SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Module 6: Analyzing Query Performance

Erin Stellato

Erin@SQLskills.com

Before You Analyze and Tune: Identify Change

- **Changes in query performance can result from a variety of factors, such as:**
  - Changes in data
  - Code and schema changes
  - SQL Server version/edition changes
  - Hardware changes (e.g., different processors)
- **Knowing how to find changes in query performance easily and quickly is essential – whether they are planned or not**
- **Historically, capturing this information has been a manual effort**

Options for Identifying Change

- **Testing for planned changes:**
  - Established baselines from manual capture or third-party utilities
  - Distributed Replay Utility
  - Database Experimentation Assistant
  - Query Store
- **Finding unplanned changes:**
  - Established baselines from manual capture or third-party utilities
  - Using DMVs, logs, etc. in SQL Server on the fly
  - Query Store

Overview

- Capturing changes in query performance
- Capturing and analyzing plans
- Common operators
- Essential information in a plan

Sources for Data Related to Query Performance

- SQL Server Management Studio
- DMVs
- Extended Events/Trace
- Query Store
- PerfMon

Data of Interest

- **Query text**
- **Query plan**
- **Compiles/recompiles**
  - Reason for recompile
- **Query execution data**
  - Individual metrics
    - SSMS
    - Extended Events and Trace
  - Aggregated metrics
    - DMVs
    - Query Store
    - PerfMon
- **The source you use will depend on your version of SQL Server and the problem you're trying to solve**

SSMS

- **Actual and estimated plans**
- **STATISTICS I/O and STATISTICS TIME**
- **Client statistics**
- **Live Query Statistics**
  - Available in SQL Server 2016, but can work with SQL Server 2014 using later versions of SSMS (17.x)
  - Can affect query performance

DMVs

- **sys.dm_exec_sql_text**
  - Provided a sql_handle, returns text of the batch
- **sys.dm_exec_cached_plans**
  - One row for each query plan currently in memory
- **sys.dm_exec_query_plan**
  - Provided a plan_handle, it returns the plan in XML format
- **sys.dm_exec_text_query_plan**
  - Output is in text format
- **sys.dm_exec_query_stats**
  - Aggregate performance statistics for cached plans
- **sys.dm_exec_procedure_stats**
  - Aggregate performance statistics for cached stored procedures
- **sys.dm_exec_function_stats (added in 2016)**
  - Aggregate performance statistics for cached functions

Extended Events

- **sql_batch_completed**
  - Completion of T-SQL batch
- **rpc_completed**
  - Completion of a remote procedure call
- **sp_statement_completed**
  - Completion of T-SQL statement within a stored procedure
- **sql_statement_completed**
  - Completion of T-SQL statement
- **query_post_compilation_showplan**
- **query_pre_execution_showplan**
- **query_post_execution_showplan**

Not recommended

- **query_thread_profile**
  - Includes actual plan information for every operator and thread
  - *Use with caution*

Query Store

- **Described as a flight data recorder**
- **Captures information about query execution**
  - Query text
  - Query plan
  - Aggregated runtime statistics
- **Available in *all* editions of SQL Server**
  - Recommended to run the latest CU for all supported versions

Query Store Details

- **Enabled at the database level**
- **Data persisted in internal tables *in* the database which has Query Store enabled**
- **Cannot be enabled for master, tempdb, or model**
  - Default settings for Query Store are taken from the model database
  - http://www.sqlskills.com/blogs/erin/sql-server-query-store-default-settings/
- **Requires VIEW DATABASE STATE to view Query Store data**
- **Requires db_owner to force/unforce plans**
- **Data is not captured on readable secondaries**
  - It is propagated from the read/write primary to the read-only secondaries

Data Captured by Query Store

Query and Plan

- Compile, bind and optimization duration
- Compile memory
- Last execution time
- Context settings
- Query text
- Query plan

