



DP-300

Administering Microsoft Azure SQL Solutions



Optimize Query Performance in Azure SQL

Introduction to Query Store, Execution Plans, Indexes, and Query Tuning

Objectives

- Analyze query plans and identify problem areas
- Evaluate potential query improvements
- Review table and index design
- Determine whether query or design changes have had a positive effect

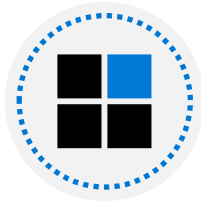
Explore Query Performance Optimization



Objectives



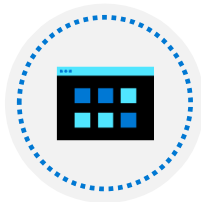
Generate and save execution plans



Compare the different types of execution plans

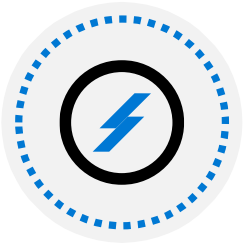


Understand how and why query plans are generated



Understand the purpose and benefits of the Query Store

What are execution plans?



Every time you execute a query, the query optimizer uses a path of operations to process your query and retrieve any results.

The optimizer also collects statistics about the operation and execution of this plan.

How are query plans generated?

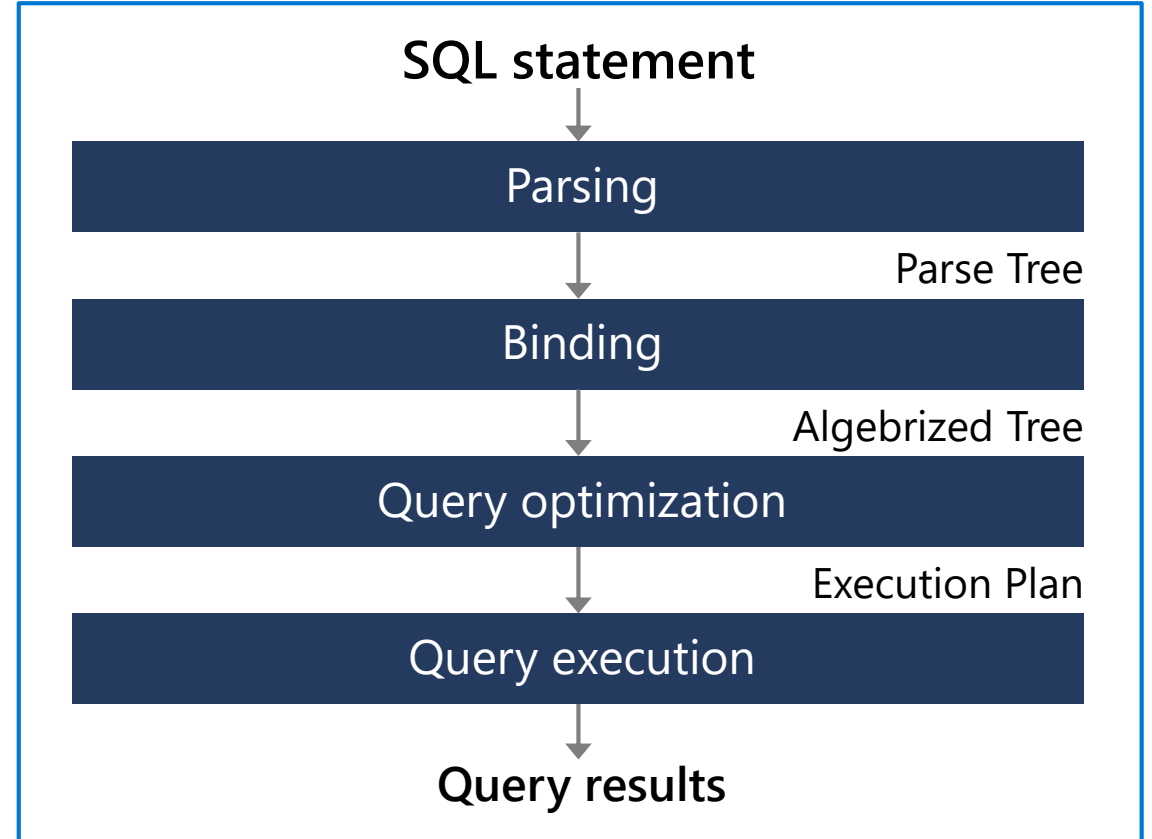
The first time you run a given query, the optimizer generates several plans and chooses the one with the lowest overall cost.

The Query Optimizer

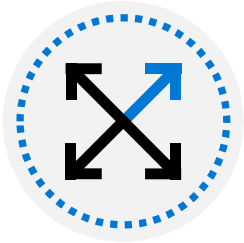
The Query Optimizer takes a query and returns a query plan

Will attempt several plans to choose the 'best guess'

Generating a plan has a high CPU cost, so SQL Server® caches plans

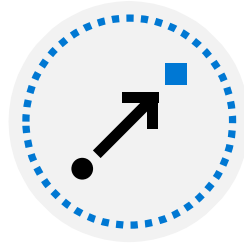


Types of execution plans



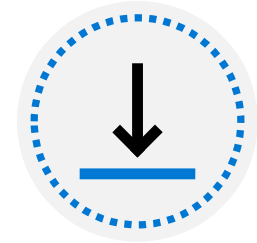
Estimated execution plans

The path the database engine plans to take to retrieve the data to answer your query



Actual execution plans

The estimated execution plan, but also including actual statistics from the execution of the query



Live query statistics

An animated view of query execution showing progress and order of operations

More about actual vs. estimated execution plans

Actual execution
plans

VS.

