

SQLintersection

Session: Tuesday, 10:15am-11:30am

SQL Server Indexing: Strategies for Performance

Kimberly L. Tripp
President / Founder, SQLskills.com
Kimberly@SQLskills.com
@KimberlyLTripp



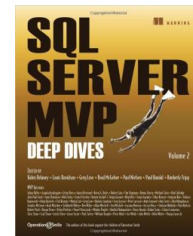
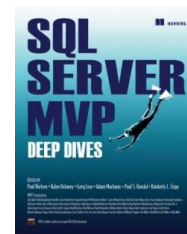
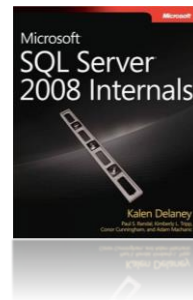
SQL
intersection



Author/Instructor: Kimberly L. Tripp



- Consultant/Trainer/Speaker/Writer
- President/Founder, SYSolutions, Inc.
 - e-mail: Kimberly@SQLskills.com
 - blog: <http://www.SQLskills.com/blogs/Kimberly>
 - Twitter: @KimberlyLTripp
- Author/Instructor for SQL Server Immersion Events
- Instructor for multiple rotations of both the SQL MCM & Sharepoint MCM
- Author/Manager of SQL Server 2005 & 2008 Launch Content
- Author/Speaker at Microsoft TechEd, SQLPASS, ITForum, TechDays, SQLIntersection
- Author of several SQL Server Whitepapers on MSDN/TechNet: Partitioning, Snapshot Isolation, Manageability, SQLCLR for DBAs
- Author/Presenter for more than 25 online webcasts on MSDN and TechNet
- Author/Presenter for multiple online courses at Pluralsight
- Co-author MPress Title: SQL Server 2008 Internals, the SQL Server MVP Project (1 & 2), and SQL Server 2000 High Availability
- Owner and Technical Content Manager of the SQLIntersection conference



PLURALSIGHT

Overview

- **SQL Server version**
- **Workload characteristics**
- **Index structures**
- **Base table structures**
 - Clustered row-based index v. clustered column-based index
 - What criteria should you look for in data access patterns and usage patterns?
- **What makes a good clustered key?**
- **Indexing for Performance: At Design**
- **Indexing for Performance: At QA**
- **Indexing for Performance: In Production**

Biggest Concern is SQL Server Version

- **SQL Server 2008 is the lowest (IMO) version for large tables, performance, scalability**
 - Added data compression (row and page compression)
 - Added filtered indexes / filtered statistics
 - Fixed fast-switching for partition-aligned, indexed views
- **SQL Server 2012 adds read-only, nonclustered columnstore indexes**
 - Some frustrating “batch-mode” limitations for partitioned views (UNION ALL)
 - If you’re using PVs then you should use a minimum of SQL Server 2014!
- **SQL Server 2014 adds updateable, clustered columnstore indexes**
 - Many of the most frustrating limitations with CS fixed – for example, UNION ALL supports batch mode (which means you can use these with partitioned views)
 - Added “incremental statistics” to help reduce when to rebuild as well as time to rebuild
- **SQL Server 2016+ takes columnstore indexes even further with updateable nonclustered columnstore indexes and row-based nonclustered indexes with clustered columnstore indexes!**

What Type of Workload is Running?

- **OLTP – Online transaction processing**
 - Priority is toward modifications
 - Lots of point queries (highly-selective queries of a small number of rows)
Search for a sale, search for a customer, lookup a product, ...
- **Dedicated Decision Support System / Relational Data Warehouse**
 - Priority is toward large-scale aggregates
 - Very large percentage or even the entire dataset – is being evaluated often
- **Hybrid**
 - OLTP might be the priority and some point query activity
 - Some range-based queries because “management” wants real-time analysis

(NOTE: We're talking indexes in this session but in the hybrid environment you should really be running with versioning enabled! Check out the data option: `read_committed_snapshot`. For more info, see resources.

However, **clustered** columnstore indexes do not work in versioned databases until SQL Server 2016.)

