SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Module 7: Putting New Features into Practice

Erin Stellato
Erin@SQLskills.com

How do you improve performance?

- Code changes
  - Tune/optimize queries
  - Optimize transactions
  - Reduce compiles/recompiles
  - Return only the data that users need *(like, really need)*
  - Minimize the use of functions, cursors, and row-based operations
  - Change isolation level
  - Improve cache plan use
- Schema changes
  - Normalize or de-normalize
  - Define PKs, FKs, and constraints
  - Data types
- Index
- Update statistics (?!)
- Add more tempdb files
- Remove/archive historical data
- ~~Partition~~
- Separate reporting from OLTP
- Upgrade
  - Bug or feature
- More/new hardware
  - CPU*/memory/storage
- Use new features…

Overview

- Columnstore
- In-Memory OLTP
- Cardinality Estimator
- Query Store plan forcing / automatic plan correction
- Upgrade Testing

Columnstore

- **Introduced in SQL Server 2012 Enterprise Edition**
  - Limited initially, enhancements added with each subsequent release
- **Primarily designed for data warehouses**
  - Storing and querying large amounts of fact data
  - Support for operational analytics added in SQL Server 2016
- **Data is stored in a compressed, column format, rather than traditional row-based storage**
  - Uses the X-Velocity In-Memory Compression Engine, which is also used in PowerPivot and Analysis Services (Tabular Mode)
- **Supported with Availability Groups**
- **Available in Standard Edition starting in SQL Server 2016 SP1**
  - Columnstore Object Pool size capped at 32GB

Nonclustered Columnstore Indexes

- **Nonclustered Columnstore Index (NCCI)**
  - Updateable as of SQL Server 2016
  - Comprised of a sub-set of columns from the table
  - Provides real-time operational analytics for OLTP workloads
    - Remove time delays from ETL operations
    - Eliminates the need for a separate data warehouse and complexity of ETL
    - Eliminates multiple rowstore nonclustered indexes to support analytical queries
  - Supports offloading analytics to readable secondary replicas with Availability Groups
  - Note: maintaining a NCCI is more expensive than a B-tree index
    - There is no in-place update for NCCI, it is a delete and then an insert

Clustered Columnstore Indexes

- **Clustered Columnstore Index (CCI)**
  - Comprised of all columns in the table
  - Can be implemented on-disk or in-memory
  - Benefits analytical queries executed against a DW database (e.g. fact tables)
    - Tables ideally partitioned with at least one million rows/partition
    - Data loading typically by ETL and bulk operations
  - Can also provide benefit for analytical queries for tables with heavy inserts, where there are few updates and deletes (DW or OLTP)
  - Use with IOT data for compression (ratios as high as 25x compared to rowstore)

© SQLskills, All rights reserved.
https://www.SQLskills.com

Batch Mode

- Maximum performance gains are realized when operators can use batch mode for columnstore indexes
- Support for batch mode has been expanded to more operators with each release
- Initially, batch mode execution was only seen with multi-threaded (parallel) queries
  - Batch mode execution for serial queries added in SQL Server 2016 (compatibility mode 130)

SQLskills immerse yourself in sql server

7

Columnstore Enhancements by Version

2012	2014	2016	2017	2019
NCCI read-only, secondary NCI indexes can be created		Updatable NCCI (only one), supports filter definition	Create NCCI online Support non-persisted computed columns (NCCI only)	
	Updatable CCI, no other indexes allowed	Updateable CCI, secondary NCI indexes can be created		
		Columnstore index on an in-memory table (only one)		Tuple-mover helped by a background task

SQLskills immerse yourself in sql server

8

Determining Which Strategy is Best

- **Clustered Columnstore**
  - INSERT mostly workload
  - Star schema/traditional DW
  - Light OLTP < 10% UPDATE/DELETE with mostly analytic queries
- **Nonclustered Columnstore**
  - Normal OLTP workload
  - Heavy UPDATE/DELETE
  - Normalized table schema

Questions to Ask

- **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 will determine whether Columnstore is the right solution