Estimated
execution plans

The plans shape will nearly always be the same

The actual execution plan will have additional information about row counts and runtime stats

The actual execution traditionally has **significant overhead** to capture

Capturing plans in Management Studio

SQL Server Management Studio allows you to capture both estimated and actual execution plans, and Live Query Statistics

Plan capture can also be enabled by keyboard commands

You can capture a text query plan by using the SET SHOWPLAN ON option

The screenshot displays the SQL Server Management Studio interface. The top toolbar includes buttons for 'Estimated Execution Plan' (a document icon with a magnifying glass), 'Actual Execution Plan' (a document icon with a play button), and 'Live Query Statistics' (a document icon with a refresh symbol). The main query window shows a SQL query: `DECLARE @SalesPersonID INT
SELECT @SalesPersonID
SELECT SalesOrderID, OrderDate
from Sales.SalesOrderHeader
WHERE SalesPersonID = @SalesPersonID
OPTION (OPTIMIZE FOR (@SalesPersonID = 277))`. The 'Execution plan' tab is active, showing a graphical plan for 'Query 1: Query cost (relative to the batch): 100%'. The plan consists of a 'SELECT' operator (Cost: 0%) connected to a 'Nested Loops (Inner Join)' operator (Cost: 0%, 0.001s, 473 of 130 (363%)). The 'Nested Loops' operator is connected to an 'Index Seek (NonClustered)' operator on '[SalesOrderHeader].[IX_SalesOrderHeader_...]' (Cost: 1%, 0.000s, 473 of 130 (363%)) and a 'Key Lookup (Clustered)' operator on '[SalesOrderHeader].[PK_SalesOrderHeader_...]' (Cost: 99%, 0.000s, 473 of 130 (363%)).

Lightweight query profiling



Supported by Azure SQL Database



Required trace flag 7412 to be enabled globally, prior to SQL Server 2019



Newer releases allow it to be enabled at the query level

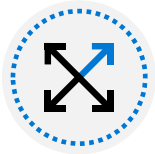


Provides additional statistics to help close gap to actual execution plan data

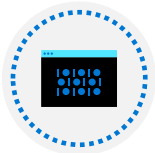
How to read an execution plan



Plans should be read from right to left and top to bottom



The **size** of the arrows connecting the objects is representative of the amount of data flowing to the next operator



Each operator contains a lot of metadata in its properties and tool tip. This is where you can learn how the decisions were made



All operators are assigned a cost



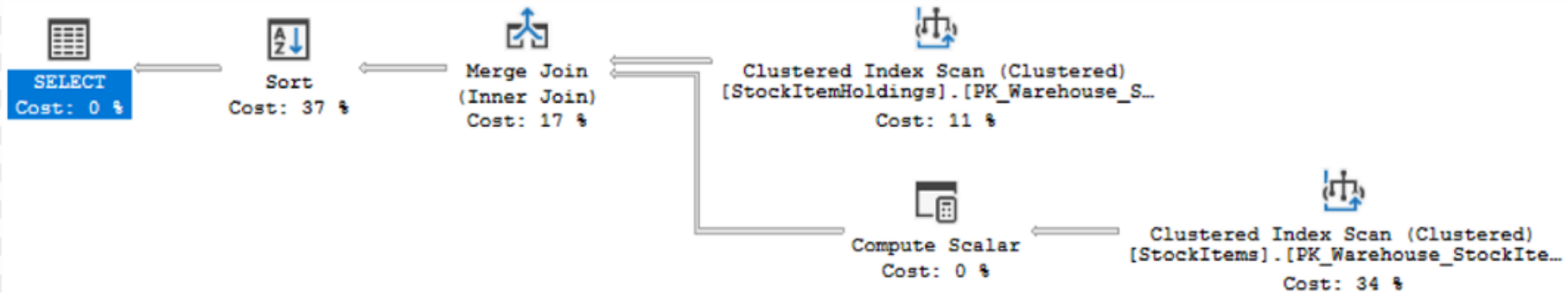
Even operators that have a 0% cost have some degree of cost—nothing is free in SQL Server query execution

Example: SQL query with a JOIN clause

```
SELECT  [stockItemName],  
        [UnitPrice] * [QuantityPerOuter] AS CostPerOuterBox,  
        [QuantityonHand]  
FROM [Warehouse].[StockItems] s  
      JOIN [Warehouse].[StockItemHoldings] sh ON s.StockItemID = sh.StockItemID  
ORDER BY CostPerOuterBox;
```

Query 1: Query cost (relative to the batch): 100%

SELECT [StockItemName] ,[UnitPrice] * [QuantityPerOuter] AS CostperOuterBox ,sh.QuantityOnHand FROM [Warehou



Tool tips and properties


Clustered Index Scan (Clustered)
[StockItems].[PK_Warehouse_StockItems]
Cost: 34 %

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0131613 (34%)
Estimated I/O Cost	0.0127546
Estimated Subtree Cost	0.0131613
Estimated CPU Cost	0.0004067
Estimated Number of Executions	1
Estimated Number of Rows	227
Estimated Number of Rows to be Read	227
Estimated Row Size	128 B
Ordered	True
Node ID	4
Object	
[WideWorldImporters].[Warehouse].[StockItems].	
[PK_Warehouse_StockItems] [s]	
Output List	
[WideWorldImporters].[Warehouse].[StockItems].StockItemID,	
[WideWorldImporters].[Warehouse].[StockItems].StockItemName,	
[WideWorldImporters].[Warehouse].	
[StockItems].QuantityPerOuter, [WideWorldImporters].	
[Warehouse].[StockItems].UnitPrice	

Properties	
Clustered Index Scan (Clustered)	
Misc	
Defined Values	[WideWorldImporters].[Warehouse].[StockItems].Stk...
Description	Scanning a clustered index, entirely or only a range.
Estimated CPU Cost	0.0004067
Estimated Execution Mode	Row
Estimated I/O Cost	0.0127546
Estimated Number of Executions	1
Estimated Number of Rows	227
Estimated Number of Rows to be Read	227
Estimated Operator Cost	0.0131613 (34%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	128 B
Estimated Subtree Cost	0.0131613
Force Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Clustered Index Scan
Node ID	4
NoExpandHint	False
Object	[WideWorldImporters].[Warehouse].[StockItems].[PK_1
Alias	[s]
Database	[WideWorldImporters]
Index	[PK_Warehouse_StockItems]
Index Kind	Clustered
Schema	[Warehouse]
Storage	RowStore
Table	[StockItems]
Ordered	True
Output List	[WideWorldImporters].[Warehouse].[StockItems].Stck...
[1]	[WideWorldImporters].[Warehouse].[StockItems].Stck...
[2]	[WideWorldImporters].[Warehouse].[StockItems].Stck...
[3]	[WideWorldImporters].[Warehouse].[StockItems].Quar...
[4]	[WideWorldImporters].[Warehouse].[StockItems].UnitP...
Parallel	False
Physical Operation	Clustered Index Scan
Scan Direction	FORWARD
Storage	RowStore
TableCardinality	227