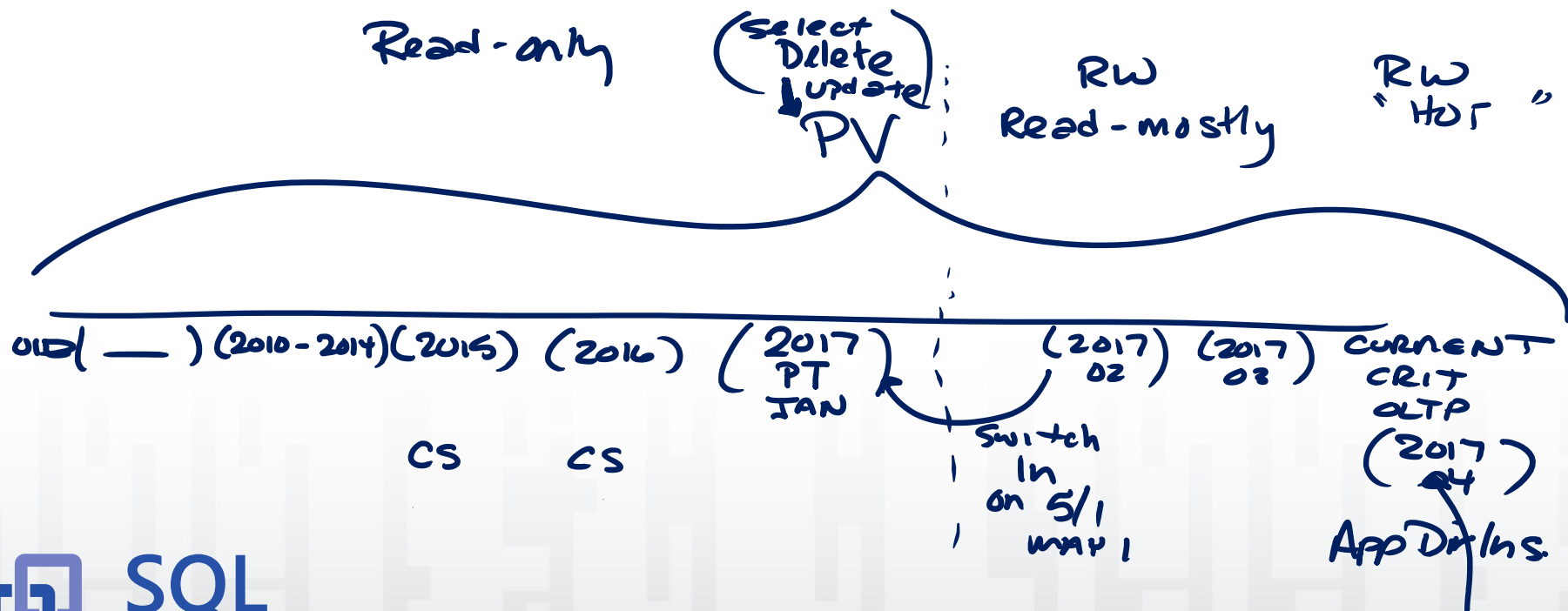
General Indexing Strategies Based on Workload

- **OLTP – Online transaction processing**
 - Traditional row-based clustered and nonclustered indexes
- **Dedicated Decision Support System / Relational Data Warehouse**
 - Prior to SQL Server 2012
 - Traditional row-based clustered and nonclustered indexes (with indexed views)
 - SQL Server 2012+:
 - Consider adding a read-only, nonclustered, columnstore index for partitioned objects leveraging partition switching as additional data is added
 - SQL Server 2014+:
 - Use the SQL Server 2012+ strategy above
 - Or, consider the new, updateable, clustered, columnstore index
 - NOTE: Still quite a few restrictions in 2014 – best to consider 2016 instead!
- **Hybrid**
 - Most likely to use traditional row-based clustered and nonclustered indexes and possibly add a nonclustered, columnstore index for the read-only data – **if** you've partitioned your data in the hybrid scenario

*Be sure to check out my
"hidden slides" for more
rules and restrictions
across versions!*

WHITEBOARD DRAWING: Architecting a “layered” approach to your table design

A better option is to separate the RW from the RO and put the RW in separate tables. Then, you can do online rebuilds at the table level. How do you deal with queries – put an updateable partitioned view over them! Inserts are usually “application directed” using dynamic strings.