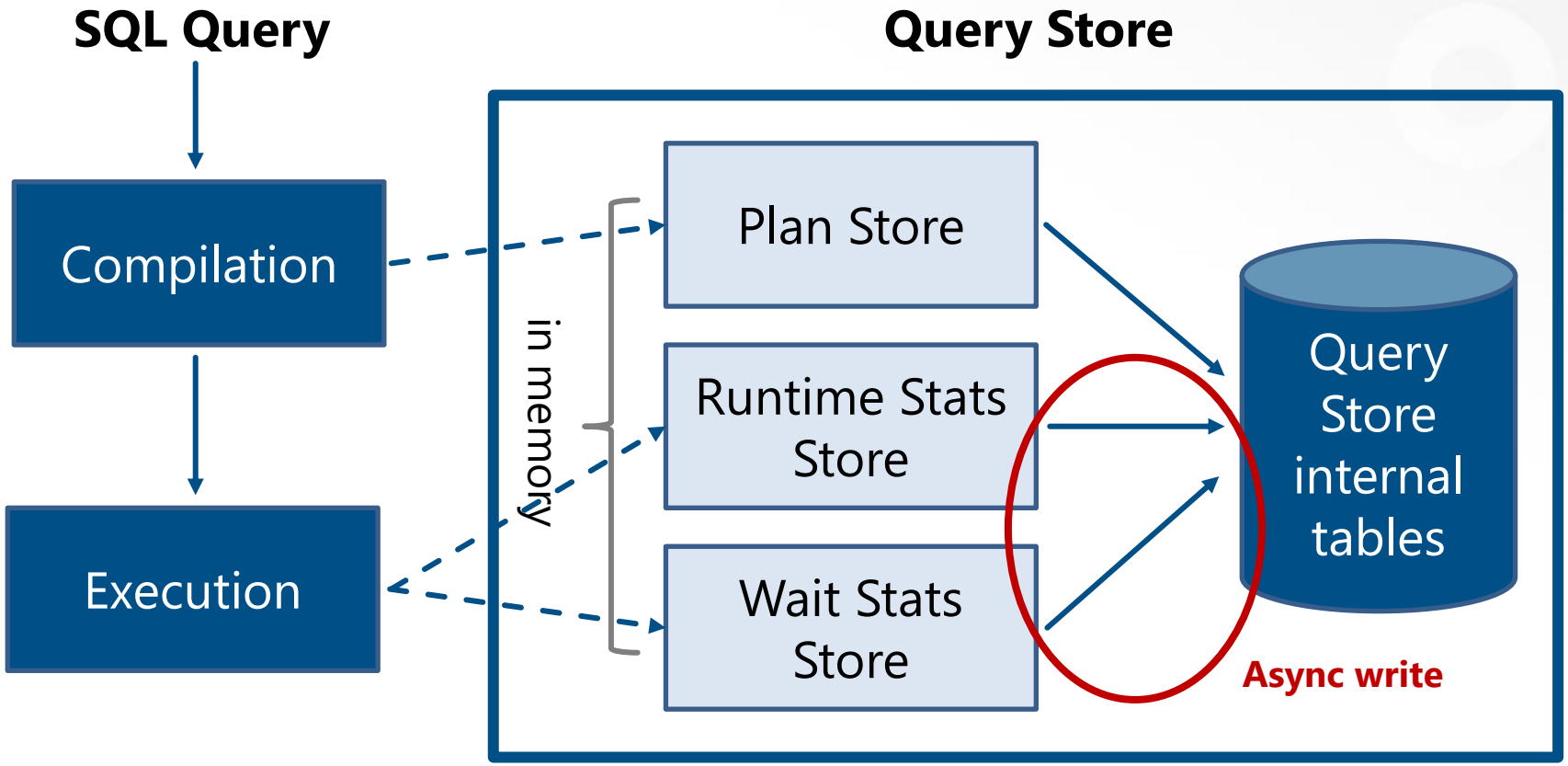
Runtime Stats

- Execution count
- Duration
- CPU
- Logical reads
- Physical reads
- Writes
- Memory use
- DOP
- Log bytes/used
- Tempdb

Wait Stats

- Wait statistics (per plan)

SQL Server 2017+ and Azure SQL Database



Adapted from: https://msdn.microsoft.com/en-us/library/mt631173.aspx

Query Store Settings

- There are nine settings related to query store configuration, and they affect what data gets collected and how it is stored
- Improper configuration can cause data to be removed from Query Store before expected, or Query Store can stop collecting data entirely

Query Store Settings

- **OPERATION_MODE = [READ_WRITE | READ_ONLY]**
  - **Why it matters:** If you're expecting READ_WRITE and QS is READ_ONLY, you need to understand why.
- **QUERY_CAPTURE_MODE = [ALL | AUTO | CUSTOM | NONE]**
  - **Why it matters:** Can affect "overhead" from QS, depending on your workload. While those "insignificant queries" may not be worth collecting, if you're looking for a specific query and can't find it, you need to understand why
- **MAX_PLANS_PER_QUERY = #**

Query Store Settings

- **MAX_STORAGE_SIZE_MB = #**
  - **Why it matters:** The default is 100MB or 1GB...what this needs to be depends on your workload and how much information you want to keep
  - "Ideal" size is 10GB or less (this is not documented)
- **CLEANUP_POLICY = (STALE_QUERY_THRESHOLD_DAYS = #)**
  - **Why it matters:** The default is 30, you may want to keep more data
- **SIZE_BASED_CLEANUP_MODE = [AUTO | OFF]**
  - **Why it matters:** If this is off and the space allocated to Query Store is completely consumed, it will switch to READ_ONLY

Query Store Settings

- **DATA_FLUSH_INTERVAL_SECONDS = #**
  - **Why it matters:** Affects how much QS data are you willing to lose
- **INTERVAL_LENGTH_MINUTES = #**
  - **Why it matters:** This will affect the space consumed by Query Store and the time windows across which you can analyze data
- **WAIT_STATS_CAPTURE_MODE [ON | OFF]**
  - **Why it matters:** Enabled by default (and enabled when upgrade to SQL 2017)
  - SQL Server 2017+ and Azure SQL Database

Query Store CUSTOM Capture Mode

- **STALE_CAPTURE_POLICY_THRESHOLD**
  - **Why it matters:** Sets the window of time for query evaluation
- **EXECUTION_COUNT**
  - **Why it matters:** If a query executes < N times, unless compile or execution CPU time exceed set values, it won't be captured
- **TOTAL_COMPILE_CPU_TIME_MS**
  - **Why it matters:** Estimation based on existing data may be difficult
- **TOTAL_EXECUTION_CPU_TIME_MS**
  - **Why it matters:** Existing data should help with estimation

