

SQLintersection

Post-Conference Session: Friday, June 14

Zero To Hero: Faster SQL Query Performance

Jonathan Kehayias

Jonathan@SQLskills.com





Jonathan Kehayias

Principal Consultant, SQLskills



Jonathan@SQLskills.com



[@SQLPoolBoy](https://twitter.com/SQLPoolBoy)



www.sqlskills.com/blogs/jonathan

Trainer/Speaker

In addition to consulting, I teach content for our IE0: Accidental DBA course, IECAg: Clustering and Availability Groups course, and our IEPTO2: Performance Tuning and Optimization course

Data Platform MVP

I have been thankfully recognized as an MVP by Microsoft since 2008

Author

Microsoft SQL Server 2012 Internals

Professional SQL Server 2008 Internals and Troubleshooting

Troubleshooting SQL Server: A Guide for the Accidental DBA

Reminder: Intersect with Speakers and Attendees

- **Tweet tips and tricks that you learn and follow tweets posted by your peers!**
 - Follow: #SQLIntersection and/or #DEVIntersection
- **Join us – Wednesday Evening – for SQLafterDark**
 - Doors open at **7:00 pm**
 - Trivia game starts at **7:30 pm**
 - Winning team receives something fun!*
 - Raffle at the end of the night
 - Lots of great items to win including a seat in a SQLskills Immersion Event!*
 - The first round of drinks is sponsored by SentryOne and SQLskills



Overview

- **How SQL Server Processes Queries**
- **Normalization and Datatypes**
- **Reading Execution Plans**
- **Designing Set Based Solutions**
- **Scalability and Testing**
- **New Features vs Old Tricks**

How SQL Server Processes Queries

Row Mode

- Efficient for OLTP workloads
- Typical for row store format tables
- Execution tree operators read each required row across all columns

Batch Mode

- Efficient for data warehouse and scanning large amounts of data
- Typical for columnstore format tables
- Processes multiple rows as a batch and eliminates the need for an exchange operator during parallel processing

SELECT Statements

- Define the result set format and columns of data to return in the column list
- Define the tables that contain the source data in the FROM and JOIN clauses
- Defines how the tables logically are related to one another in the ON clause following a JOIN or the WHERE clause
- Defines the filtering conditions for the output rows in the WHERE or HAVING clause of a GROUP BY
- Does not define how SQL Server will execute the request unless specific hints are used (non-procedural)

Query Execution Plans

- **Parsing**

- Syntax is checked to determine if the query is written properly
- Produces a parse tree that is handed off

- **Binding**

- Algebrizer analyzes the parse tree for name resolution
 - Do the tables exist, are columns in the where clause in the tables, permissions
- Produces a query processor tree that is then handed to the optimizer

- **Optimization**

- The optimizer attempts to determine the most efficient method of executing the request based on costs
- Produces an execution plan that is used by the execution engine to process the request
- The goal of query optimization is not to find BEST execution plan, but to find a *good enough* execution plan fast

Operator Precedence

Level	Operators
1	~ (Bitwise NOT)
2	* (Multiplication), / (Division), % (Modulus)
3	+ (Positive), - (Negative), + (Addition), + (Concatenation), - (Subtraction), & (Bitwise AND), ^ (Bitwise Exclusive OR), (Bitwise OR)
4	=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	= (Assignment)

Normalization

- **Process of organizing data in a database into tables with relationships to other tables**
 - Better flexibility – design changes
 - Reduce redundancy – wasted disk/memory space
 - Enforce data integrity – reduce where data changes must occur
 - Remove inconsistent dependencies – group data into related subjects
- **Rules define a standard of database normalization**
 - First Normal Form
 - Second Normal Form
 - Third Normal Form – typical for OLTP

First Normal Form (1NF)

- **Eliminate repeating groups in individual tables**
 - Does not store multiple values within a single column
 - Defines each value as atomic – cannot be broken to smaller pieces
- **Create a separate table for each set of related data**
 - Does not use multiple columns in a table to store similar data
 - E.g. Phone Numbers, vendor codes
- **Identify each set of related data with a primary key**
 - 1NF requires a unique constraint on the table - no exact duplicate rows exist

First Normal Form (1NF)

Name	Address	Phone1	Phone2	PostalCode	Country
Jonathan Kehayias	123 Disney Hwy, Orlando, FL 34582	407-528-1124	727-485-3325	34582	USA
Paul Randal	36 BAOSHAN JIUCUN, BAOSHAN DISTRICT	251-112-2425	927-555-1212, 855-243-8514	201900	CHINA