Rules, Requirements, and Restrictions (1 of 2)

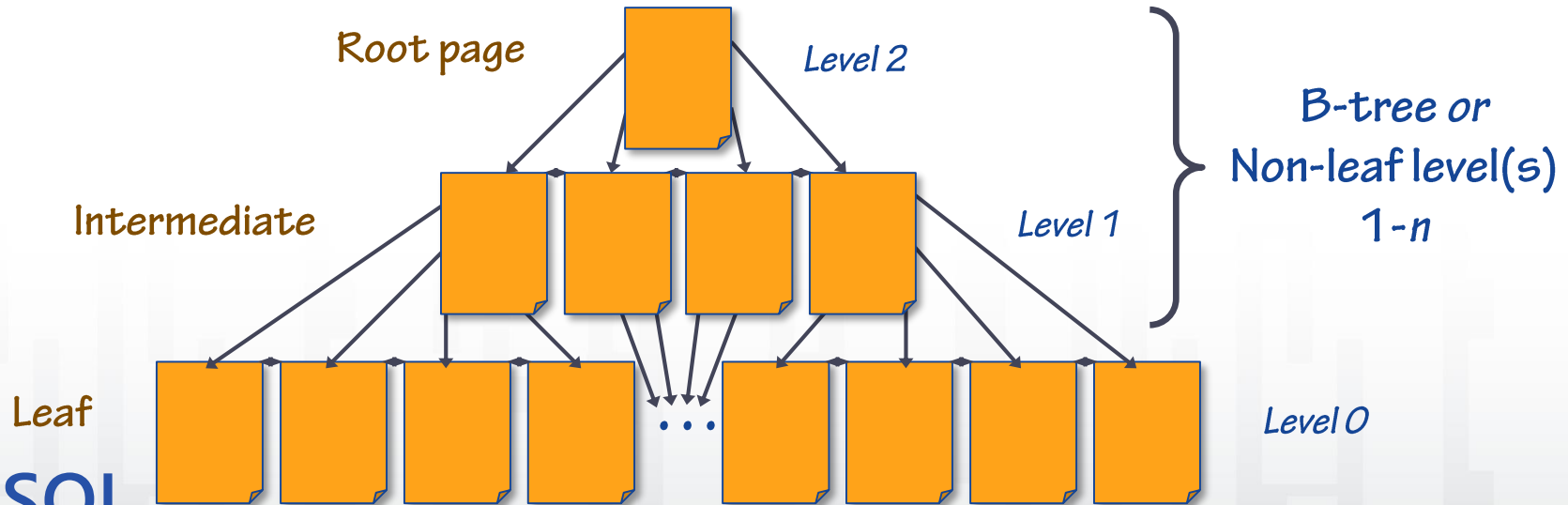
- **Columnstore indexes are improving significantly from version to version**
 - Many restrictions in SQL Server 2012 and SQL Server 2014
 - At this point, SKIP SQL Server 2014 and start designing/prototyping on SQL Server 2016 to go live on 2016 or 2016+
- **SQL Server clustered columnstore indexes DRASTICALLY reduce the overall size of the table (and IOs)**
 - BUT no other indexes were allowed on SQL Server 2014
 - If you do a lot of point queries then clustered columnstore might not be ideal (you'll really need to test to be sure because IOs are so small even scanning a "row group" for a row might not be too bad)
 - SQL Server 2016 offers all combinations / choices
 - Updateable, clustered columnstore index with row-based nonclustered indexes
 - Row-based clustered index with an updateable, nonclustered, columnstore index

Rules, Requirements, and Restrictions (2 of 2)

- If you decide against a clustered columnstore index (because of limitations with SQL Server 2014) then you might go with a row-based, clustered index and then a read-only, nonclustered, columnstore index for read-only partitions (using either partitioned tables or partitioned views)
 - ❑ **Fast-switching concerns:** None! Fast-switching is supported for read-only, columnstore indexes
 - ❑ **Storage concerns:** consider row / data compression for older data (this is nowhere near as efficient as columnstore or columnstore_archive)
 - ❑ **Performance concerns:** performance enhancements around batch-mode processing (UNION ALL) are included in SQL Server 2014+ but NOT in SQL Server 2012
 - ❑ **Version concerns:** SQL Server 2016 doesn't have any of these concerns or limitations