Query Store Trace Flags

- **By default, SQL Server will wait until the Query Store that's in-memory is written to disk before fully shutting down**
  - This could delay a failover
- **TF 7745 bypasses writing Query Store to disk at shutdown**
  - Query Store data may be lost, the amount is dependent on the value for DATA_FLUSH_INTERVAL_SECONDS
- **Current configuration prevents queries from executing until all Query Store data has been loaded into memory**
  - May be an issue for larger data sets
  - Check for QDS_LOADDB wait type
- **TF 7752 allows queries to execute while Query Store data loads asynchronously during SQL Server startup**
  - Data about query execution will not be collected until Query Store data is loaded into memory
  - No longer needed in SQL Server 2019

Demo

Query Store configuration and use

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id

sys.query_store_query

plan_id	query_plan

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	1	400	400	400	400

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	2	600	800	800	400

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	3	566	500	800	400

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	4	500	300	800	300

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	894	378	401	800	295

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

query_id
786

sys.query_store_query

plan_id	query_plan
805	<ShowPlanXML xmlns="http://sc

sys.query_store_query_plan

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	894	378	401	800	295
2933	805	1	350	350	350	350

sys.query_store_runtime_stats

Understanding Runtime Statistics

```
SELECT [OrderID], [OrderDate], [CustomerPurchaseOrderNumber]
FROM [Sales].[Orders]
WHERE [CustomerID] = @CustID
ORDER BY [OrderDate];
```

runtime_stats_interval_id	start_time	end_time
2932	2021-04-19 12:00:00.0000000 +00:00	**2021-04-19** 13:00:00.0000000 +00:00
2933	**2021-04-19** 13:00:00.0000000 +00:00	2021-04-19 14:00:00.0000000 +00:00

sys.query_store_runtime_stats_interval

runtime_stats_interval_id	plan_id	count_executions	avg_duration	last_duration	max_duration	min_duration
2932	805	4	500	300	800	300
2933	805	1	350	350	350	350

sys.query_store_runtime_stats

Demo

Comparing query performance data

Query Store Overhead

- **Designed to have minimal overhead**
- **High volume, ad-hoc workloads may appear as though they have performance issues**
  - This does not mean that these issues weren't present in your workload already
  - On *any* version of SQL Server, you run the risk of running into performance issues because of the way your workload is designed
- **These workloads will require more space in Query Store**
  - It may not be possible to retain a significant amount of data
- **Performance optimizations added in SQL Server 2017 and SQL Server 2019 have been back-ported to SQL Server 2016**
- **Query Store Performance Overhead: What you need to know**
  - https://www.sqlskills.com/blogs/erin/query-store-performance-updated/

Monitoring Performance with Query Store Enabled

- **CPU (PerfMon)**
- **Memory**
  - sys.dm_os_memory_clerks
  - Monitor types that include QDS
- **query_store_db_diagnostics Event**
  - Database specific
  - In SQL Server 2016+
- **query_store_global_mem_obj_size_kb Event**
  - Instance level
  - Exists in SQL Server 2016+
- **If OPERATION_MODE frequently changes to READ_ONLY, review size of Query Store and how much data you're keeping**

Overview

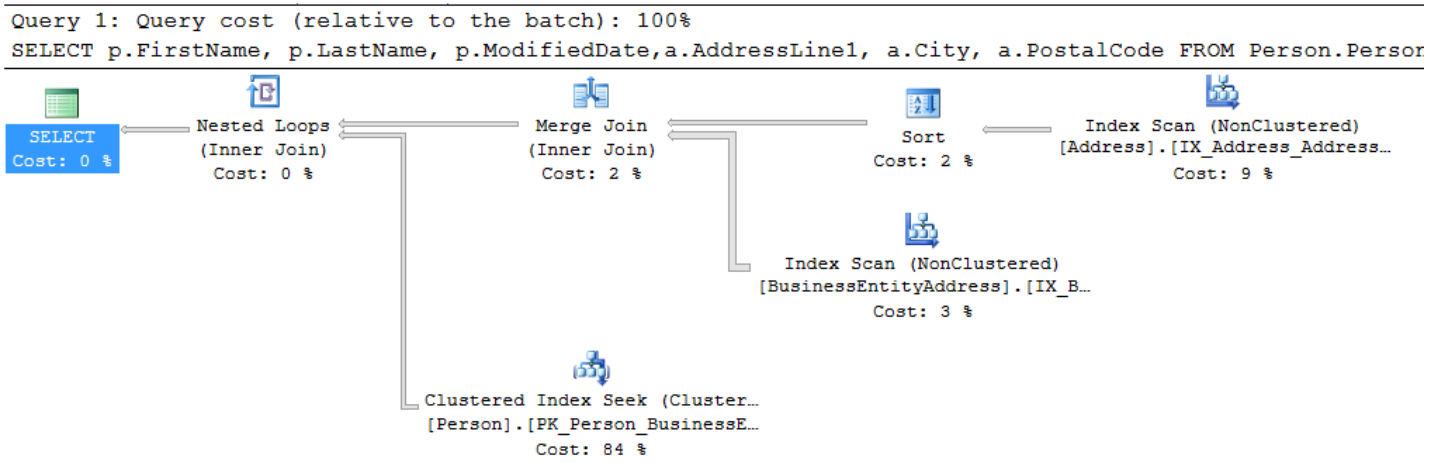
- Capturing changes in query performance
- Capturing and analyzing plans
- Common operators
- Essential information in a plan