Common plan operators: Finding data

Index Seek

Reads the portion of the index which contains the needed data

Index Scan

Reads the entire index for the needed data

Table Scan

Reads the entire table for the needed data

Key Lookup

Looks up values row by row for values which are missing from the index used

Table Valued Function

Executes a table valued function within the database



Index Seek (NonClustered)
[AdventureWorks].[IX_Website_LogonId_C
Cost: 8 %



Clustered Index Scan
[AdventureWorks].[PK_#Logons__ACC1
Cost: 0 %



Table Scan
[AdventureWorks].[FilteredAccountComp
Cost: 25 %



Key Lookup (Clustered)
[AdventureWorks].[License].[PK_License]
Cost: 19 %



Table Valued Function
[AdventureWorks].[FilteredAccountComp
Cost: 0 %

Common plan operators: Filtering & sorting

Nested Loops

Performs inner, outer, semi and anti semi joins. Performs search on the inner table for each row of the outer table

Hash Match

Creates a hash for required columns for each row. Then creates a hash for second table and finds matches

TOP

Returns the specified top number of rows

Sort

Sorts the incoming rows

Stream Aggregate

Groups rows by one or more columns and calculates one or more aggregate expressions



Nested Loops :
(Inner Join)
Cost: 0 %



Hash Match
(Inner Join)
Cost: 3 %



Top
Cost: 0 %



Sort
(Distinct Sort)
Cost: 1 %

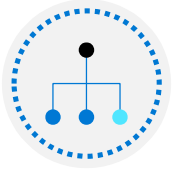


Stream Aggregate
(Aggregate)
Cost: 5 %

Demo: Reading an execution plan

Generate a query execution plan
Reading a query execution plan

Dynamic management objects



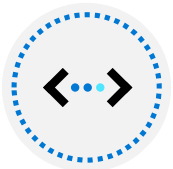
Dynamic Management Views and Functions supply data about database performance and state



These objects may be scoped at the database or server level



Azure SQL Database has some of its own DMVs to provide Azure specific data at the database level



All objects are in the sys schema and following the naming convention *sys.dm_**

Query Store

Introduced in SQL Server 2016, this data collection tool tracks query execution plans and runtime statistics



Can help you quickly identify queries that have regressed in performance



Allows you to easily identify the most expensive queries in your database



Available in Azure SQL Database and Azure SQL Managed Instances



A version of the Query Store is available in Azure Database for MySQL and Azure Database for PostgreSQL

Query Store cont'd

Common scenarios

- Detecting regressed queries
- Determining the number of times a query was executed in a given time window
- Comparing the average execution time of a query across time windows to see large deltas

Enabled by default for new Azure SQL Database databases, not enabled for

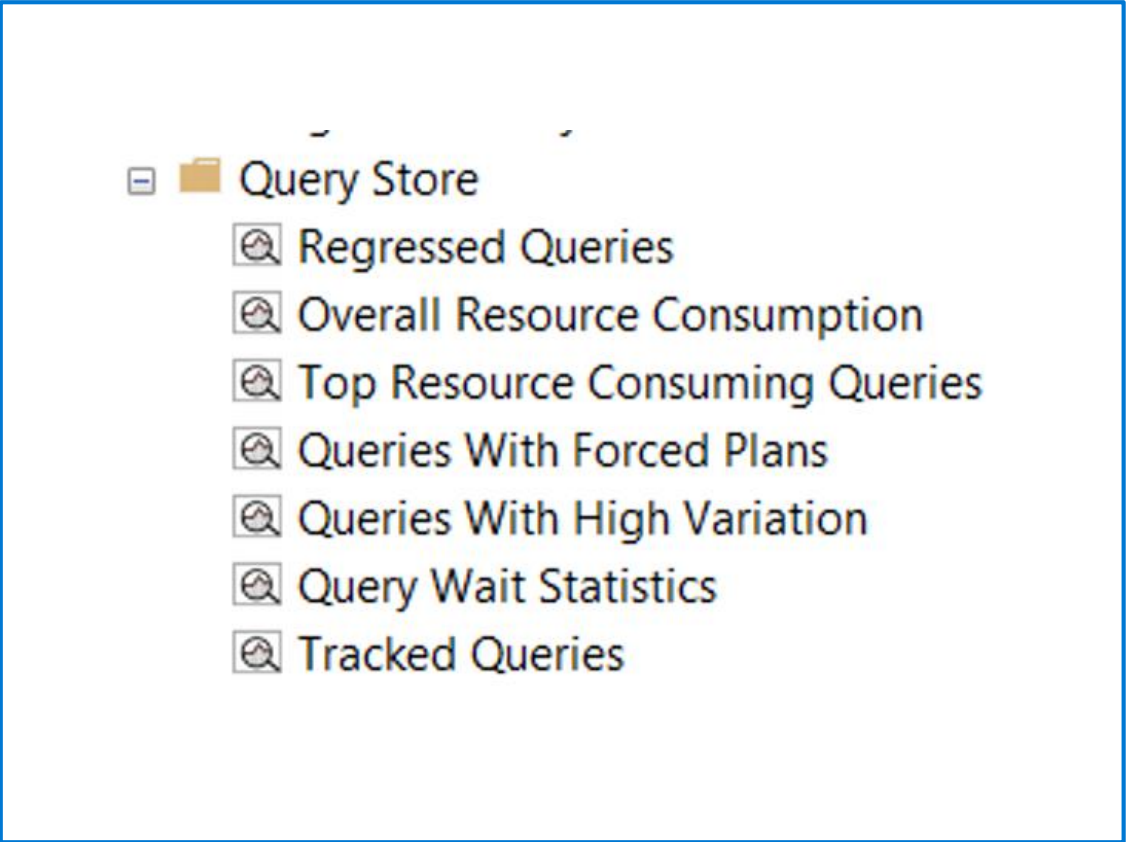
```
ALTER DATABASE <dbname> SET QUERY_STORE = ON (OPERATION_MODE = READ_WRITE);
```

Query Store reports

The Query Store includes several reports built into SQL Server Management Studio

These reports can be filtered by a range of time, allowing you to identify performance issues that happened in the past

These reports can also help you identify a problematic change in execution plans leading to a performance regression

