup will be the same size)
- **Cons**
 - Atomic operation – potentially long rollback on interrupt, all or nothing semantics
 - Requires creating complete new index before dropping old one
 - When offline – SCH-M table lock for nonclustered or clustered index rebuild
 - 2014: blocking lock can be postponed to allow better concurrency

ALTER INDEX ... REORGANIZE

- Replaced DBCC INDEXDEFRAG in SQL Server 2005 onward
- **Pros**
 - ALWAYS online – only requires table IX lock
 - Interruptible with no loss of work – stops instantly
 - Has progress reporting in sys.dm_exec_requests / percent_complete
 - Optionally compacts LOB storage (on by default)
 - Usually faster for a lightly fragmented index
 - Can reorganize one or all partitions
 - Does not require any extra disk space
- **Cons**
 - Usually slower for a heavily fragmented index
 - Always fully-logged, single CPU only, does not update statistics
 - Does not do as good a job as removing fragmentation when there is plenty of contiguous free space

CREATE INDEX ... WITH (DROP_EXISTING=ON)

- **Don't use this if you just want to rebuild the index with no changes**
- **Pros**
 - Same as ALTER INDEX ... REBUILD
 - Can move the index to a new location
 - Can rebuild the index with a new partitioning scheme
 - Can change the index schema (keys, sort order, etc)
 - Can do all of this online (with same limitations as regular index rebuild)
- **Cons**
 - Same as ALTER INDEX ... REBUILD
 - Need to know the index schema

Comparison Points: REBUILD vs. REORGANIZE

- **Space required**
 - This may force you to do REORGANIZE
- **Log generated**
 - This may force you to do 'staggered index maintenance' using REORGANIZE
- **Algorithm speed on amount of fragmentation**
- **Locks required (i.e. online or not)**
 - This may force you to do REORGANIZE
- **Interruptible or not**
- **Progress reporting or not**
- **Consider using SORT_IN_TEMPDB for REBUILD, especially in 2014**
 - Reduces churn from intermediate sort, and speed boost on 2014

When To Rebuild vs. Reorganize

- Much debate on this, basically it depends!
- I had to come up with numbers for Books Online so I chose:
 - < 10% do nothing
 - 10% <> 30% defrag/reorganize
 - 30%+ rebuild
 - And don't do anything if the index has < 1-5000 pages
- Your mileage may (and will) vary

Paul's Method...

- Create a table with one row per index you want to work on
 - I call it the 'driver table'
- Call the DMV for the indexes listed in the driver table
- Use per-index fragmentation thresholds to determine whether to rebuild, reorganize, or do nothing
- Log what you decide to do for future reference
- Optional: keep a counter of how many times in succession an index is rebuilt and programmatically reduce fillfactor
- Much easier: use code someone's already written...
 - Ola Hallengren's maintenance solution, free and wonderful
 - <http://ola.hallengren.com>

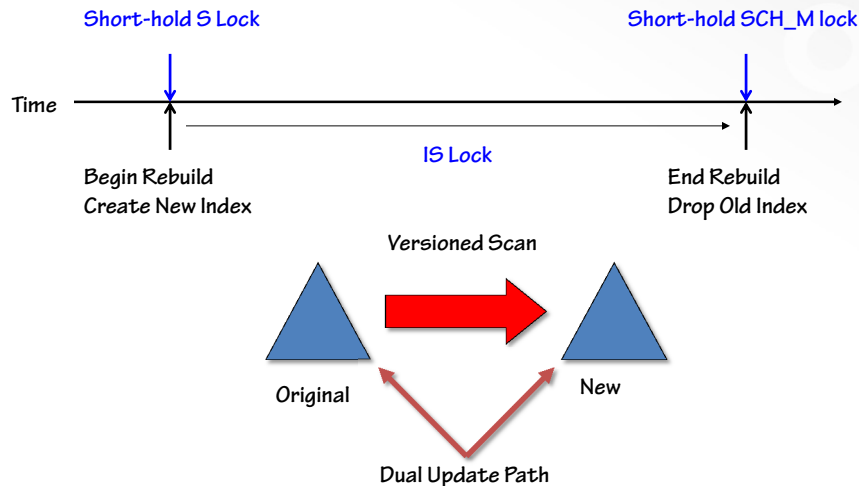
Setting FILLFACTOR During REBUILD/REORGANIZE