Query Plan Analysis

- **Every query that is submitted to SQL Server has a query plan generated**
- **T-SQL is a declarative language: you tell SQL Server WHAT to do but not HOW to do it**
  - There are exceptions such as hints and plan guides
  - Features such as columnstore indexes are sensitive to query construction
- **We use query plans to find out HOW the results are being retrieved**
  - From the plan, we can try to optimize query performance
- **Query plans help you triage what occurred in the plan, and then address the areas that would benefit the most from improvement**
  - This insight gives you the ability to prioritize across statements within a batch, as well as operations within a specific statement

Query Optimization Principles

- Think of the optimizer as a framework used for finding a query plan
- The optimizer is given a query tree that has logical operations
  - Logical operations describe what will occur, such as a sort or a scan
- The optimizer then uses transformation rules to generate physical operations from the logical ones
- The physical operators, when assembled, create a query plan



Operators

- **Operators are building blocks for a query, each one has a specific functionality**
  - Some operators access data (scan, seek), others perform operations such as aggregations or joins
  - There is a one-to-many mapping of logical to physical
    - For example, an INNER JOIN could be implemented as a Loop, Hash, or Merge join
- **Specific operators are not "good" or "bad"**
  - Some can consume more resources than others or have overhead of which you should be aware
  - Some are more appropriate in given contexts
- **SQL Server 2019 has 100+ operators**
- **Operators may also be referred to as iterators**

Query Optimization and Cost

- **Each operation in the query tree is given a cost, and those are totaled to determine total cost for each possible plan**
- **Cost-based optimization looks at statistics and the size of the data that would be retrieved to estimate I/O**
  - The optimizer always assumes a cold cache
  - The optimizer assumes random I/Os will be spread out evenly among pages in an index or table
    - In some cases, an operation is over-costed
  - Additional assumptions exist (e.g. every query reads every row into the result set)

Operator Cost (1)

- Cost used to equate to elapsed time in seconds required to run on a specific Microsoft employee's machine (from SQL Server 7 era)

Operator Cost (2)

- **"Cost" today in the context of query plans is a unit-less measure**
  - Cost <> time
  - Cost <> CPU
- **Cost is used for relative comparison across plan operators and between plans**
- **"cost threshold for parallelism" option refers to this very same cost (default 5)**
  - The default value of 5 is very low considering today's CPU architecture
- **Estimated costs remain as "estimates" for actual plans**

Operator Cost (3)

- **Operator cost = I/O cost + CPU cost**
  - I/O cost assumes physical I/O required (i.e. cold cache)
- **Cost calculation varies by operator**
  - Some have I/O and CPU costs, some have just CPU cost
    - # of executions increase cost for specific operations
- **Sub-tree cost = cost of specific operator + descendants**
- **Total cost for the plan is found in root operator**
- **I/O cost assumptions (example):**
  - Data pages NOT in cache
  - Random I/O = 0.003125
  - Sequential I/O = 0.000740741
- **These aren't formally documented, but you can start calculating for yourself and mapping to I/Os via sys.dm_db_partition_stats and SET STATISTICS IO**

Cost Sensitivity to Row and Page Count

- **Estimated costs are sensitive to specific pieces of data**
  - e.g. the number of anticipated data pages and rows
- **Consider the Clustered Index Scan operator**
  - Estimated I/O cost of a Clustered Index Scan shows sensitivity to anticipated data page counts, but not row counts.
    - Think about the estimated I/O cost impact that high fragmentation may have (same number of rows across many partially-filled data pages)
  - Estimated CPU cost of a Clustered Index Scan sensitivity to row counts, but not data page counts.
    - Regarding row counts, even for narrow rows, consider the estimated CPU cost impact that high row counts may have.