ID	Name	Address	Phone
1	Jonathan Kehayias	123 Disney Hwy	407-528-1124
2	Jonathan Kehayias	123 Disney Hwy	727-485-3325
3	Paul Randal	4432 Nowhere Lane	251-112-2425
4	Paul Randal	4432 Nowhere Lane	927-555-1212
5	Paul Randal	4432 Nowhere Lane	855-243-8514

Second Normal Form (2NF)

- **Must first meet the requirements of 1NF**
- **Create separate tables for sets of values that apply to multiple records**
 - Records should only depend on another table's primary key
 - E.g. Addresses stored in an address table and referenced by AddressKey in Customer, Orders, Shipping, etc tables
- **Define relationships between tables with a foreign key**
 - Enforce data integrity and prevent orphaned data

Second Normal Form (2NF)

PersonID	Name	Address	PostalCode	Country
1	Jonathan Kehayias	123 Disney Hwy	34582	USA
2	Paul Randal	36 BAOSHAN JIUCUN, BAOSHAN DISTRICT	201900	China

PhoneID	PersonID	PhoneNumber
1	1	407-528-1124
2	1	727-485-3325
3	2	251-112-2425
4	2	927-555-1212
5	2	855-243-8514

Third Normal Form (3NF)

- **Ideal for OLTP applications**
 - Reduces duplication of data
 - Typically free of INSERT/UPDATE/DELETE anomalies
- **Must first meet the requirements of 2NF**
- **Eliminates columns that do not depend on the key**
 - E.g. Cities, states, zip codes
 - May not always be followed for every table (see above examples for addresses)
 - Should be applied for frequently changing data that affects multiple relationships or records – only update parent table field

Third Normal Form (3NF)

PersonID	Name	Address	PostalCodeID
1	Jonathan Kehayias	123 Disney Hwy	1
2	Paul Randal	36 BAOSHAN JIUCUN	2

PhoneID	PersonID	PhoneNumber	PostalCodeID	PostalCode	Country	City	Provence
1	1	407-528-1124	1	34582	USA	Orlando	FL
2	1	727-485-3325	2	201900	China	BAOSHAN DISTRICT	Shanghai
3	2	251-112-2425					
4	2	927-555-1212					
5	2	855-243-8514					

Use Appropriate Data Types

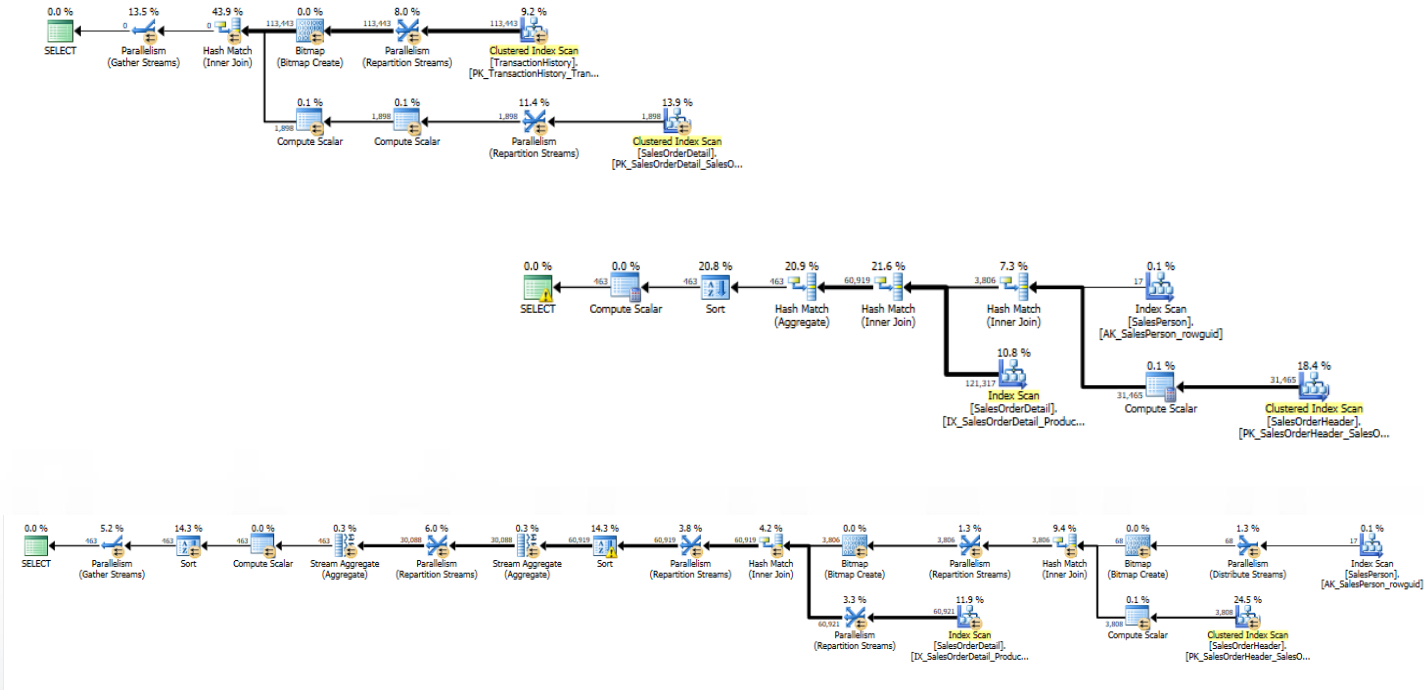
- Understand the storage costs and implications of data types during schema design – especially for keys