What You're Looking For…

- **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 point queries
- **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 loading strategy determine effectiveness
- **Do I need to perform analytics on a transactional workload?**
  - Updatable NCCIs with filter criteria on "warm" data

Demo

Performance changes with columnstore

Data types not supported

- **ntext, text, and image**
- **nvarchar(max), varchar(max), and varbinary(max)**
  - Supported in SQL Server 2017 CCI
- **rowversion (and timestamp)**
- **sql_variant**
- **CLR types (hierarchyid and spatial types)**
- **xml**
- **uniqueidentifier**
  - Supported in SQL Server 2014 and higher

Index Limitations

- **Maximum of 1024 columns**
- **NCCI and CCI cannot have constraints (unique, PK, FK)**
  - With NCCI, base table/CI can have constraints
  - With CCI, NCI can have constraints
- **Cannot be created on a view or indexed view**
- **Cannot include a sparse column**
- **Must drop and recreate a columnstore index to change its definition (only supports ALTER INDEX for REBUILD)**
- **Cannot be created by using the INCLUDE keyword**

Features Not Supported

- **Computed columns**
  - Non-persisted computed column supported in SQL Server 2017 CCI
- **Page and row compression, and vardecimal storage format**
  - Columnstore data is already compressed
  - COLUMNSTORE_ARCHIVE (added in SQL Server 2016)
- **Replication**
- **FILESTREAM**

Overview

- **Columnstore**
- **In-Memory OLTP**
- **Cardinality Estimator**
- **Query Store plan forcing / automatic plan correction**
- **Upgrade Testing**

In-Memory OLTP (1)

- **Introduced in SQL Server 2014 Enterprise Edition**
  - Additional capabilities added with subsequent releases
  - Also available in SQL Server 2016 SP1 Standard Edition with a limit of 32GB of In-Memory objects *per database*
    - https://msdn.microsoft.com/library/cc645993.aspx
    - Note: if you also use columnstore, the max is 32GB for disk-based. If you use memory-optimized columnstore, it counts against the 32GB in-memory limit.
- **With the reduced cost of memory and CPU, I/O often remains a limiting factor in fast performance**
- **Typical bottlenecks with traditional, disk-based structures can exist around locking, latching, spinlocks, and writing to the transaction log which manifest as concurrency and latency issues**

In-Memory OLTP (2)

- **In addition to accessing disk-based structures, there can be a large number of computer instructions to execute a transaction which affect overall duration**
  - Increasing the number of CPUs doesn't linearly scale to address this
- **Natively compiled procedures reduce the number of computer instructions**
- **Microsoft proposed the original concept for an engine to support in-memory workloads in 2008 (codename Hekaton); planning and design started in 2010**

In-Memory OLTP (3)

- **An original goal was to execute OLTP transactions in microseconds (less than 1 millisecond)**
- **In-Memory OLTP provides optimistic concurrency and removes locking and latching, in addition to having data reside in memory**
  - Data structures provide efficient data access
  - With no locking or latching, solution can scale linearly
  - Log records only written on transaction commit, or at a set time if using delayed durability (SQL Server 2014 and higher)
- **Per Microsoft, customers can get up to 30x performance improvement**
  - YMMV…it depends on workload and access patterns; up to 10x improvement more realistic

In-Memory Objects

- **Memory-optimized tables**
- **Memory-optimized table types**
- **Memory-optimized indexes**
- **Memory-optimized filegroup**
- **Natively compiled T-SQL modules**
- **Memory-optimized tempdb metadata (SQL Server 2019)**

Tables

- **Memory-optimized tables store user data**
  - Tables are durable by default
    - Data will persist across a restart
  - Can be configured as delayed-durable or non-durable
    - Use non-durable for transient data that can be re-populated if needed
  - Not all data types supported