Finding a Query Plan

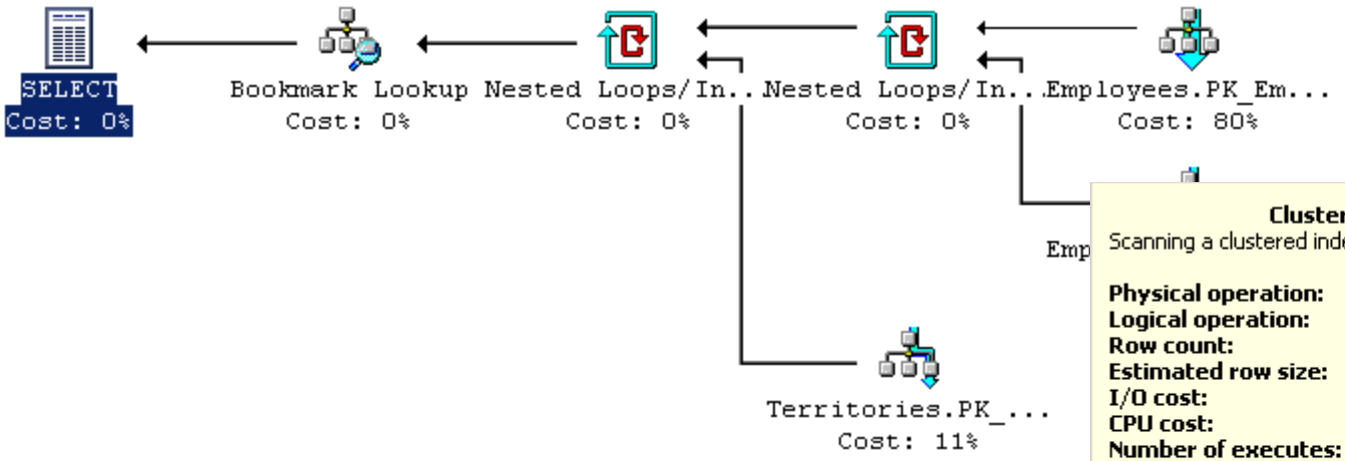
- **The majority of plans are stored in cache**
  - Exceptions covered in Module 9
- **The plan retrieved from cache and Query Store is the *query* plan (aka *estimated* plan)**
  - sys.dm_exec_query_plan or sys.dm_exec_cached_plans
  - sys.query_store_plan
  - SET SHOWPLAN_XML
  - Graphical Showplan
- **The *query plan* + *runtime statistics* (aka *actual* plan) can only be retrieved when you execute the query**
  - SET STATISTICS XML
  - Graphical Showplan
  - Trace or Extended Events…these are not recommended
  - sys.dm_exec_query_plan_stats in SQL Server 2019 and Azure SQL DB

"Estimated" vs. "Actual"

- **The plan that's stored in cache is the one that has been used**
  - This is the query plan. It does not contain runtime statistics.
- **When you retrieve the plan from cache, you only see estimates (number of rows, number of executions)**
- **When you capture the "actual" query plan, it's typically the same as the plan that exists in cache (though not always!) but it *also includes* actual runtime statistics (actual number of rows, actual number of executions)**
  - Useful to see if there is disparity between the estimates and actuals

Remember Plans in SQL 2000?

```
Query 1: Query cost (relative to the batch): 100.00%
Query text: SELECT e.LastName, e.Firstname, t.TerritoryDescription  FROM dbo.Employees e  JOIN (
```



Clustered Index Scan

Scanning a clustered index, entirely or only a range.

Physical operation:	Clustered Index Scan
Logical operation:	Clustered Index Scan
Row count:	5
Estimated row size:	79
I/O cost:	0.0375
CPU cost:	0.000088
Number of executes:	1
Cost:	0.037667(80%)
Subtree cost:	0.0376
Estimated row count:	5

	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp
1	SELECT e.LastName, e.Firstname, t.TerritoryDescription ...	1	1	0	NULL	NULL
2	|--Bookmark Lookup(BOOKMARK:([Bmk1002]), OBJECT:([Nor...	1	3	1	Bookmark Lookup	Bookmark Lookup
3	|--Nested Loops(Inner Join, OUTER REFERENCES:([e...	1	4	3	Nested Loops	Inner Join
4	|--Nested Loops(Inner Join, OUTER REFERENCE...	1	5	4	Nested Loops	Inner Join
5	|--Clustered Index Scan(OBJECT:([North...	1	6	5	Clustered Index Scan	Clustered Index Scan
6	|--Index Seek(OBJECT:([Northwind].[dbo...	1	7	5	Index Seek	Index Seek
7	|--Index Seek(OBJECT:([Northwind].[dbo].[Te...	1	8	4	Index Seek	Index Seek