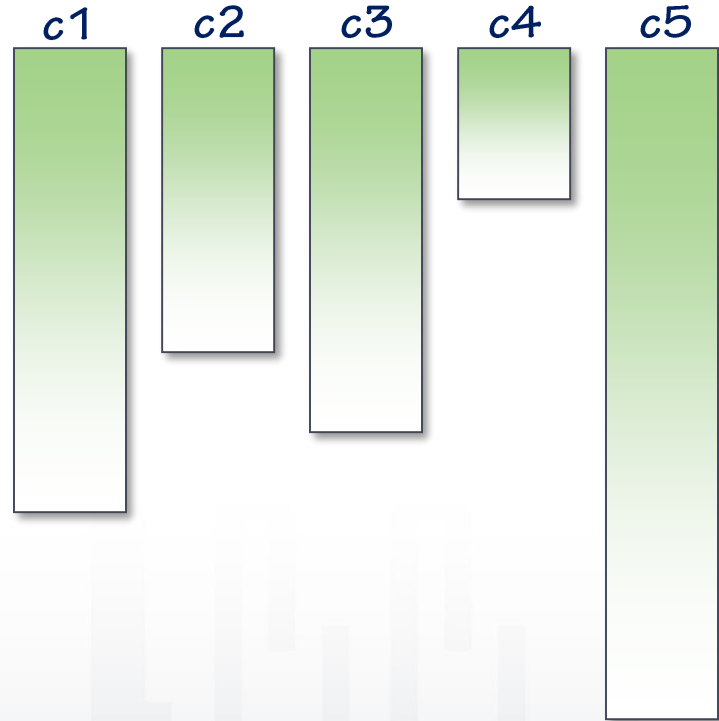
Index Structures: Row-based Indexes

- **Leaf level:** contains something for every row of the table in indexed order
- **Non-leaf level(s) or B-tree:** contains something, specifically representing the FIRST value, from every page of the level below. Always at least one non-leaf level. If only one, then it's the root and only one page. Intermediate levels are not a certainty.



Index Structures: Column-based Indexes


- Column-based
- Only data from that column is stored on the same page – potentially **HIGHLY** compressible!
- Data is segmented into groups of ~1M rows for better batch processing
- SQL Server can do “segment elimination” (similar to partition elimination) and further reduce the number of batch(es) to process!



Row-based Indexes v. Column-based Indexes

- **Supports data compression**
 - Row compressed
 - Page compressed
- **Can support point queries / seeks**
- **Wide variety of supported scans**
 - Full / partial table scans (CL)
 - Nonclustered covering scans (NC)
 - Nonclustered covering seeks with partial scans (NC)
- **Biggest problems**
 - More tuning work for analysis: must create appropriate indexes per query
Must store the data (not as easily compressed)
- **Column-based indexes**
 - Significantly better compression
 - COLUMNSTORE / COLUMNSTORE_ARCHIVE
- **Supports large scale aggregations**
- **Support partial scans w/“segment” elimination**
 - Only the needed columns are scanned
 - Data is broken down into row groups (roughly 1M rows) and segments can be eliminated
 - Combine w/partitioning for further elimination
 - Parallelization through batch mode processing
- **Biggest problems**
 - Minimum set for reads is a row group (no seeks)
 - Limitations of features for batch mode by version
 - Limitations with other features (less and less by SQL Server version)

Base Table Structure Key Points

- **If you're not on SQL Server 2012+ your options are limited to row-based indexes**
- **Then, if you have DSS / RDW workloads with partitioning (partitioned views or partitioned tables):**
 - Create and test a nonclustered, columnstore index on your large fact tables
 - They're super easy to create
 - You can have only one
 - There's no column order
 - It's highly compressed so it doesn't take a lot of space
 - You might get HUGE gains for large scan / aggregate queries – especially those that are narrow (in terms of columns)
 - For better batch processing you must use SQL Server 2014 or higher
- **If you're debating about your next version – don't!**
 - Go straight to SQL Server 2016 (or higher!) 