Data type	Range	Storage
bigint	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 Bytes
int	-2,147,483,648 to 2,147,483,647	4 Bytes
smallint	-32,768 to 32,767	2 Bytes
tinyint	0 to 255	1 Byte

Use Appropriate Data Types (2)

- **Consider precision requirements for dates and times**
 - DATETIME2 = 6, 7, or 8 bytes with nanosecond precision
 - 6 bytes for precisions less than 3
 - 7 bytes for precisions 3 and 4
 - All other precisions require 8 bytes
 - DATETIME = 8 bytes with fractions of second precision
 - Fraction of seconds rounded to .000, .003, or .007 seconds
 - SMALLDATETIME = 4 bytes with minute precision
 - DATE = 3 bytes
- **Don't store dates or times as CHAR, VARCHAR, NCHAR, or NVARCHAR**

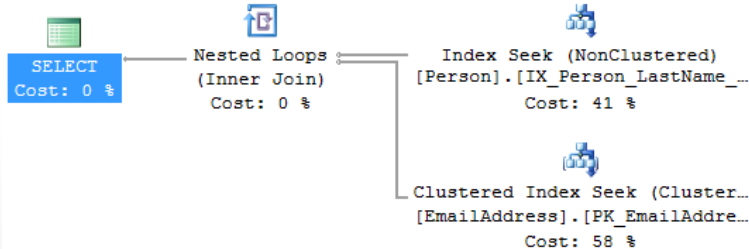
Where Do You Start with a Execution Plan?



Reading Plans

- An operator reads rows from a leaf-level data source OR from child operators and return rows to the parent
- Control flow starts at the root (left-to-right)
- Data flow starts at the leaf level (right-to-left)

Query 1: Query cost (relative to the batch): 100%
`SELECT p.FirstName, p.LastName, e.EmailAddress FROM Person.Per`



Control flow

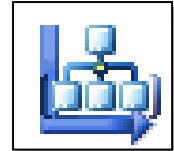
Data
flow

Operators

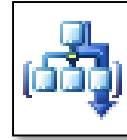
- **Operators are the building blocks for an execution plan**
 - Each operator has a specific functionality – access data (scan, seek), perform aggregations or join data sets
 - One-to-many mapping of logical to physical operations
 - E.g. an INNER JOIN could be implemented as a Loop, Hash, or Merge join
- **No specific operator is “good” or “bad” for performance**
 - Certain operators have more overhead and consume more resources
 - Some are more appropriate in given contexts
- **SQL Server 2017 has 100+ operators**
- **Operators may also be referred to as iterators**

Table and Index Scans

- **Table Scan: indicating a retrieval of ALL rows from a table**
 - Indicates a heap table
 - Is it a red flag?
 - Probably for larger tables (I/O)
- **Clustered Index Scan: indicating a retrieval of all rows**
 - Is it a red flag?
 - If it's a large table or you expect a seek operation
- **What about nonclustered index scan of leaf level?**
 - Depends on the size; it may or may not be an issue