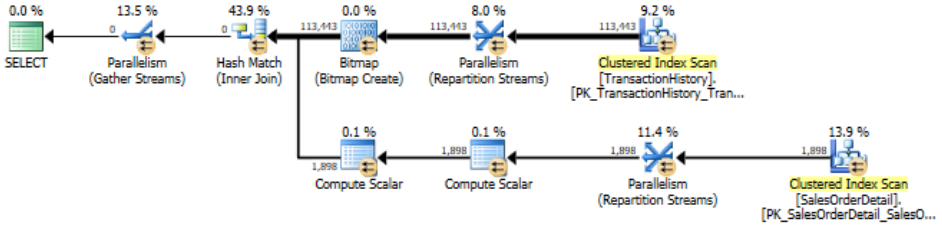
Plans in XML vs. Text

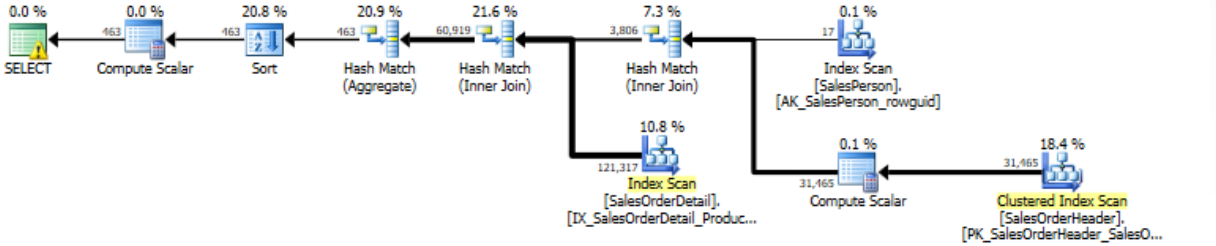
- **XML showplan formats may seem more unwieldy and verbose than their text-based / tabular counterparts, but there are some key advantages:**
  - Can be processed via XPath and XQuery
  - Easier to add attributes/elements from version to version
- **Saving the SHOWPLAN_XML or STATISTICS_XML output to a file with ".sqlplan" extension can be opened in SSMS in the graphical format (and XML)**
- **XML schema describes the structure of an XML document**
  - http://schemas.microsoft.com/sqlserver/2004/07/showplan/
  - Much of what you can find in the graphical plan you can see in XML, so the primary benefit is in parsing and finding key elements and attributes programmatically

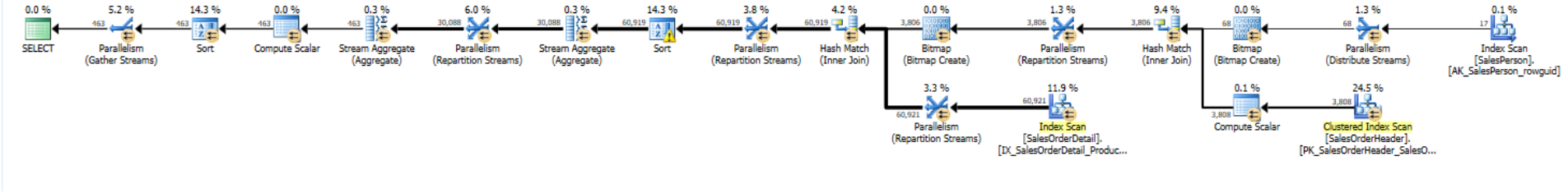
Information in a Query Plan

- **The plan tells you how the query was executed… how the optimizer decided to retrieve the results**
  - What tables and/or indexes were accessed
  - Whether scans or seeks were performed
  - What operators (a.k.a. iterators) were used
  - The estimated cost of each operation
  - How many rows were expected
  - More information continues to be added to plans
    - Duration
    - I/O information
    - Wait statistics

Where Do You Start With a Plan?





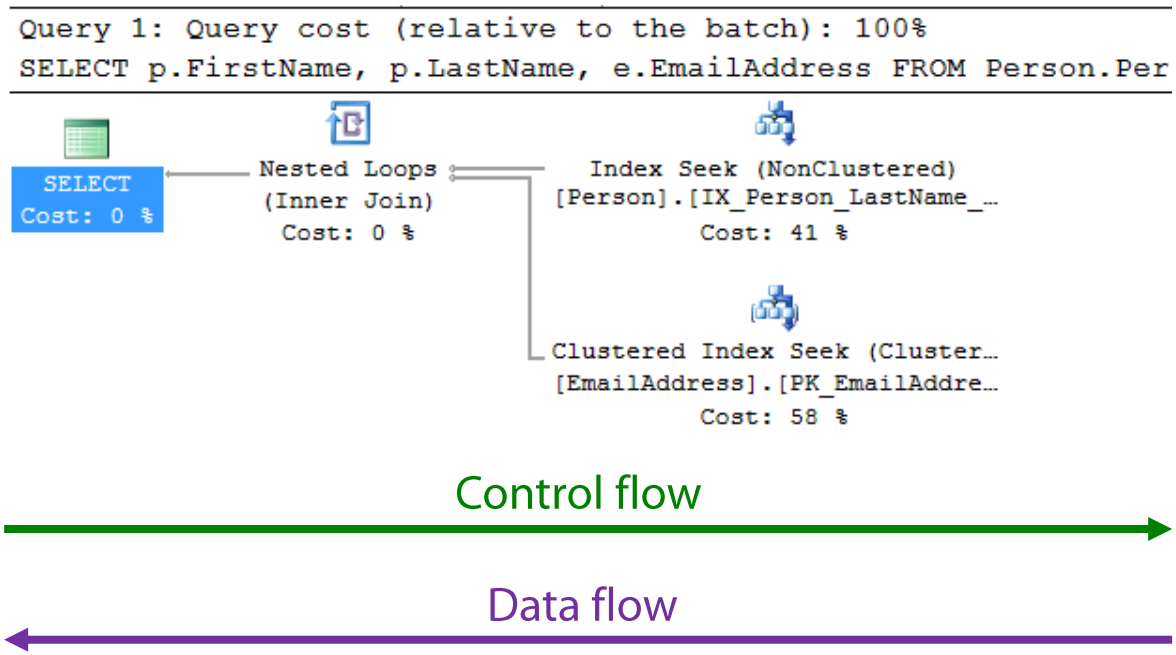


Where Do You Start With a Plan?