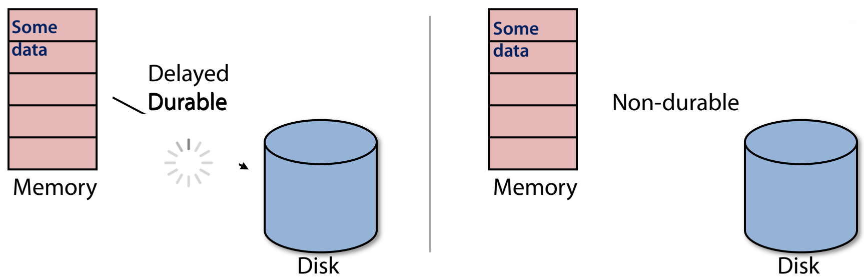


Table Types

- **Memory-optimized table types**
  - Use for temp tables, TVPs, and table variables to hold transient data
  - Only stored in memory using same structure as tables, nothing in tempdb
  - Must have one index

Table Types Example

```
CREATE @OrderInfo TABLE (
    [RowNum] INT IDENTITY (1,1),
    [OrderID] INT,
    [CustomerID] INT,
    [CustomerPONum] NVARCHAR(40)
    );
```

"Traditional" method – create a table variable (still backed by tempdb)

```
CREATE TYPE dbo.OrderInfo
   AS TABLE (
        [RowNum] INT IDENTITY (1,1) NOT NULL,
        [OrderID] INT PRIMARY KEY NONCLUSTERED,
        [CustomerID] INT,
        [CustomerPONum] NVARCHAR(40)
        )
   WITH
        (MEMORY_OPTIMIZED = ON);

DECLARE @OrderInfo dbo.OrderInfo;
```

New option – create a table type first, as an in-memory structure, then reference it in code

SQLskills immerse yourself in sql server

23

Indexes (1)

- **Every memory-optimized table must have at least one memory-optimized index**
  - Maximum of eight (8) indexes through in SQL 2014 and SQL 2016
  - No limit in SQL 2017+ and Azure SQL Database
- **Nonclustered vs. Hash**
  - Nonclustered ideal for range scans, inequalities, and when sort order is needed
  - Hash is optimal for equality predicates on all key columns
  - Hash requires estimating the number of distinct values for the index key
- **Columnstore**
  - Includes all columns (clustered)

SQLskills immerse yourself in sql server

24

Indexes (2)

- **Nonclustered and hash indexes are not represented on disk; index changes are not written to the log**
- **These indexes are rehydrated (data streamed from disk to memory) when:**
  - A database is restored
  - The instance restarts or the server reboots
  - Change a database from READ_WRITE to READ_ONLY (or vice versa)
  - Change the READ_COMMITTED_SNAPSHOT setting
  - A database is taken OFFLINE, then brought ONLINE
- **With an AG failover, as REDO occurs at the secondary, in-memory objects are updated, providing an advantage in the event of a failover**
- **Do not fragment like disk-based indexes**
- **Columnstore in-memory indexes are persisted**

The Filegroup

- **In order to use In-Memory OLTP, you must create a separate filegroup**
  - Defined as MEMORY_OPTIMIZED_DATA
  - Only one filegroup of this type allowed
  - You create one or more containers for the filegroup
  - Recommended to have enough space to support 4x the size of each memory-optimized table that is *durable*
- **The filegroup contains checkpoint files (data and delta) to track changes to durable objects**
  - Used to recreate durable (and delayed-durable) tables and indexes after a restart
- **Make sure Instant File Initialization is enabled**

Natively compiled T-SQL modules

- **Natively compiled T-SQL modules (stored procedures, triggers, scalar UDFs)**
  - Optimized and compiled into machine language
  - Removes compilation time and CPU
  - Parameter sniffing is not used, compiled using UNKNOWN values
    - Can use OPTIMIZE FOR to try and force a specific plan
    - Interpreted SPs do use parameter sniffing
  - Statistics automatically updated in SQL Server 2016 with compatibility mode 130
  - Updates to statistics *do not* initiate re-compilation