Columnstore Performance Gains

- **Two complimentary technologies:**

- Storage

- Data is stored in a compressed columnar data format (stored by column) instead of row store format (stored by row)
 - Columnar storage allows for less data to be accessed when only a sub-set of columns are referenced
 - Data density/selectivity determines how compression friendly a column is – example “State” / “City” / “Gender”
 - Translates to improved buffer pool memory usage and fewer IOs

- New “batch mode” execution

- Data can then be processed in batches (1,000 row blocks) versus row-by-row
 - Depending on filtering and other factors, a query may also benefit by “segment elimination” - bypassing million row chunks (segments) of data, reducing I/O

Batch Mode Processing

- **Allows processing of 1M row blocks as an alternative to single row-by-row operations**
 - Enables additional algorithms that can reduce CPU overhead significantly
 - Batch mode “segment” is a partition broken into million row chunks with associated statistics used for Storage Engine filtering
- **Batch mode can work to further improve query performance of a columnstore index, but this mode isn’t always chosen:**
 - Some operations aren’t enabled for batch mode:
 - E.g. outer joins to columnstore index table / joining strings / NOT IN / IN / EXISTS / scalar aggregates / and UNION ALL (meaning PVs)
 - Row mode might be used if there is SQL Server memory pressure or parallelism is unavailable
 - Confirm batch vs. row mode by looking at the graphical execution plan

The Clustering Key

- **For columnstore indexes there's no clustering key defined; however, converting an existing clustered index (cannot be a PK) to a columnstore index can provide benefits in segment elimination (use DROP_EXISTING)**
 - Check out the conference session *Conquering Columnstore Indexes* (by Tim Chapman) for more details on columnstore indexes!
- **For row-based indexes the clustering key is critical for performance**
 - Clustering key defines data order
 - Some clustering keys are more prone to fragmentation
 - Others can have issues with contention
 - Others cause inefficiencies in the nonclustered indexes
 - Clustering key is used for “lookups” into the data
 - This means that nonclustered indexes are actually different when created on a table that has a clustered index (as opposed to a table that does not)
 - This dependency should affect our choice in clustered index!

How is the Clustering Key Used in Nonclustered Indexes?

Imagine the internals of a nonclustered index on SocialSecurityNumber with 3 different versions of the Employee table – each with different clustering keys

| SSN | Lookup | Uniquifier |
|-------------|--------|-------------|
| 000-00-0184 | Smith | 0 (0 bytes) |
| 000-00-0236 | Jones | 1 (4 bytes) |
| 000-00-0395 | Smith | 1 (4 bytes) |
| 000-00-0418 | Jones | 0 (0 bytes) |

The lookup value is non-unique
(and wide as an nvarchar(40)),
what if there are two (or more?)
Smith / Jones / Anderson?

CL: Lastname

| SSN | Lookup |
|-------------|--------------------------------------|
| 000-00-0184 | 92CF41D7-17BF-49F7-B5C8-D3246C19B302 |
| 000-00-0236 | 2F87EEBB-FBA1-4C06-B7F1-BE63285B5935 |
| 000-00-0395 | 2EF09CA4-6E48-47AA-A688-3D9FDEA220E0 |
| ⋮ | ⋮ |

The lookup value
is a GUID = 16 bytes

CL: GUID

| SSN | Lookup |
|-------------|--------|
| 000-00-0184 | 31101 |
| 000-00-0236 | 22669 |
| 000-00-0395 | 18705 |
| ⋮ | ⋮ |

The lookup value
is an int = 4 bytes

CL: EmployeeID

Each table starts at 80,000 rows over 4,000 pages (due to the average row size of 400 bytes/row and therefore 20 rows/page). Then EACH/EVERY index must include the (entire) lookup value.

Lookups

Nonclustered Indexes are Wider!

- Or, what about 100 million rows w/12 nonclustered indexes

| Simple disk space calculations of *JUST* the CL key cost in the NC leaf level! | | | | |
|--|-----------------|-------------|------------|-----------|
| Description | Width of CL key | Rows | NC Indexes | MB |
| int | 7 | 100,000,000 | 12 | 8,010.86 |
| bigint | 11 | 100,000,000 | 12 | 12,588.50 |
| datetime, int | 15 | 100,000,000 | 12 | 17,166.14 |
| datetime, bigint | 19 | 100,000,000 | 12 | 21,743.77 |
| guid | 19 | 100,000,000 | 12 | 21,743.77 |
| composite32, nullable | 35 | 100,000,000 | 12 | 40,054.32 |
| composite64, nullable | 67 | 100,000,000 | 12 | 76,675.42 |

- You're looking at GBs of storage, memory, backups and fundamentally even insert/update performance as well as maintenance requirements.
- My point – it really does add up! It IS something you want to CHOOSE and DESIGN!

Scenario: What is the Real Cost?