- 
- [-] Query Store
 - 🔍 Regressed Queries
 - 🔍 Overall Resource Consumption
 - 🔍 Top Resource Consuming Queries
 - 🔍 Queries With Forced Plans
 - 🔍 Queries With High Variation
 - 🔍 Query Wait Statistics
 - 🔍 Tracked Queries

Performance overhead of Query Store



Any monitoring system has some degree of observer overhead to capture data



The query store has been developed with performance in mind and has relatively low overhead



The query store allows adjusting of collection parameters to reduce data collection on particularly busy systems

Plan forcing in the Query Store

- ▶ This feature allows you to force the database engine to use a specific execution plan for a given query
- ▶ Plan forcing is typically used as a temporary fix for a sudden query plan regression
- ▶ The full fix is to fix the query and/or indexes to ensure plan stability
- ▶ The Automatic Tuning feature in Azure SQL uses this feature

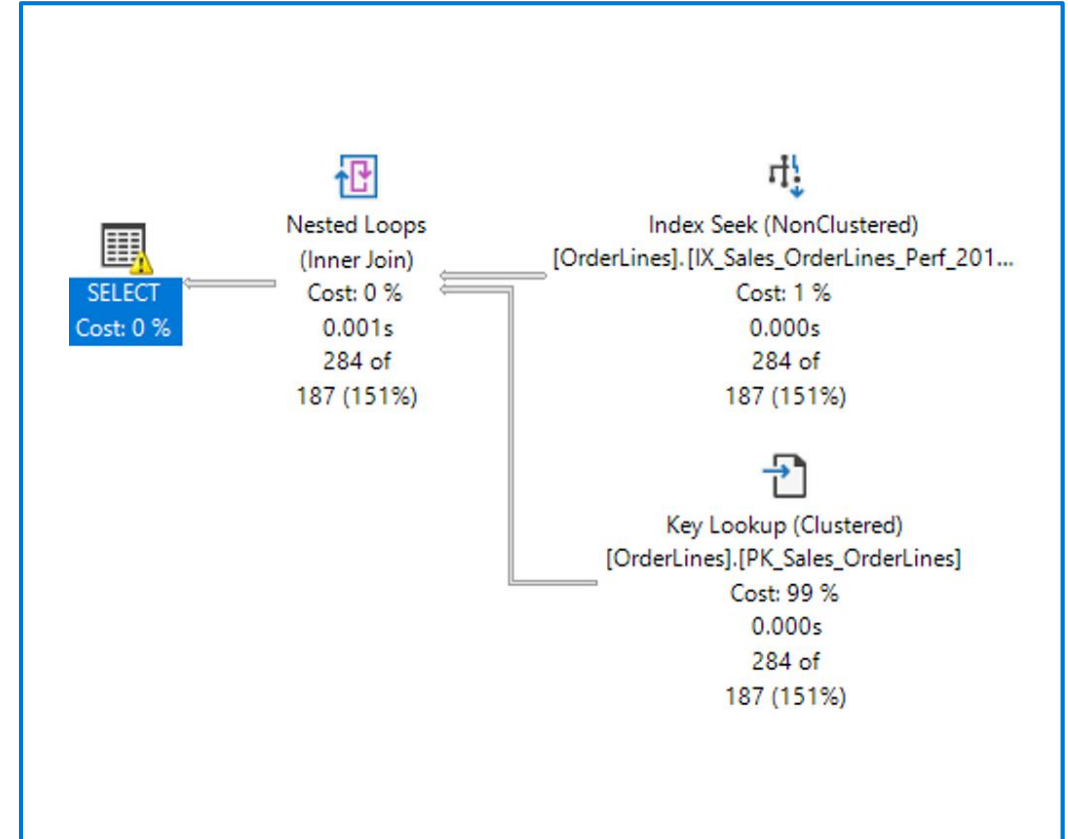
Demo: Query Store

**Showcase Query Store Reports in SQL
Server Management Studio**

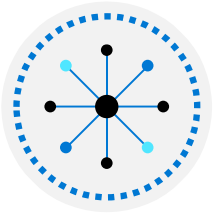
Identifying problematic execution plans

This query has a few problems:

- There is a warning that can be identified by the yellow triangle on the SELECT operation
- There is also an expensive Key Lookup that is using 99.6% of the query's execution cost



SARGability



This is a term that refers to a Search Arguable query predicate that can use an index to improve the performance of a query

Predicates that are not SARGs include some of the following examples:

```
WHERE LastName LIKE '%SMITH%'
```

```
WHERE CONVERT(CHAR(10), CreateDate, 121) = '2020-03-22'
```

- Functions can influence the SARGability of predicates, so you should be careful with their use

Missing indexes



SQL Server includes missing index warnings in execution plans



Look for queries that may **scan a clustered index** to return a **small** amount of records from a **large table**



SQL Server stores missing index information and index utilization information in the database (***sys.dm_db_missing_index_details***)



Indexes can **impact write** performance and **consume space**, yet the performance gains offset the additional resource costs many times over



Drop an index that is not used by any queries in the database

Hardware constraints



Typically manifested through **wait statistics**



SOS_SCHEDULER_YIELD and CXPACKET wait types can be associated with **CPU pressure**, but can also point to **improper parallelism** settings or **missing indexes**



PAGEIOLATCH_* wait types can be indicative of **storage performance issues**



Detect CPU contention with '% Processor Time'



SQL Server will write to its error log if disk I/O takes longer than 15 seconds to complete



Storage system performance is best tracked at the operating system level with '**Disk Seconds/Read**' and '**Disk Seconds/Write**'

Statistics



Having up to date column and index statistics allows for the query optimizer to make the **best possible execution plan decisions**



SQL Server, Azure SQL Database and Managed Instance **default** to auto-updating statistics



In some cases, you may want to update statistics **more frequently** to ensure consistent performance for your queries

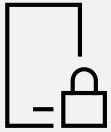


Use ***sys.dm_db_stats_properties*** to see the last time statistics were updated and the number of modifications that have been made since the last update

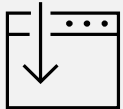


Auto-update statistics uses a dynamic percentage, which as the table grows, the percentage of row modifications that was required to trigger a statistics update gets smaller

Blocking and locking in SQL Server



Databases lock records and block other transactions as part of their normal behavior and their consistency model