Index Seeks



- **Clustered Index Seek**
 - Retrieving rows based on a SEEK predicate from clustered index
- **Nonclustered Index Seek**
 - Same, but from a nonclustered index
- **There is nothing in the query plan that differentiates between singleton or range scan operations**
 - Can use `sys.dm_db_index_operational_stats` to determine
 - `range_scan_count`
 - `singleton_lookup_count`

Filter



- **Predicates can be evaluated within operators that read data from table/indexes**
- **Query Optimizer aims (when possible) to “push” filter down the tree (leaf level) to reduce rows moved**
- **If a predicate is high in cost or complexity a separate Filter operator may be used**
- **When you see these, take note of where they are happening**
 - Late in the data flow can translate to higher overhead as the operators pull data

Predicates

- **Seek Predicate**

- Used in actual index seek operation
 - Leveraging index keys

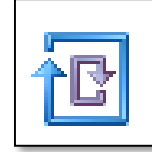
- **Predicate (Residual Predicate)**

- Search condition that isn't SARGable – so it remains as an extra predicate
 - For Merge Join: check Graphical Showplan Properties for “Residual” value
 - For Hash Match: check “Probe Residual” value in plan itself (tooltip over operator)

Join Considerations

- **Beware of advice telling you that specific join types (or operators, for that matter) are “good” or “bad”**
- **Join hints and/or forcing order = red flag**
 - Generally, “edge” cases or extreme tuning scenarios warrant their use
 - Otherwise, ask questions and find out why this is happening

Nested Loop

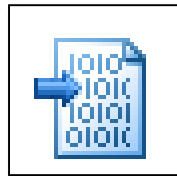


- **Uses each row from one input to find rows from a second input that satisfy the join predicate**
- **Usually seen with smaller data sets and lookups, where the inner input is indexed on the join predicate**
- **Algorithm:**
 - For one row in the outer (top) table, find matching rows in the inner (bottom) table and return them
 - After no matching rows on the inner table are found, retrieve the next row from the outer (top) table and repeat until end of outer (top) table rows

Nested Loop Join Performance Characteristics

- **Look for “smaller” table as outer (top) table**
 - Bad cardinality estimates can lead to this NOT being the case
- **Nested Loops may be associated with inflated random I/Os when the row estimates end up being incorrectly estimated**
- **Look for under-estimates for inner table or index scans**
- **Memory requirements are lower comparatively**

Key Lookup



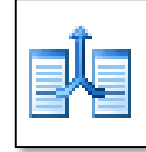
- **A.k.a. Bookmark Lookups**
- **Key Lookup = bookmark lookup on table with clustered index (always via Nested Loop)**
 - If you see WITH PREFETCH then QP is using read-ahead
- **Is this good or bad?**
 - For each row in the non-clustered index, an associated clustered index I/O is required (random I/O)
 - Even if all pages are cached, you can STILL have inflated overhead (compared to a covering index) due to the increased number of random logical reads
- **May be prone to deadlock conditions**

RID Lookup



- **Simply a bookmark lookup to a heap (using the RID)**
 - Just like with Key Lookups, you'll only see this with Nested Loop Joins
- **Is it good or bad?**
 - Same considerations as a Key Lookup
 - You also may research good vs. bad because you're going against a heap

Merge Join

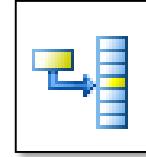


- **Joins two inputs which are sorted on the joining columns and returns matching rows**
 - Typically benefits moderate-sized data sets
- **Algorithm:**
 - Retrieve row from the outer input
 - Advance through the inner input until no more matches are found
 - Retrieve the next row from the outer input and repeat
 - Note: worktables are needed to support many-to-many merge joins (outer input is not distinct)

Merge Join Performance Characteristics

- **Outer (top) / inner (bottom) requires sort on join key**
- **Pre-existing sorting (via index) is ideal, but sorts can be automatically added**
- **If the sort is injected into the plan by the Query Optimizer, take note of it**
 - Query Optimizer injected sorts have a risk of spilling to disk (tempdb)
- **Memory requirements are generally lower**
 - Many-to-many joins have overhead in the form of worktables
 - Look for ManyToMany attribute

Hash Match Join



- **Joins two unsorted inputs and outputs the matching rows**
 - Often seen with large data sets
 - Grouping aggregates
- **Algorithm:**
 - Build a hash table (hash buckets) via computed hash key values for each row of the “build” input (top/outer table)
 - For each probe row (bottom/inner table), compute a hash key value and evaluate for matches in the “build” hash table (buckets)
 - Output matches (or output based on logical operation)