AdventureWorksDW: FactInternetSales

- **Clustered index:**
 - SalesOrderNumber **Data type:** `nvarchar(20)`
 - SalesOrderLineNumber `tinyint`
- **Nonclustered indexes:**
 - IX_FactInternetSales_ShipDateKey: ShipDateKey
 - IX_FactInternetSales_CurrencyKey: CurrencyKey
 - IX_FactInternetSales_CustomerKey: CustomerKey
 - IX_FactInternetSales_DueDateKey: DueDateKey
 - IX_FactInternetSales_OrderDateKey: OrderDateKey
 - IX_FactInternetSales_ProductKey: ProductKey
 - IX_FactInternetSales_PromotionKey: PromotionKey

Demo

AdventureWorks

The impact of key choice on nonclustered indexes



SQL
intersection

Clustered Index Criteria

Keeping our Clustering Key as Streamlined as Possible!

- **Unique**
 - Yes: No extra time/space overhead, data takes care of this criteria
 - NO: SQL Server must “uniquify” the rows on INSERT
- **Static**
 - Yes: Reduces overhead
 - NO: Costly to maintain during updates to the key
- **Narrow**
 - Yes: Keeps the NC indexes narrow
 - NO: Unnecessarily wastes space
- **Non-nullable/fixed-width**
 - Yes: Reduces overhead
 - NO: Adds overhead to ALL nonclustered indexes
- **Ever-increasing**
 - Yes: Reduces fragmentation
 - NO: Inserts/updates might cause splits (significant fragmentation)

 *added slide
w/extra details*

Clustering on an Identity

 *added slide
w/extra details*

- **Naturally unique**
 - (should be combined with constraint to enforce uniqueness)
- **Naturally static**
 - (should be enforced through permissions and/or trigger)
- **Naturally narrow**
 - (only numeric values possible, whole numbers with scale = 0)
- **Naturally non-nullable/fixed-width**
 - (an identity column cannot allow nulls, a numeric is fixed-width)
- **Naturally ever-increasing**
 - Creates a beneficial hot spot...
 - Needed pages for INSERT already in cache
 - Minimizes cache requirements
 - Helps reduce fragmentation due to INSERTs
 - Helps improve availability by naturally needing less defrag

Clustering Key Suggestions

- **Identity column**
 - Adding this column and clustering on it can be extremely beneficial – even when you don’t “use” this data
- **DateCol, identity**
 - In that order and as a composite key (not date alone as that would need to be “uniquified”)
 - Great for very large tables / partitioned objects
 - Great for ever increasing tables where you have a lot of date-related queries
- **GUID**
 - NO: if populated by client-side call to .NET client to generate the GUID. OK as the primary key but not as the clustering key
 - NO: if populated by server-side NEWID() function. OK as the primary key but not as the clustering key
 - Maybe: if populated by the server-side NEWSEQUENTIALID() function as it creates a more sequential pattern (and therefore less fragmentation)
 - But, this isn’t really why you chose to use a GUID...
- **Key points: unique, static, as narrow as possible, and less prone to require maintenance – by design**

Nonclustered Indexes: Key to Better Performance

- **In a row-based indexing strategy performance hinges on your choice of nonclustered indexes:**
 - Indexing strategies are extremely challenging
 - Users lie 😊
 - Workload specific
 - Data modifications are impacted by indexes (indexes add overhead to INSERTs/UPDATEs/DELETEs)
 - The type and frequency of the queries needs to be considered
 - This can change over time
 - This can change over the course of the business cycle
 - Need to have an understanding of how SQL Server works in order to create the “RIGHT” indexes, you CANNOT just guess!
 - **To do a good job at tuning you must:**
 - Know your data
 - Know your workload
 - Know how SQL Server works!

Indexing for Performance: At Design

- **First and foremost: choose a GOOD clustering key**
- **Create your primary keys and unique keys**
- **Create your foreign keys**
 - Manually index your foreign keys with nonclustered indexes
- **Create any nonclustered indexes needed to help with highly selective queries (lookups are OK for highly selective queries)**
- **STOP: this is your “design” base**
- **Add indexes slowly and iteratively over time while learning and understanding your workload as well as query priorities and always re-evaluate if/when things change!**
 - Better done in QA / Production...