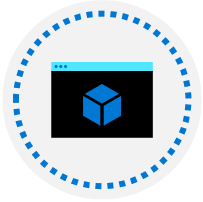


SQL Server uses a process called lock escalation to reduce the impact of these locks, taking low level locks and only escalating as needed

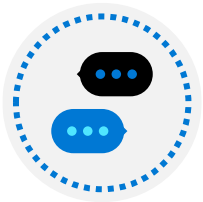
Blocking activity can be problematic in two main scenarios:

- ▶ Poor transaction handling in application code (leaving transactions open for too long)
- ▶ Transactions taking too long to complete because of poor indexing or external operations like linked server queries

Isolation levels



SQL Server offers multiple isolation levels to allow for a balance of concurrency and data quality



Isolation levels are specified at the session level, but may be overridden by the query



Lower isolation like **Read Uncommitted** may allow for more users to read the data, but increase the chance that the queries return incorrect results

Monitoring for blocking problems



The *Sys.dm_tran_locks* DMV contains all active locks, and can be joined with the *sys.dm_exec_requests* in order to provide statements holding locks



Querying DMVs is good for identifying a problem in real-time



For longer term monitoring creating an extended events session to monitor for blocking issues

What is parameter sniffing?

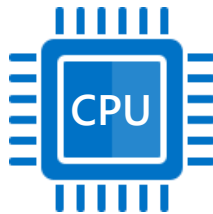


Default behavior in SQL Server

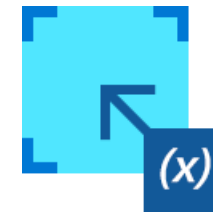


Optimizer makes decisions based on statistics, using initial parameter values

Parameter Sniffing



Reduces recompilations and CPU load



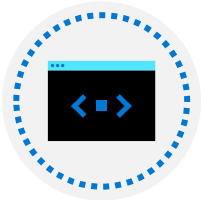
Local variables get standard assumption

Knowledge check



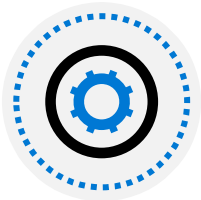
Which type of execution plan is stored in the plan cache?

- ☒ Estimated execution plan
 - ☐ Actual execution plan
 - ☐ Live Query Stats
-



Which DMV should you use to find index utilization?

- ☒ sys.dm_db_index_usage_stats
 - ☐ sys.dm_db_missing_index_details
 - ☐ sys.dm_exec_query_plan_stats
-



Which of the following wait types would indicate excessive CPU consumption?

- ☒ SOS_SCHEDULER_YIELD
- ☐ RESOURCE_SEMAPHORE
- ☐ PAGEIOLATCH_SH

Instructor led labs: Identify and resolve blocking issues

Run blocked queries report

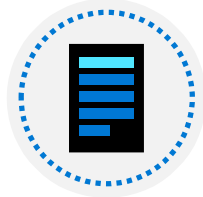
Enable Read Commit Snapshot isolation level

Evaluate performance improvements

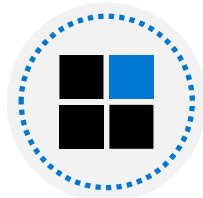
Explore Performance-based Database Design



Objectives



Explore normal forms and how they affect database design



Choose appropriate datatypes for your data



Evaluate appropriate index types

Normalization (First Normal Form)

Create a separate table for each set of related data

Eliminate repeating groups in individual tables

Identify each set of related data with a primary key

First Normal Form eliminates repeating groups, each row refers to a particular product, and there is a ProductID to be used a primary key

ProductID	ProductName	Price	ProductionCountry	ShortLocation
1	Widget	15.95	United States	US
2	Foop	41.95	United Kingdom	UK
3	Glombit	49.95	United Kingdom	UK
4	Sorfin	99.99	Republic of the Philippines	RepPhil
5	Stem Bolt	29.95	United States	US

Normalization (Second Normal Form)

If a table does not have a composite key, first and second normal form are the same

Same requirements as first normal form, plus..

If the table has a composite key, all attributes must depend on the complete key, not just part of it

ProductID	ProductName	Price	ProductionCountry	ShortLocation
1	Widget	15.95	United States	US
2	Foop	41.95	United Kingdom	UK
3	Glombit	49.95	United Kingdom	UK
4	Sorfin	99.99	Republic of the Philippines	RepPhil
5	Stem Bolt	29.95	United States	US

Normalization (Third Normal Form)

Same requirements as second normal form, plus..

All non-key relationships are non-transitively dependent on the primary key

Most OLTP databases should be in third normal form or as it is commonly abbreviated 3NF

ProductID	ProductName	Price	ProductionCountry	ShortLocation
1	Widget	15.95	United States	US
2	Foop	41.95	United Kingdom	UK
3	Glombit	49.95	United Kingdom	UK
4	Sorfin	99.99	Republic of the Philippines	RepPhil
5	Stem Bolt	29.95	United States	US