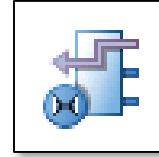
Hash Match Join Performance Characteristics

- **Doesn't require ordering of inner or outer inputs**
- **Hash table must be generated FIRST before the probe begins, and this is a blocking operation**
- **Typical case is that the smaller table is the “build table”, which ideally reduces the latency between the build and probe phases**
 - Red flag if you see otherwise
- **Hash build or probe can spill to disk if there is insufficient memory (higher memory requirements)**

Hash Joins: Performance Variations

- **SQL Server can also do “role reversal”**
 - \geq one spill, build/probe roles can be switched
 - Not visible to us and should be rare
- **Hash Warning events can be found in Extended Events (also in Trace) and spill notifications are within the actual plan from SQL Server 2012 onwards**
- **Reasons for spills include cardinality estimate issues (skewed data distributions, missing or stale statistics), inappropriate join selection or memory pressure**
 - Estimates based on both cardinality and average row size

Batch Mode Adaptive Join



- **New operator in SQL Server 2017+: Adaptive Join**
 - Requires compatibility mode 140
- **Indicates that the optimizer has the choice of a Hash Join or Nested Loop**
 - Decision is deferred until after the first input is scanned
 - Threshold established by the adaptive join determines at what point a plan will switch to a nested loop
- **Plan is still cached, join type is determined at run-time**
- **Ideal for workloads with varied inputs/skewed data**
- **Applies to SELECT statements only**

Sort



- **As named, this operator orders rows received from an input**
- **Variations include:**
 - Distinct Sort
 - Top N Sort
- **Noteworthy: Sort tempdb spills**
- **Keep an eye on these for the following reasons:**
 - Sort can occur in tempdb for large data sets and memory constraints (see Sort Warnings)
 - Has resource overhead (CPU / I/O / memory)
 - May not be needed if you have supporting indexes or unnecessary ORDER BY

Query Memory

- **Some queries require memory to store data while sorting and joining rows, thus a memory grant is requested**
 - Lifetime of the grant is equivalent to the lifetime of the query
- **Pay attention to heavy memory-consuming operators:**
 - Hash operations (JOIN and Aggregations) and Sort
- **When available memory is insufficient, queries that require lots of memory may wait to execute (RESOURCE_SEMAPHORE wait type)**
 - Under-estimating memory (due to cardinality estimation issues) can cause spills to tempdb (I/O)
 - Over-estimating memory can reduce concurrency!

Controlling Memory Grants

- **Query OPTION min_grant_percent and max_grant_percent available in SQL Server 2012 SP3, SQL Server 2014 SP2, and SQL Server 2016**
 - min_grant_percent is guaranteed to the query
 - Overrides the sp_configure option (minimum memory per query (KB)) regardless of the size
 - Be very careful with this....
 - max_grant_percent is the maximum limit for a query
 - <https://support.microsoft.com/en-us/kb/3107401>

```
SELECT [CustomerID], [SalesOrderID], [OrderDate], [SubTotal]
FROM [Sales].[SalesOrderHeaderB]
WHERE [OrderDate] BETWEEN '2012-01-01 00:00:00.000'
      AND '2013-12-31 23:59:59.997'
ORDER BY [OrderDate]
OPTION (min_grant_percent = 20, max_grant_percent = 50);
```

Designing Set Based Solutions

- **The most common bottleneck for database efficiency is loop based development thinking**
- **Think of data vertically in sets rather than horizontally in rows for query concepts**
 - Aggregations with GROUP BY
 - Common Table Expressions and Window Functions

Row-By-Agonizing-Row Processing

- **Certain constructs force SQL Server into RBAR (row-by-agonizing-row) processing of results**
- **Well-known:**
 - WHILE loops
 - Cursors
- **Less well-known:**
 - Scalar user defined functions (changes in SQL Server 2019)
 - Correlated subqueries

Cursors and Loops

- **Characteristics**

- Explicit cursor declaration
- WHILE loop
- SqlDataReader in application

- **Problems**

- Row based processing over Set Based

- **Replace with**

- Appropriate set based operation
- Move looping code into SQLCLR or Middle Tier
- Consume and dispose of SqlDataReader as quickly as possible

Correlated Sub-Queries

- **Characteristics**

- Refer to the outer query in the inner query in SELECT statement
- SELECT Statement used as column value in UPDATE