27

Memory-Optimized tempdb Metadata

- **System tables for tempdb can be memory-optimized**
  - These tables track the temporary tables that are created in tempdb
  - For high-volume systems, contention on these tables (metadata) can occur
    - Even with temp table caching (introduced in SQL 2005)
  - Enabling this option removes contention on these system tables to improve scalability
- **Requires an instance restart**
- **Implement if you see PAGELATCH contention on system objects such as sysobjvalues and sysseobjvalues**
  - This will *not* address contention for PFS and SGAM pages
- **Limitations:**
  - Columnstore indexes cannot be created on temporary tables when memory-optimized tempdb metadata is enabled

28

In-Memory OLTP Solutions (1)

- *In-Memory OLTP is not a solution for all performance problems*
- **It can address problems related to query execution and data access**
  - It will not benefit code related to client connectivity or transaction logging
    - Exception: if implemented as non-durable
- **Ideal workload pattern addressed is a large volume of small transactions**
- **Typical uses:**
  - Increase transaction throughput
  - Increase the rate of data ingestion
  - Decrease latency because application/business is time-sensitive
  - Transient data

In-Memory OLTP Solutions (2)

- **Where it works really well:**
  - Inserts/updates/deletes
  - Data that is heavily read (high concurrency) that is read-only or modified infrequently via SPs
  - Replacing #temp tables, table variables, TVPs
  - Staging data during ETL processes
  - Initial data load (then move data to disk-based, columnstore)
  - Session state database (e.g. for ASP.NET)
  - Caching

© SQLskills, All rights reserved.
https://www.SQLskills.com

In-Memory OLTP Solutions (3)

- **Where it doesn't work well:**
  - Resource limitations
    - Not enough memory to support In-Memory tables
    - Slow I/O for the transaction log
  - Queries that return a lot of data or perform aggregations
  - Query plans with large range scans/table scans
  - Query plans with parallelism
    - Note: if ETL writes are parallel with disk-based tables, they won't be with in-memory (http://www.nikoport.com/2018/01/20/parallelism-in-hekaton-in-memory-oltp/)
- **If your original latency is due to factors outside SQL Server, In-Memory OLTP may not provide any benefit (e.g., "chatty" application)**
  - Understand source of the existing problem before you go down this path

Demo

Testing performance changes with In-Memory OLTP

Requirements

- **Separate filegroup in the database**
  - Cannot be removed (must drop the database to "remove" it)
  - Cannot create database snapshot for databases with this filegroup
- **Enough memory to hold the In-Memory tables and indexes**
  - Table will be the approximate size of the disk-based table, indexes are typically smaller
- ***Additional* memory to support the workload, including row-versioning**
- **Disk space to support the size of durable memory-optimized objects**
  - https://blogs.msdn.microsoft.com/sql_server_team/choosing-the-right-server-memory-for-restore-and-recovery-of-memory-optimized-databases/
- **Note: Natively compiled T-SQL modules are *optional*, but highly recommended to maximize performance gains**

Limitations for Tables

- **Not all features are supported, for example:**
  - Compression
  - Partitioning
  - Replication*
  - Linked Servers
  - DDL triggers
  - Most cross-database transactions
- **Not all data types supported (e.g., datetimeoffset, geography, xml)**
  - Computed columns are supported in SQL Server 2017
- **IDENTITY must seed at 1 and increment by 1, cannot reseed**
- **TRUNCATE TABLE is not supported**
- **Migrating an existing table to in-memory is not an online process (ALTER TABLE not supported for this operation)**
- **DBCC CHECKDB cannot validate in-memory tables**

Items of Note for Natively-compiled SPs

- Can only access memory optimized tables and table types
- No parallel processing
- Compiled using UNKNOWN values (can use OPTIMIZE FOR hint)
- Query plans use nested loop joins
  - Only stream aggregation is available for aggregates
- Execution statistics not collected by default due to perf impact
  - Must enable via sys.sp_xtp_control_query_exec_stats or sys.sp_xtp_control_proc_exec_stats

Limitations for Natively-compiled SPs

- Cannot create or access tables in tempdb
  - Use memory-optimized tables or table types/table variables
- EXISTS cannot be used with IF and WHILE
- MERGE is not supported
- UPDATE statements that use the FROM clause are not supported
- DELETE…JOIN syntax not supported
- Cursors are not supported