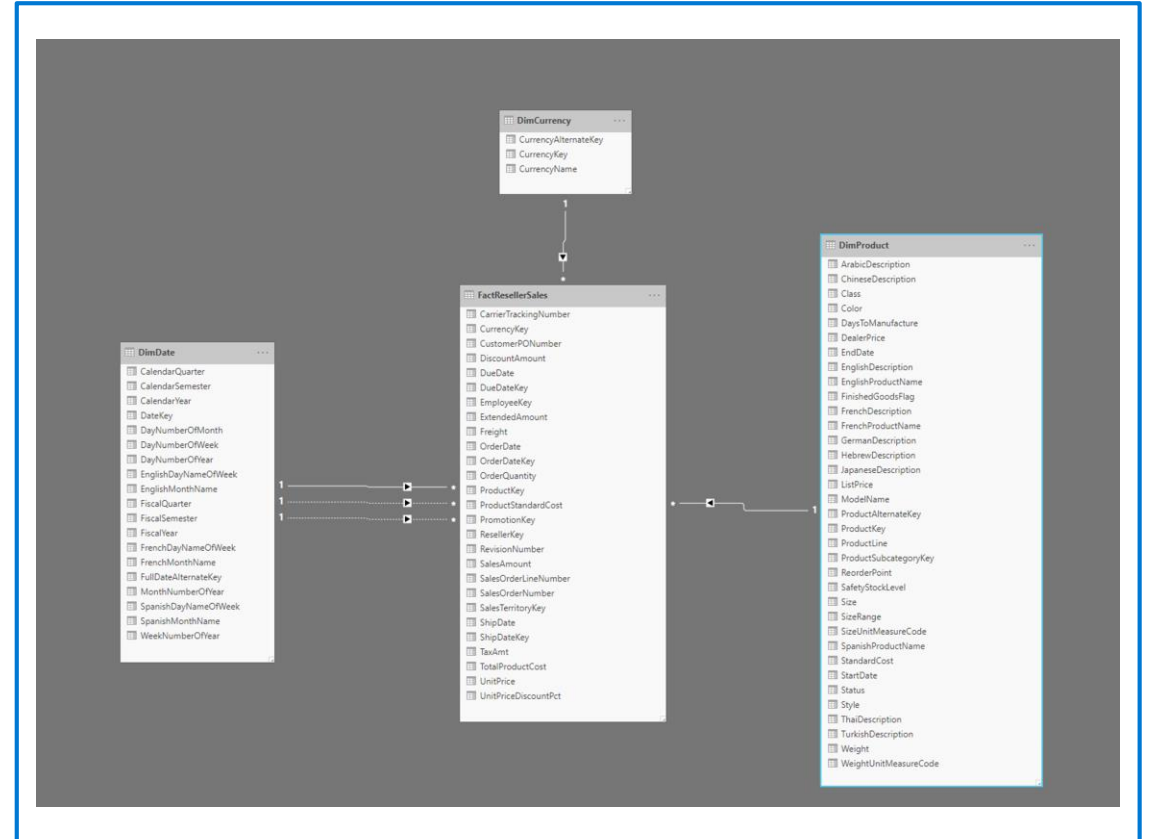
This table is NOT in 3rd normal form because ShortLocation depends on ProductionCountry, and not just on the primary key

Denormalization

Denormalized database designs are commonly used in data warehouses and data mart

This is helpful for reducing the number of joins required for reporting, but is less effective for updates

Two examples of denormalized structures are star (shown below) and snowflake schemas



Understand data types for performance

- ▶ Importantly, ensure that the data types in your application code match the data types in your database tables
- ▶ Conversions are expensive, and can cause much more expensive execution plans
- ▶ In general, smaller data types are better where possible
 - ▶ Data compression can help with this

Unicode data

Normal data types use **1 byte** per character

The problem with this is that the data type can only represent about **256 characters**

Unicode types use 2 bytes per character and can support up to **65,536 patterns**

Unicode data is needed for Cyrillic languages, Asian languages and many symbols

Types of indexes



Clustered
Indexes

Nonclustered
Indexes

Columnstore
Indexes

Rowstore
Indexes

Designing indexes



Cluster key values should be sequential (an increasing integer key is a good example here)



Clustered Indexes should be as narrow (minimal columns) as possible



In nonclustered index keys, the most selective column should be the leftmost column in the key



Use included columns in your nonclustered indexes where possible, to keep keys narrower



You should not create redundant indexes and avoid creating unused indexes on your tables

Clustered indexes



A clustered index sorts and stores the data in a table based on key values



There can only be one clustered index per table, since rows can only be stored in one order



Clustered indexes are frequently the primary key for a table



Clustered indexes have no uniqueness requirement

What are nonclustered indexes?



Nonclustered indexes are secondary indexes used to help the performance of queries not served by the clustered index



You can create multiple nonclustered indexes on a table



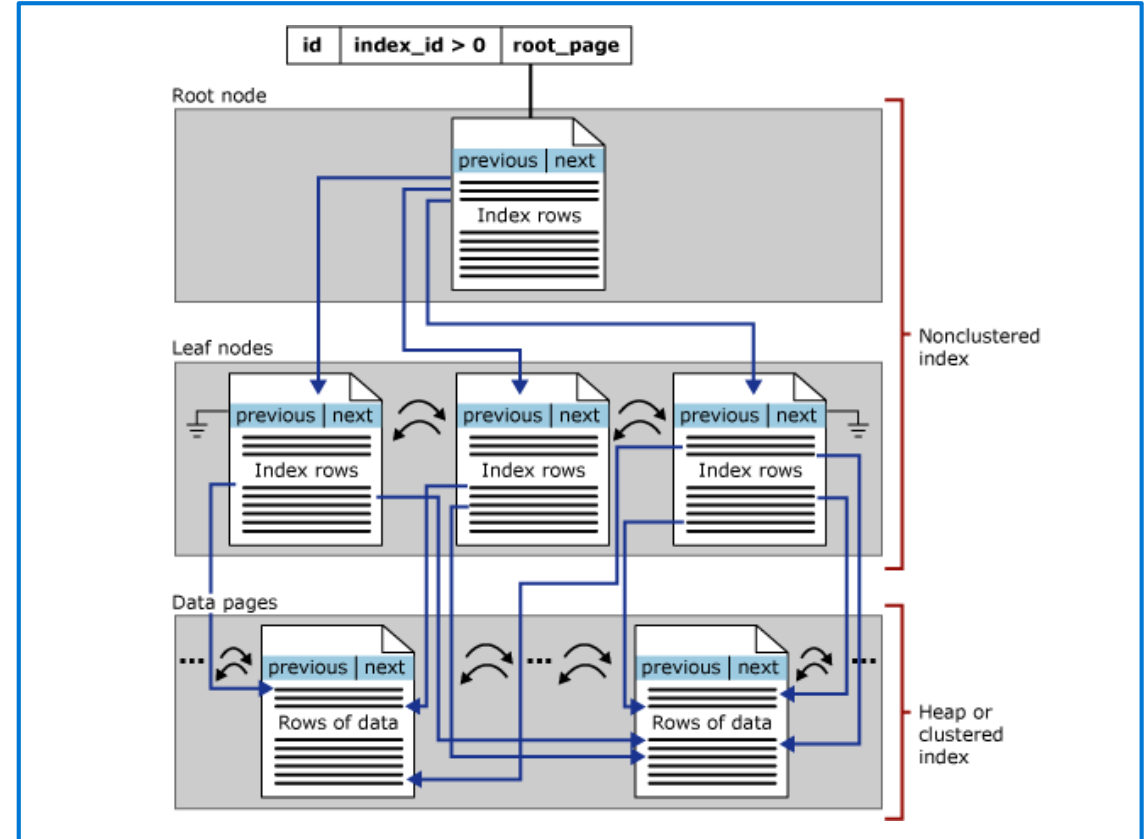
You can also create filtered indexes, for tables with large data skew



The cost of nonclustered indexes is space and insert/update performance

Nonclustered indexes

- Each page in an index b-tree is called an index node, and the top node of b-tree is called the root node.
- The bottom nodes in an index are called leaf nodes and the collection of leaf nodes is the leaf level.