- Can only be set when creating or rebuilding an index
 - Stores the fillfactor in the index metadata
- Cannot be set directly with REORGANIZE
- REBUILD and REORGANIZE use the metadata-stored fillfactor, if there is one, otherwise they will use the instance-wide fillfactor
 - Unless a fillfactor is specified on the REBUILD
 - I.e. REBUILD specified fillfactor overrides metadata-stored fillfactor, which overrides instance-wide fillfactor
- Fillfactor is only set in index metadata by a CREATE or REBUILD

Online Index Operations

- **BEWARE: now a fully logged operation in 2008 onwards!!**
 - See <http://support.microsoft.com/kb/2407439>
- 'Online' is a bit of a misnomer
 - Concurrent activity is impacted for (hopefully) very brief period at start and completion of rebuild
 - 2014: blocking lock can be postponed to allow better concurrency
- Performance may be impacted during operation as DML is directed to two indexes and versioning is used
- Indexes with LOB columns cannot have online operations until SQL Server 2012+
 - E.g. a clustered index on a table with a LOB column
- Only one online index operation per table at a time
 - Can online create, drop, rebuild, (re)partition an index

Inside Online Index Operations



Online Progress Report

EventClass	TextData	EventSubClass	ObjectID	ObjectName	BigIntData1	BigIntData2	Applicatio
ExistingConnection	-- network protocol: LPC set quoted_id...						SQL Serv
SQL:BatchStarting	alter index test on member rebuild with [...]						SQL Serv
Progress Report: Online Inde...		1 - Start	213575799	test			SQL Serv
Progress Report: Online Inde...		2 - Stage 1 execution begin					SQL Serv
Progress Report: Online Inde...		6 - Inserted row count	213575799		7221	0	SQL Serv
Progress Report: Online Inde...		6 - Inserted row count	213575799		10000	0	SQL Serv
Progress Report: Online Inde...		3 - Stage 1 execution end					SQL Serv
Progress Report: Online Inde...		7 - Done	213575799	test			SQL Serv
SQL:BatchCompleted	alter index test on member rebuild with [...]						SQL Serv
Trace Pause							
Trace Start							

alter index test on member rebuild with (online = on)
go

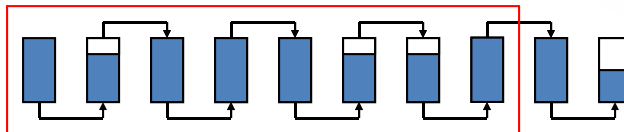
Trace is running. Ln 11, Col 3 Rows: 29 Connections: 1

Inside REORGANIZE (1)

- **Stage 1: leaf-level compaction**
 - Make leaf-level pages have free space near to original FILLFACTOR if they have too much
 - Compacts pages by squishing rows together in groups of eight pages
 - Deletes ghost rows and deallocates empty pages
- **Stage 2: leaf-level defragment**
 - Makes logical order of leaf-level same as physical order
 - Uses logical order scan and allocation order scan in lockstep
 - Drops and reestablishes scan position after every page so totally online apart from page X locks
 - Reorders pages currently allocated to the index

Inside REORGANIZE (2)