- SELECT DISTINCT is supported in SQL Server 2017
- CASE expressions are supported in SQL Server 2017
- APPLY operator is supported in SQL Server 2017
- JSON functions supported in SQL Server 2017

Steps to Determine if In-Memory OLTP is Viable

- **Transaction Performance Analysis Report**
  - Analyzes existing workload to determine where In-Memory OLTP may help improve performance
  - Table and SP execution statistics are captured
  - Identifies incompatibilities
  - Check out Ned Otter's post for an alternate method:
    - http://nedotter.com/archive/2017/06/migrating-tables-to-in-memory-oltp/
- **Memory Optimization Advisor**
  - Validates if table can be migrated to use In-Memory OLTP
    - Will not make any modifications if there are limiting factors
  - Can be used to migrate tables or generate script
- **Native Compilation Advisor**
  - Validates if a stored procedure can be migrated
  - Procedure code cannot be migrated via UI

Measuring Performance Change

- **You need a baseline**
  - Capture with Query Store, manually, or with a third-party tool
- **What metrics do you care about?**
  - This is what you need to capture
- **Nothing else can change**
  - Not data, not maintenance, not indexes, not one *other* thing

Testing In-Memory OLTP

- **It is recommended to perform testing in a non-production environment**
- **Typical testing challenges exist**
  - How to generate a comparable workload and/or "busiest" scenario?
  - Is it possible to test all related code?
- **Isolate changes to one table, or a small set of related tables, for testing**
- **Implementation (and thus, roll-back) requires an outage; testing is critical**
- **Basic steps:**
  - Capture performance metrics in production environment for an existing disk-based table and/or related stored procedures
  - Restore to a testing environment and create the appropriate In-Memory objects (i.e., filegroup, table and indexes, stored procedure(s))
  - Simulate production workload and capture the same performance metrics

Overview

- **Columnstore**
- **In-Memory OLTP**
- **Cardinality Estimator**
- **Query Store plan forcing / automatic plan correction**
- **Upgrade Testing**

New Cardinality Estimator

- **The Query Optimizer evaluates the cost of one or more plans when deciding which plan to ultimately execute**
- **One factor used to determine cost is the number of estimated rows that will need to be processed for each operator**
  - This is the cardinality estimate
- **The cardinality estimator (CE) component was significantly changed in SQL Server 2014**
  - First redesign since SQL Server 7.0

41

Cardinality Estimate issues

- **Major red flag to watch for, not just when upgrading to 2014+**
  - Skewed estimate vs. actual
- **Magnification and distortion as we move through the plan tree**
- **Other symptoms:**
  - Query performs badly or doesn't execute at all due to memory error
  - Performance may be good sometimes and bad other times

42

Cardinality Estimate First Steps

- **Key areas to validate**
  - CE version
  - Query
  - Execution plan
  - Statistics
- **Areas to investigate further:**
  - Missing indexes / missing or stale statistics
  - Table variables
  - TVF

Cardinality Estimator Version in SQL Server 2014

- **The new CE will be used in SQL Server 2014 if the database has the compatibility level set to 120**
- **If database compatibility level is less than 120, the new CE can be used on a per-query basis by using the QUERYTRACEON and trace flag 2312**
  - QUERYTRACEON requires sysadmin permissions
  - Can be used with Plan Guides
  - Takes precedence over server and session trace flags
- **For databases using compatibility level 120, use QUERYTRACEON and trace flag 9481 to *revert* to the legacy cardinality estimator**
- **Databases that are upgraded to, attached to, or restored to a SQL Server 2014 retain their compatibility level and therefore will use the legacy CE by default**

Cardinality Estimator Version in SQL Server 2016+

- CE version is determined by the LEGACY_CARDINALITY_ESTIMATION database scoped setting*
  - Database compatibility level is relevant for new CE
- If LEGACY_CARDINALITY_ESTIMATION = ON, then the old CE is used, regardless of database compatibility level
  - Equivalent to using trace flag 9481
- If LEGACY_CARDINALITY_ESTIMATION = OFF, then CE version is determined by database compatibility level
- Trace flags 9481 and 2312 can still be used to change CE for individual queries (with QUERYTRACEON hint)
- CE version for tempdb is relevant if you use temporary tables