Columnstore indexes



Data is stored in columns, not rows



This allows for higher levels of compression, which reduces storage and memory footprint of data



Columns that are not referenced in a query are not scanned, giving reducing the amount of I/O needed



Best used in large tables (e.g., data warehouse fact tables, temporal history tables)



Clustered columnstore must include all columns of the table



Use **clustered columnstore index** for fact tables and large dimension tables for DW workloads.
Use **nonclustered columnstore index** to perform analysis in real time on an OLTP workloads

Columnstore indexes – data load performance comparison

```
SQLQuery3.sql - VM...VM1\jdantoni (52))* X
SET STATISTICS TIME ON
INSERT INTO FactResellerSales_CCI_Demo
SELECT TOP 1024000 *
FROM FactResellerSalesXL_CCI
INSERT INTO FactResellerSales_Page_Demo
SELECT TOP 1024000 *
FROM FactResellerSalesXL_CCI

100 %
Messages
SQL Server parse and compile time:
  CPU time = 0 ms, elapsed time = 6 ms.

SQL Server Execution Times:
  CPU time = 7609 ms, elapsed time = 7902 ms.
(1024000 rows affected)

SQL Server Execution Times:
  CPU time = 17031 ms, elapsed time = 19685 ms.
(1024000 rows affected)

Completion time: 2020-04-22T14:02:07.5526154+00:00
```

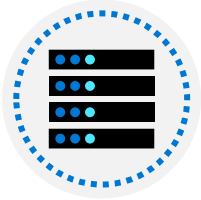
Clustered columnstore index

Clustered index with page compression

8 secs

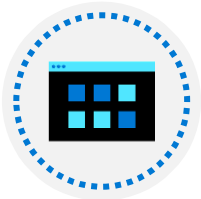
20 secs

Knowledge check



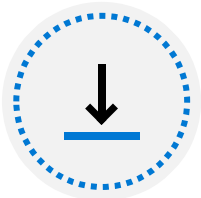
What type of database design should you use for a data warehouse when you want to reduce the data volume of your dimensions?

- ☐ Star Schema
- ☐ 3rd normal form
- ☒ Snowflake schema



What is the minimum number rows do you need to bulk insert into a columnstore index?

- ☐ 1,000,000
- ☒ 102,400
- ☐ 1000



Which compression type offers the highest level of compression?

- ☐ Page Compression
- ☐ Row Compression
- ☒ Columnstore Archival

Instructor led labs: Identify database design issues

Examine the query and identify the problem

Identify ways to fix the warning message

Improve the code

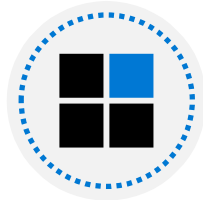
Evaluate Performance Improvements



Objectives



Describe wait statistics



Understand important aspects when tuning indexes



Explore when to use query hints

Wait statistics



Analysis of wait statistics is a holistic method of evaluating server performance



The database engine tracks what each thread is waiting on



This wait data is logged into memory and then persisted to disks



This can be helpful to isolate hardware, code, and configuration problems



Stored in *sys.dm_os_waits_stats* and in *sys.dm_db_wait_stats* in Azure SQL Database

Examples of wait statistics

High **RESOURCE_SEMAPHORE** waits

This is indicative of **queries waiting on memory to become available**, and may indicate excessive memory grants to some queries

High **LCK_M_X** waits

This can indicate a **blocking problem** that can be solved by either changing to the READ COMMITTED SNAPSHOT isolation level, or making changes in indexing to reduce transaction times, or possibly better transaction management within T-SQL code

High **PAGEIOLATCH_SH** waits

This can indicate a **problem with indexes**, where SQL Server is scanning too much data, or if the wait count is low, but the wait time is high, can indicate storage performance problems