- **There is no one "right" answer**
- **Don't believe statements that specific operators or behaviors translate to an absolute problem**
  - We'll discuss things to watch for, but see them as areas of investigation
    - Some might be red herrings instead of red flags
- **The emphasis is on patterns you are more likely to see**
- **You have to practice reading and understanding plans to determine where you can improve a query, and this takes time**

Reading Plans

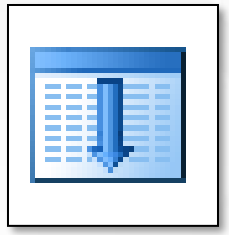
- An operator reads rows from a leaf-level data source OR from child operators and return rows to the parent
- Control flow starts at the root (left-to-right)
- Data flow starts at the leaf level (right-to-left)



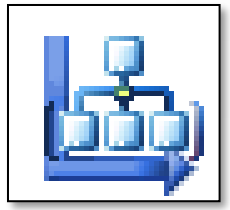
Overview

- Capturing changes in query performance
- Capturing and analyzing plans
- Common operators
- Essential information in a plan

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

Columnstore Index Operators

- Columnstore Index Scan 
- Build Hash: build of a batch hash table for a columnstore index 

- Key area to check: Batch vs. Row mode

Columnstore Index Scan (NonClustered)	
Scan a columnstore index, entirely or only a range.	
Physical Operation	Columnstore Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows	123695104
Actual Number of Batches	137744
Estimated I/O Cost	0.0068287

Filter

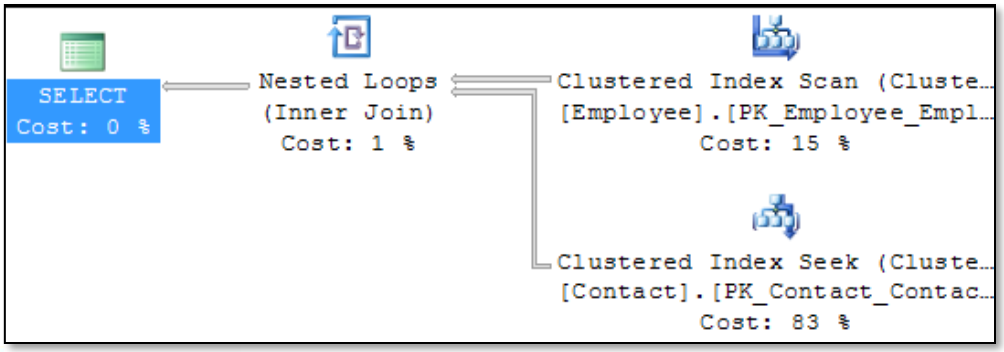
- Predicates can be evaluated within operators that read data from table/indexes
- Query Optimizer aims (when possible) to "push" filter down the tree (leaf level) to reduce rows moved
- If a predicate is high in cost or complexity a separate Filter operator may be used
- When you see these, take note 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)

Seek Predicate with No Residual

```
SELECT
[e].[EmployeeID],
[c].[Title],
[c].[FirstName],
[c].[MiddleName],
[c].[LastName],
[c].[Suffix]
FROM [HumamResources].[Employee] [e]
INNER JOIN [Person].[Contact] [c]
ON [c].[ContactID] = [e].[ContactID];
```



Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	290
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	290
Estimated Number of Executions	290
Estimated Operator Cost	0.071616 (83%)
Estimated Subtree Cost	0.071616
Estimated Number of Rows	1
Estimated Row Size	187 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3

Object

[AdventureWorks].[Person].[Contact].
[PK_Contact_ContactID] [c]

Output List

[AdventureWorks].[Person].[Contact].Title,
[AdventureWorks].[Person].[Contact].FirstName,
[AdventureWorks].[Person].[Contact].MiddleName,
[AdventureWorks].[Person].[Contact].LastName,
[AdventureWorks].[Person].[Contact].Suffix

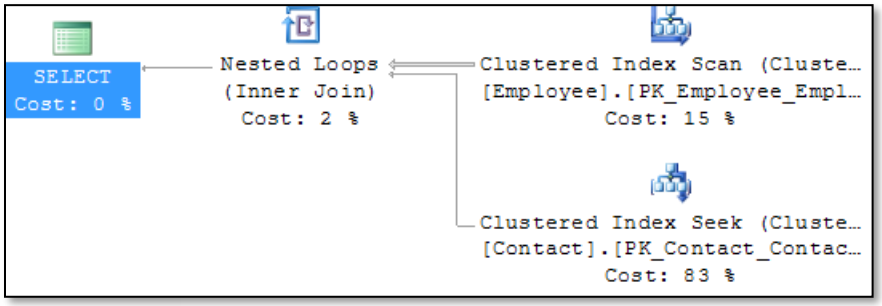
Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks].[Person].
[Contact].ContactID = Scalar Operator([AdventureWorks].
[HumanResources].[Employee].[ContactID] as [e].
[ContactID])

SQLskills
immerse yourself in sql server

Seek Predicate and Residual

```
SELECT
[e].[EmployeeID],
[c].[Title],
[c].[FirstName],
[c].[MiddleName],
[c].[LastName],
[c].[Suffix],
[c].[EmailAddress]
FROM [HumamResources].[Employee] [e]
INNER JOIN [Person].[Contact] [c]
ON [c].[ContactID] = [e].[ContactID]
WHERE [LastName] = 'Ellerbrock';
```



Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	290
Estimated Number of Executions	290
Estimated Operator Cost	0.071616 (83%)
Estimated Subtree Cost	0.071616
Estimated Number of Rows	1
Estimated Row Size	200 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3