Verifying Cardinality Estimator Version Used

- CardinalityEstimationModelVersion attribute lists what CE was used
- Found in the XML or in the Properties of the plan
  - 70 = Legacy
  - 120,130, 140, 150 = New

Demo

Testing CE changes with Query Store

Upgrade Options

- **Test before you upgrade to SQL Server 2014 or higher**
  - Identify problematic queries and address them prior to upgrading
- **Upgrade to SQL Server 2014 or higher without testing**
  - Keep using the old CE
- **Upgrade to SQL Server 2014 or higher without testing**
  - Use the new CE
  - Prepare to fight fires in production
- ***Upgrading without testing creates a significant risk for your business***
- **The Importance of Database Compatibility Level in SQL Server**
  - https://www.sqlskills.com/blogs/glenn/database-compatibility-level-in-sql-server/
- **Avoiding SQL Server Upgrade Performance Issues**
  - https://www.sqlskills.com/blogs/glenn/avoid-sql-server-upgrade-performance-issues/

48

Overview

- Columnstore
- In-Memory OLTP
- Cardinality Estimator
- Query Store plan forcing / automatic plan correction
- Upgrade Testing

How Do You Fix a Poorly-Performing Query?

Change code and/or schema	Add RECOMPILE	Manually get the "best" plan in cache
	Use a plan guide	Force a plan in Query Store

Forcing Plans with Query Store

- **Query Store allows you to easily find queries with multiple plans and force one plan**
  - Can be done in the UI
  - Can be done with T-SQL
- **If a plan is no longer optimal, Query Store can continue to use it unless you remove it**
- **Monitor failures with Extended Events**
  - query_store_plan_forcing_failed
  - Can also check sys.query_store_plan
- **Adding hints changes the query text which creates a new query (and query_id) in Query Store**

51

Demo

Forcing plans

Points to Remember

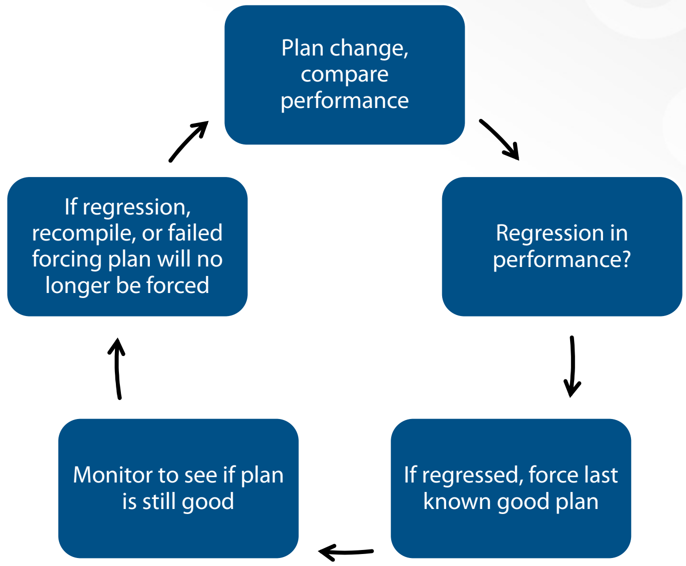
- It may not always be obvious that a plan is forced – check the plan and Query Store to see if it is
- Query performance can be different across environments for multiple reasons – including forced plans!
- If object_id changes, a forced plan will no longer be tied to the object
- If an index name changes, a forced plan cannot be used
- Pay attention to forced plans when testing code and schema changes

53

Automatic Plan Correction

- Available in SQL Server 2017+ EE and Azure SQL Databases
  - Enabled per database
  - Uses Query Store
- Tool to quickly mitigate query performance issues based on regressions
  - Based on CPU change
    - Thresholds are not documented, as they may change

54



Automatic Plan Correction

- **Reasons a plan will be un-forced**
  - Regression
  - Recompile due to statistics or schema change
  - Failed forcing
- **Can use the information captured to make corrections manually**
  - Stored in sys.dm_db_tuning_recommendations
    - Does not persist, snapshot to a table or use XE if you want to retain information
  - This DMV is not populated in Enterprise Edition
- **Plan forcing is typically not a recommended long-term solution, best practice is to address reported plan regressions through code/schema changes**

Demo

Automatic Plan Correction

Can I Trust It?