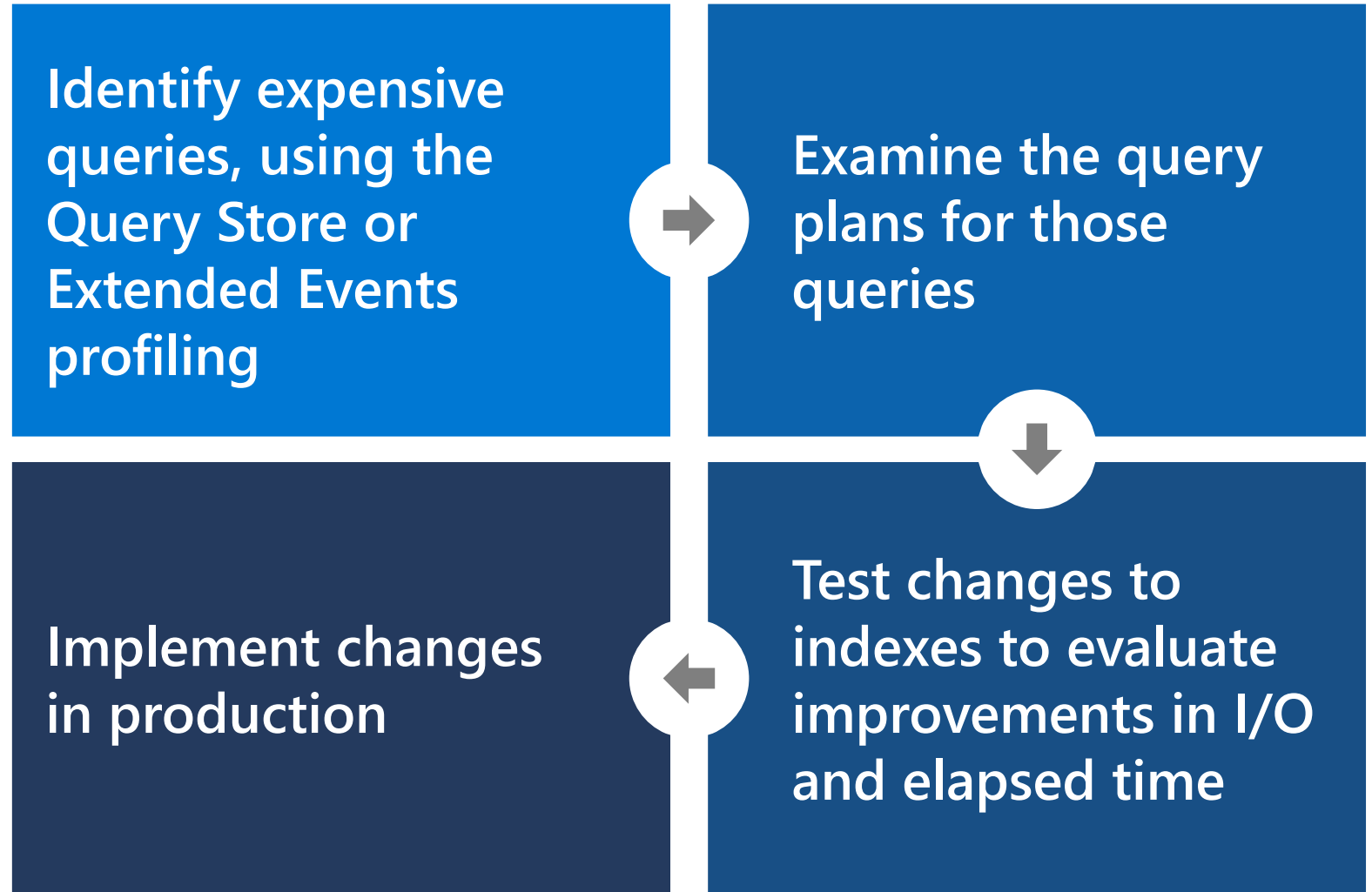
High **SOS_SCHEDULER_YIELD** waits

This can indicate **high CPU utilization** - very high workloads, or missing indexes

Demo: Query wait statistics

Querying sys.dm_os_wait_stats
Reviewing Query Store Wait report

Index tuning methodology



Index tuning



The goal is faster queries using the fewest indexes and IOs

Common tuning approach:

- ▶ Evaluate existing index usage - *sys.dm_db_index_operational_stats* and *sys.dm_db_index_usage_stats*
- ▶ Eliminate unused and duplicate indexes
- ▶ Review and evaluate expensive queries from the Query Store or Extended Events and refactor where appropriate
- ▶ Create the index(s) in a non-production environment and test query execution and performance
- ▶ Deploy the changes to production

Index maintenance



Over time, inserts, updates and deletes can cause fragmentation in an index

- ▶ Fragmentation causes the data to occupy more pages than it otherwise would
- ▶ Fragmentation can cause query performance degradation
- ▶ Indexes should be regularly **reorganized** or **rebuilt**
- ▶ SQL Server and Azure SQL Database allow for indexes to rebuilt **online**
- ▶ Newer versions of SQL Server allow **resumable** index operations
- ▶ Stored in the DMV *Sys.dm_db_index_physical_stats*

Index maintenance

Reorganization



- Defragments an index by physically reordering the leaf-level index pages to match the logical sorted order of the leaf nodes
- Compacts the index pages based on the index's fill factor setting
- It is an online process

Rebuilding



- Drops and recreates the pages of the index
- Causes the statistics to be updated
- Use when fragmentation is greater than 30%
- It can be either online or offline

Columnstore index maintenance



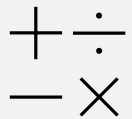
Columnstore indexes also require maintenance due to fragmentation

- ▶ If 20% or more of the rows are deleted, you should consider **reorganizing** your index
- ▶ Fragmentation is caused when deletes and updates happen as they both leave deleted rows in the columnstore index
- ▶ These deleted records are not included in query results, but are scanned in each query
- ▶ Fragmentation is measured by looking for deleted rows in the *sys.dm_dm_colum_store_row_group_physical_stats* DMV

Query hints



Query hints specify that the indicated behavior should be used throughout the query

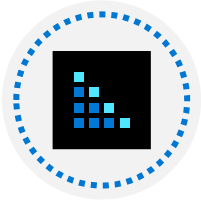


They affect all the operators in the query

Some common use cases for query hints include:

- ▶ Limit the amount of memory granted to query
- ▶ Force the use of a specific index
- ▶ Control join behavior

Knowledge check



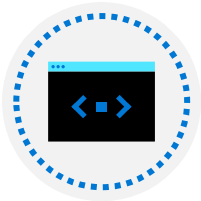
What type of index is best used on a data warehouse fact table?

- ☐ Nonclustered Columnstore
 - ☐ Clustered b-tree
 - ☒ Clustered Columnstore
-



Which DMV provides information about server level wait statistics?

- ☐ Sys.dm_exec_session_wait_stats
 - ☒ Sys.dm_os_wait_stats
 - ☐ Sys.dm_db_index_physical_stats
-



Which DMV can you use to capture the actual execution plan for a given query?

- ☐ sys.dm_exec_query_plan
- ☐ sys.dm_exec_cached_plans
- ☒ sys.dm_exec_query_plan_stats

Instructor led labs: Isolate problem areas in poorly performing queries in a SQL Database

Generate actual execution plan

Resolve a suboptimal query plan

Use Query Store to detect and handle regression

Examine Top Resource Consuming Queries report

Force a better execution plan

Use query hints to impact performance

Summary

Explore query performance optimization:

- Understand how plans are generated
- Understand query plan warnings and guidance
- Explore the Query Store
- Types of query execution plans

Explore performance-based database design:

- Learn the concepts of database normalization
- Choosing the proper data types for your database
- Understand the types of indexes in SQL Server

Evaluating performance improvements:

- Capturing data from dynamic management views and functions
- Understand wait statistics
- Understand index and statistics maintenance

References

SQL Server Execution plans:

<https://www.red-gate.com/simple-talk/books/sql-server-execution-plans-third-edition-by-grant-fritchey/>

Monitoring performance by using the Query Store:

<https://docs.microsoft.com/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store>

Query processing architecture guide:

<https://docs.microsoft.com/sql/relational-databases/query-processing-architecture-guide>

Index architecture and design:

<https://docs.microsoft.com/sql/relational-databases/sql-server-index-design-guide>

Thank you