Indexing for Performance: At QA

- **While testing primary workload characteristics monitor query performance:**
 - By duration and IO (at a minimum)
 - Review the cumulative costs of frequently executed queries
 - sys.dm_exec_query_stats (also review query_hash)
 - sys.dm_exec_procedure_stats (this is cumulative)
- **Identify key performance problems**
- **Consider wider indexes through testing / analysis**
 - Be sure to evaluate the impact to OLTP as wider indexes are added
 - Be sure to evaluate the disk / memory costs for wider indexes

Indexing for Performance: In Production

1. Make sure you don't have any "dead weight"

- ❑ Remove duplicate indexes
- ❑ Consider the removal or consolidation of redundant indexes

2. Make sure you have a good maintenance strategy

- ❑ How are you analyzing for fragmentation?
 - ❑ Use a limited scan (no need for "sampled" or "detailed")
- ❑ Are you dealing with statistics appropriately?
 - ❑ Rebuilding indexes updates the statistics of the rebuilt index with the equivalent of a full scan but does not update other statistics
 - ❑ Reorganizing indexes does NOT update statistics at all
 - ❑ Do you have any other statistics (column-level statistics?)

3. Only after steps 1 and 2 are done can you add indexes slowly and iteratively over time while learning and understanding your ACTUAL workload as well as query priorities. And, always re-evaluate if / when things change!

Indexing Key Points

- Long term scalability doesn't happen by accident
- SQL Server has a tremendous number of indexing options available but they all have trade-offs
- Prototyping plus early analysis is critical to getting it right
 - Can learn where combinations of features do or don't work well together
 - Can see the disk space requirements and do estimates to scale
- The sooner you begin the code, the longer it's going to take!

Review

- **SQL Server version**
- **Workload characteristics**
- **Index structures**
- **Base table structures**
 - Clustered row-based index v. clustered column-based index
 - What criteria should you look for in data access patterns and usage patterns?
- **What makes a good clustered key?**
- **Indexing for Performance: At Design**
- **Indexing for Performance: At QA**
- **Indexing for Performance: In Production**

Resources

- **Demo code/samples: SQLskills, Resources, Demo Scripts and Sample Databases**
- **Courses on Pluralsight: www.pluralsight.com**
 - ❑ SQL Server: Why Physical Database Design Matters
 - ❑ SQL Server: Optimizing Ad Hoc Statement Performance
 - ❑ SQL Server: Optimizing Stored Procedure Performance (Parts 1 and 2)
 - ❑ Part 2 has an entire section on session settings (for performance-related features)
 - ❑ SQL Server: Indexing for Performance (7+ hours)
 - ❑ Coming soon: SQL Server: Query Tuning with Indexes
 - ❑ If you want to know more about columnstore indexes then check out Joe Sack's Pluralsight course on SQL Server 2012's read-only, nonclustered columnstore indexes:
 - ❑ SQL Server 2012: Nonclustered Columnstore Indexes (<http://bit.ly/1PYVU2a>)



PLURALSIGHT

Resources

- **Paul's index "fanout" blog post: On index key size, index depth, and performance**
 - <http://www.sqlskills.com/blogs/paul/on-index-key-size-index-depth-and-performance/>
- **Additional columnstore resources:**
 - ColumnStore Index: Microsoft SQL Server 2014 and Beyond
 - <https://channel9.msdn.com/Events/Ignite/2015/BRK4556>
 - SQL Server 2014: Security, Optimizer, and Columnstore Index Enhancements
 - <http://www.microsoftvirtualacademy.com/training-courses/sql-server-2014-security-optimizer-and-columnstore-index-enhancements?prid=ch9courselink>
 - Niko Neugebauer blog post series on Columnstore indexes:
 - <http://www.nikoport.com/columnstore/>
- **BI Foundations Sessions from PASStv**
 - <http://www.sqlpass.org/summit/2015/PASStv/Microsoft.aspx>

Questions?

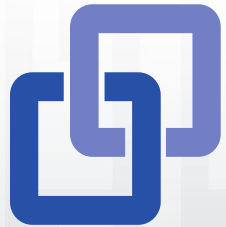


Don't forget to complete an online evaluation!

SQL Server Indexing Strategies for Performance

Session by Kimberly L. Tripp

Your evaluation helps organizers build better conferences
and helps speakers improve their sessions.



SQL
intersection

Thank you!