- **Problems**

- May cause row-by-row processing to occur
- Performance decreases exponentially as row count increases

- **Replace with**

- Table Join
- Derived Table Join
- Cross Joined Table Valued Function

Scalar User Defined Functions

- **Characteristics**

- Encapsulate common code blocks/business logic in a single call.
- If columns are passed as parameters it is not inline

- **Problems**

- Cause row-by-row processing to occur
- Performance decreases exponentially with data access

- **Replace with**

- Inline expressions
- Derived Table Join
- Cross Joined Inline Table Valued Function

Scaling Row Based Processing

- **Application design patterns**

- Multi-threaded apartment for asynchronous execution
- Retrieve initial data set from SQL Server and execute row based processes on background or “worker” class threads in the application
 - Best for situations where initial data set is not changing
 - Watch for blocking conditions for long open result sets – fire hose cursor consumption of data

- **SQL Server design pattern**

- Service Broker queue activation allows parallel processing of messages natively within SQL Server
- Transactional based and ensures restart of processing after failure
- Implemented entirely with TSQL as a part of the database

Sargability Matters

- **A query is sargable (Search ARGument ABLE) if an index seek can be used to speed up the execution of the query**
- **Anti-patterns to sargable expressions include:**
 - Functions in the WHERE clause
 - Implicit/Explicit data type conversions on a column
 - Leading wildcard expressions with LIKE '%<SearchTerm>'
 - Catch all queries and search procedures

Functions on WHERE Clause Columns

- **Characteristics**

- Used to change the data stored to match criteria being checked
- Conversion of data to a different type

- **Problems**

- Causes Table/Index Scan over Seek

- **Replace with**

- Appropriate Table Design to support business needs
- Indexed/Persisted Computed Column
- Indexed View
- Other coding paradigm to eliminate Scan

Implicit/Explicit Column Conversions

- **Characteristics**

- Column data type is of lower precedence than filtering parameter / joining column data type
- Common in LINQ to SQL/EF and other ORMs

- **Problems**

- Causes Table/Index Scan over Seek

- **Replace with**

- Higher precedence column data type
- Matching data type for filtering parameter

Catch-All Search Queries

■ Characteristics

- ❑ Used to search across multiple columns using parameters
- ❑ Not all parameters require input values
- ❑ WHERE clause similar to (`@Param1 IS NULL OR Column1 = @Param1`)

■ Problems

- ❑ No optimized execution plan
- ❑ Causes Table/Index Scan over Seek

■ Replace with

- ❑ Separate search procedures for different parameters passed
- ❑ Parameterized Dynamic SQL

Replacing IN with UNION

- **Characteristics**

- WHERE clause similar to (Column 1 IN (12, 16))
- WHERE clause similar to (Column1 = 12 OR Column1 = 16)

- **Problems**

- May causes Table/Index Scan over Seek
- May cause a range seek

- **Replace with**

- Separate SELECT statements with WHERE clause for each criteria using UNION ALL to concatenate results to final output
- Dynamic SQL to build the UNION ALL query string

Profile Your Workload During Development

- **Learn to use Extended Events (2012+) or SQL Trace to profile your workload during testing**
- **Know the important events to watch for during development**
 - Statement/Batch/RPC completed events
 - SP completed/Module End events – (procedure/trigger/function executions)
 - Execution warnings (sort, hash, missing join predicate)
- **Profiling during development can uncover nasty RBAR issues and performance effecting side effects of trigger executions**
- **Be aware of “*observer overheads*” but not typically a problem with development/test workloads**

Develop and Test Against Realistic Scale

- **Development databases often do not contain realistic datasets which can hide/mask potential performance problems**
 - Key Lookups on small data sets may become index scans on larger data sets
 - Missing index impacts may be hidden by data residing in memory for small data sets
 - RBAR problems are often hidden until data sizes scale up
- **Testing a single execution in isolation is not load testing**
 - Testing needs to be performed at scale through load generation to measure accumulated effects
 - Only testing at scale can identify “*death by 1000 cuts*” problems

New Features vs Old Strategies

- **New features in SQL Server help solve common performance problems out of the box**
 - IMOLTP
 - Availability Group Readable Secondaries
 - Columnstore Indexes
- **New features won't scale poor design architecture**
 - Know the problem a feature is intended to solve
 - Know the trade-offs of using new features – an old design strategy might still be a better fit...