- It is not perfect, but it has been developed with telemetry from Azure SQL Database implementations
- Catches severe regressions
- Its ability to recovery from any “bad decision” is highly reliable as there is continual validation of forced plans and automatic back-off logic built-in

Monitoring with Extended Events

- **Create an Extended Events session that captures automatic tuning events, writes to an event_file target, and starts when the instance starts (always running)**
  - automatic_tuning_error
  - automatic_tuning_plan_regression_detection_check_completed
  - automatic_tuning_plan_regression_verification_check_completed
  - automatic_tuning_recommendation_expired

Overview

- **Columnstore**
- **In-Memory OLTP**
- **Cardinality Estimator**
- **Query Store plan forcing / automatic plan correction**
- **Upgrade Testing**

Distributed Replay Utility (DRU)

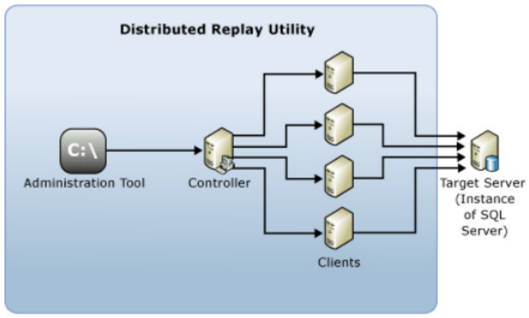
- **Introduced in SQL Server 2012**
- **DRU is an upgrade tool**
  - Primary use is helping customers upgrade to the latest version of SQL Server
- **Can also be used to examine the impact of hardware, software, and application changes**
- **Provides the capability to capture a trace and then replay from multiple clients (workstations)**
  - More scalable than Profiler replay as Profiler is limited to replay from a single client

DRU Topology

- **Distributed Replay controller**
  - Only one controller is permitted
  - Runs as a Windows service (SQL Server Distributed Replay Controller)
  - Orchestrates actions of clients
- **Distributed Replay clients**
  - One or more clients (up to 16) can be used, and together they simulate a typical workload
  - Each runs as a Windows service (SQL Server Distributed Replay Client)
  - Use of more than one client requires Enterprise Edition
    - Developer Edition only allows one client
- **Distributed Replay administration tool**
  - DReplay.exe is used to talk to controller

DRU Topology

- **Target server**
  - Hosts a SQL Server instance against which trace data is replayed by clients
  - Data about replay performance should be captured against this server



https://technet.microsoft.com/en-us/library/ff878183(v=sql.130).aspx

Order of Events for Replay

- **Start a COPY_ONLY full backup**
- **Start replay trace to capture events**
  - This will continue to run after the backup completes; how long is determined by the workload you're trying to capture
- **Stop trace and filter out events from prior to backup completion**
  - Aligning the backup and trace reduces the likelihood of problems related to constraint violations
- **Restore database to another instance (Test/QA/Dev)**
  - Provide db_owner to DR Client and Controller accounts
- **Preprocess trace file(s) using DReplay**
- **Replay trace file(s) using one or more clients**
  - Capture performance data on the instance where the database is restored (e.g. trace, PerfMon)

DRU Configuration Files

- **Controller File**
  - DReplayController
    - Specify logging level
- **Client Configuration File**
  - DReplayClient
    - Specify controller, working and result directories, logging level
- **Preprocess Configuration File**
  - DReplay.Exe.Preprocess
    - Specify whether to include system session activity
    - Specify whether to reduce idle time
- **Replay Configuration File**
  - DReplay.Exe.Replay