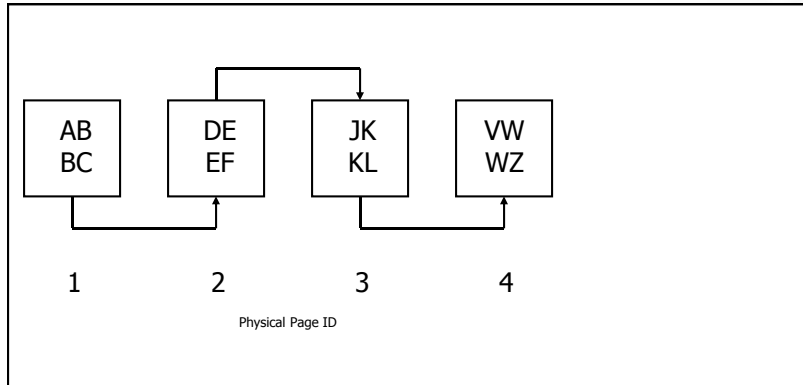
- Uses a 'sliding window' compaction algorithm
- Found certain applications created pathological cases more frequently than expected (e.g. SAP) when only consider two pages at a time



- This algorithm will only perform compaction if there is enough space in an 8-page window to remove one page

Inside REORGANIZE (3)

- Page reordering example



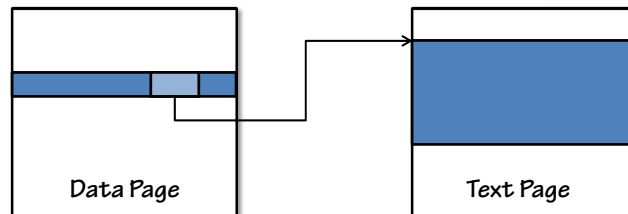
Demo

Removing fragmentation

LOB/Row-Overflow Data (revisited)

(E.g. `varchar (max)`, `text`, `XML`)

- Data/index record stores a pointer to the actual LOB value on a different page



- You remove fragmentation but scan performance is still poor...
- Accessing the LOB data or row-overflow data requires (potentially) another I/O – usually random!
 - Can be a cause of poor performance even if no index fragmentation

Additional: Are Your Indexes Being Used?

- There are lots of bad practices around index strategy, including creating extra indexes
 - E.g. an index for each column in the table
- Extra, unused indexes waste resources as they must be maintained by DML operations
- Use the `sys.dm_db_index_usage_stats` DMV to tell if an index is being used at all during the business cycle
 - Beware of indexes not being used but enforcing unique constraints
 - Beware that in 2012+ the stats are zero'd for indexes rebuilt online
 - And this will not change – bug report closed as 'by design'

Summary: Prevent Fragmentation

- As you can see, fragmentation is very expensive when it happens
- Many people say not to bother about fragmentation
 - They're WRONG!
 - Lots of wasted storage space and extra I/Os
 - Lots of wasted buffer pool memory
 - Lots of extra log to back up, ship, mirror, scan...
 - Performance hit of the page splits happening
- Still a problem even when using SSDs
- Set appropriate FILLFACTORS for indexes that get heavily fragmented
 - Reduce all three extra uses of space
 - Start with FILLFACTOR = 70 and tweak as needed
- Consider changing index keys (carefully)

Whitepapers on This and More...

- Microsoft SQL Server 2000 Index Defragmentation Best Practices
 - <http://technet.microsoft.com/library/Cc966523>
 - Based on SQL Server 2000, so discusses DBREINDEX vs. INDEXDEFRAG
 - Concepts translate to 2005+
- Online Indexing Operations in SQL Server 2005
 - <http://technet.microsoft.com/library/Cc966402>
- Sample scripts to automate index maintenance
 - <http://ola.hallengren.com/>
 - <http://weblogs.sqlteam.com/tarad/archive/2009/11/03/DefragmentingRebuilding-Indexes-in-SQL-Server-2005-and-2008Again.aspx>
(<http://bit.ly/72ATFB>)
 - <http://sqlfool.com/2011/06/index-defrag-script-v4-1/>

Review

- Data access methods
- What is fragmentation?
- How does fragmentation happen?
- Detecting, mitigating, removing fragmentation

Questions!