In-Memory OLTP

- **Designed to eliminate PAGELATCH contention and locking contention for high throughput workloads**
- **Use Cases:**
 - High volume data ingestion
 - Caching
 - Session state
 - ETL
 - Temporary object replacement

In-Memory OLTP Considerations

- **Crash recovery time**
 - All memory optimized objects must be loaded into memory before the database can be opened for use
 - If the server does not have enough memory, the database will go suspect
- **Optimizations are for write heavy workloads specifically**
 - Reads from Buffer Pool are already memory resident and low latency
 - SELECT queries achieve marginal performance gains from IMOLTP
 - Consider SQL Server 2019 Hybrid Buffer Pool on PMEM devices instead

Designs Before IMOLTP

- **Use heaps and bulk loading strategies for parallel loading**
- **Hash based partitioning of tables for fast ingestion of data**
 - Presents challenges with querying data but eliminates hot page contention
- **Standard partitioning based on data values or sources**
- **Adding a padding column to fill page**
 - Reduces latch contention due to one row per page
- **Sharding with federated views**
 - Partitions data across multiple tables/databases based on design

Availability Group Readable Secondaries

- **Allows offloading of read-only workloads to secondary replicas inside an Availability Group**
- **Native round-robin load balancing support of read-only connections**
- **Distributed Availability Group configurations allow scaling beyond 8 replicas**

Readable Secondary Considerations

- **Requires FULL recovery model for the databases**
- **Requires synchronizing the entire database whether the data is needed by the workload or not**
 - Storage requirements are the same for each replica
- **Indexes must be created on primary replica for readable workloads**
- **Query Store data is only generated on primary replica**

Designs Before Availability Groups

■ Transactional Replication

- ❑ SIMPLE recovery model allowed
- ❑ Filtered publications can distribute/limit data set sizes
- ❑ Subscribers can have different indexes from primary keeping primary small
- ❑ Query Store available to subscriber workload for analysis and forcing plans

■ Scalable Shared Database

- ❑ Leverages a SAN snapshot mounted in read-only mode to multiple SQL Servers – updates require new snapshot
- ❑ Requires proper I/O sizing to avoid performance bottlenecks

Clustered Columnstore Usage Scenarios

- **Data warehouse fact tables**

- ❑ Queries primarily perform aggregations/analytics on ranges of values
- ❑ Tables are typically partitioned with at least one million rows per partition
- ❑ Data loading typically by ETL and bulk operations

- **IOT data for compression**

- ❑ Unstructured LOB data stored as JSON in SQL Server
- ❑ Compression ratios as high as 25x vs. rowstore (100GB data in 4GB)

Questions To Ask For Columnstore

- **How large is my table/data?**
- **Do my queries mostly perform analytics that scan large ranges of values?**
- **Does my workload perform lots of updates and deletes?**
- **Do I have fact and dimension tables for a data warehouse?**
- **Do I need to perform analytics on a transactional workload?**
- **What version of SQL Server am I running on?**

These determine whether columnstore is the right solution!

Questions To Ask For Columnstore

- **How large is my table/data?**
 - Compression may provide significant space and I/O savings
- **Do my queries mostly perform analytics that scan large ranges of values?**
 - Columnstore works best for large range scans and not single row values
- **Does my workload perform lots of updates and deletes?**
 - Columnstore works best on stable/static data, typically < 10% DELETE/UPDATE
- **Do I have fact and dimension tables for a data warehouse?**
 - Schema design and data loading strategy affects columnstore effectiveness
- **Do I need to perform analytics on a transactional workload?**
 - Updatable nonclustered columnstore indexes with filter criteria on “warm” data

BLUF: You can't blindly implement columnstore indexes!

Designs Before Columnstore

- Traditional data warehouse Star schema
- Table partitioning and partition switching for data loading
- Filtered indexes and statistics
- Indexed Views for aggregations (NOEXPAND hint)
- Partitioned Views over multiple tables with check constraints

Review

- **How SQL Server Processes Queries**
- **Normalization and Datatypes**
- **Reading Execution Plans**
- **Designing Set Based Solutions**
- **Scalability and Testing**
- **New Features vs Old Tricks**

Questions?



Don't forget to complete an online evaluation!

Zero To Hero: Faster SQL Query Performance

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



Thank you!

Save the Date

www.SQLintersection.com

Week of November 18, 2019
We're back in Vegas baby!