Replay Settings

- **DRU provides the option of replaying the trace in two modes:**
  - Synchronization mode
  - Stress mode
- **In synchronization mode, the replay occurs in the order of the original events, and is synchronized across all the clients**
- **Stress mode, which is the default, can be used to drive the workload, and there is no synchronization across clients**
  - Can decrease "think time" and "connect time" options to dial back the workload
    - Default value for both ThinkTimeScale and ConnectTimeScale is 100, which is a percentage
- **Can also change whether connection pooling is used, and number of threads per replay client (default is 255, max is 512)**

Data Collection During Replay

- **Previously a manual effort**
  - Could use ReadTrace to compare captured trace files
- **Database Experimentation Assistant released in Fall 2016**
  - Current release is version 2.6 (March 2020)
  - Provides a UI to capture and replay a trace/XE
    - Trace is replayed against original (or comparable) server and new server
  - Also provides workload analysis reports
    - Compares performance between the executions
- **Source versions are SQL Server 2005 and higher**
- **Target versions are SQL Server 2012 and higher**

Query Tuning Assistant

- **Available in 18.x version of SSMS**
- **Created to help with testing changes in compatibility level**
- **Uses Query Store to capture workload performance metrics and then compares and analyzes the data**
  - Tests regressed queries with different hints, including FORCE_LEGACY_CARDINALITY_ESTIMATION
- **Requires db_owner permission**

Order of Events for QTA

- Restore a backup of the database
- Initiate QTA from the database menu
  - Configure how long to collect data (minimum is 1 day) and Query Store settings
- Start the workload and let run for the testing duration
  - This captures a baseline
- When the collection time completes, upgrade the database compatibility level
- Run the workload again
  - You can monitor regressed queries during this time
- When the workload has finished, queries that regress are identified and can then be selected for experimentation
- After experimentation, queries that can optimized are listed with the option to implement a plan guide to stabilize performance

69

New Features = Immediate Win?

- Columnstore
- In-Memory OLTP
- Cardinality Estimator
- Query Store plan forcing / automatic plan correction

70

Key Takeaways

- **New features can provide a method to improve and/or stability query performance**
- **Columnstore and In-Memory OLTP can provide a performance boost for the right workload, testing is essential**
- **The new Cardinality Estimator frequently improves query performance, but regressions are definitely possible**
  - Testing prior to upgrading is critical
- **Beyond capturing query performance data, Query Store can be used to force plans (temporary solution) manually and automatically via Automatic Plan Correction**

Additional Resources

- **Pluralsight**
  - SQL Server: Automatic Tuning in SQL Server 2017 and Azure SQL Database
    - https://bit.ly/2JUmONZ
- **Blog posts**
  - http://www.nikoport.com/columnstore/
  - https://www.sqlskills.com/blogs/jonathan/installing-and-configuring-sql-server-2012-distributed-replay/
  - https://www.sqlskills.com/blogs/jonathan/performing-a-distributed-replay-with-multiple-clients-using-sql-server-2012-distributed-replay/

Additional Resources

- Microsoft Docs
  - https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/migrating-to-in-memory-oltp
  - https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/estimate-memory-requirements-for-memory-optimized-tables
- Whitepapers
  - SQL Server In-Memory OLTP and Columnstore Feature Comparison
    - https://download.microsoft.com/download/D/0/0/D0075580-6D72-403D-8B4D-C3BD88D58CE4/SQL_Server_2016_In_Memory_OLTP_and_Columnstore_Comparison_White_Paper.pdf
  - SQL Server In-Memory OLTP Internals for SQL Server 2016
    - https://docs.microsoft.com/en-us/sql/whitepapers/sql-server-in-memory-oltp-internals-for-sql-server-2016
  - In-Memory OLTP – Common Workload Patterns and Migration Considerations (2014)
    - https://msdn.microsoft.com/library/dn673538.aspx

Review

- Columnstore
- In-Memory OLTP
- Cardinality Estimator
- Query Store plan forcing / automatic plan correction
- Distributed Replay


Questions?
SQLskills
immerse yourself in sql server