Predicate

[AdventureWorks].[Person].[Contact].[LastName] as [c].[LastName]=N'Ellerbrock'

Object

[AdventureWorks].[Person].[Contact].[PK_Contact_ContactID] [c]

Output List

[AdventureWorks].[Person].[Contact].Title,
[AdventureWorks].[Person].[Contact].FirstName,
[AdventureWorks].[Person].[Contact].MiddleName,
[AdventureWorks].[Person].[Contact].LastName,
[AdventureWorks].[Person].[Contact].Suffix,
[AdventureWorks].[Person].[Contact].EmailAddress

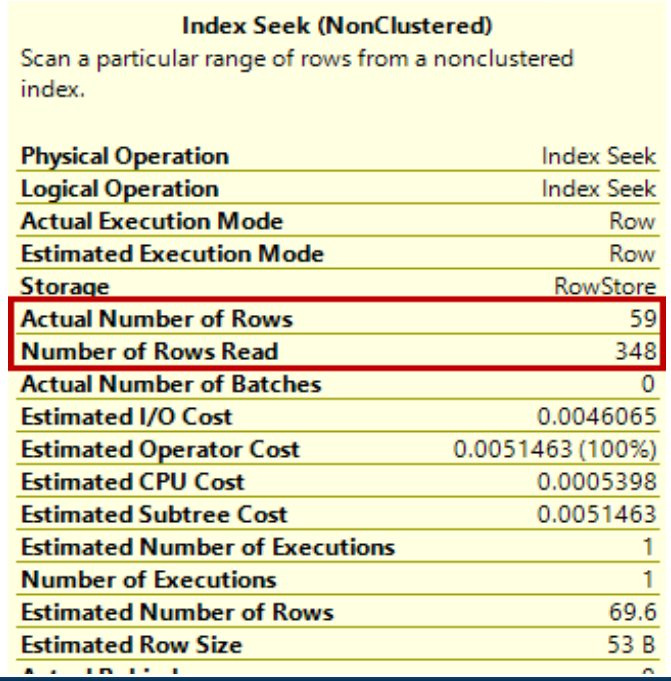
Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks].[Person].[Contact].ContactID = Scalar Operator([AdventureWorks].[HumanResources].[Employee].[ContactID] as [e].[ContactID])

SQLskills
immerse yourself in sql server

Additional Predicate Diagnostics

- ActualRowsRead attribute was added in SQL Server 2012 SP3 and SQL Server 2014 SP2 (also in SQL Server 2016+) to provide additional information about predicate evaluation
  - https://support.microsoft.com/en-us/kb/3107397
- Compare against ActualRows, which is the number of rows output from the operator



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	59
Number of Rows Read	348
Actual Number of Batches	0
Estimated I/O Cost	0.0046065
Estimated Operator Cost	0.0051463 (100%)
Estimated CPU Cost	0.0005398
Estimated Subtree Cost	0.0051463
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	69.6
Estimated Row Size	53 B

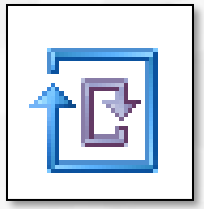
Demo

Data access operators

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

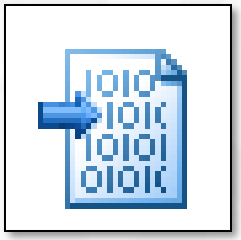


- **Uses each row from one input to find rows from a second input that satisfy the join predicate**
- **Usually seen with smaller data sets and lookups, where the inner input is indexed on the join predicate**
- **Algorithm:**
  - For one row in the outer (top) table, find matching rows in the inner (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

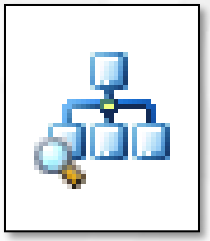
- **Look for "smaller" table as outer (top) table**
  - Bad cardinality estimates can lead to this NOT being the case
- **Nested Loops may be associated with inflated random I/Os when the row estimates end up being incorrectly estimated**
- **Look for under-estimates for inner table or index scans**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

Key Lookup



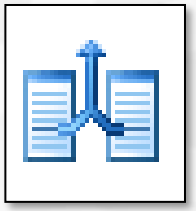
- **A.k.a. Bookmark Lookups**
  - SQL Server 2000: Bookmark Lookup
  - Early versions of 2005: "Clustered Index Seek" + keyword LOOKUP
  - 2005 SP2+: "Key Lookup" (but XML format still displayed "Clustered Index Seek")
- **Key Lookup = bookmark lookup on table with clustered index (always via Nested 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 applicable pages are cached, you can STILL have inflated overhead (compared to a covering index) due to the increased number of random logical reads

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

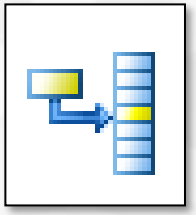


- **Joins two inputs which are sorted on the joining columns and returns matching rows**
  - Typically benefits moderate-sized data sets
- **Algorithm:**
  - Retrieve row from the outer input
  - Advance through the inner input until no more matches are found
  - Retrieve the next row from the outer 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)

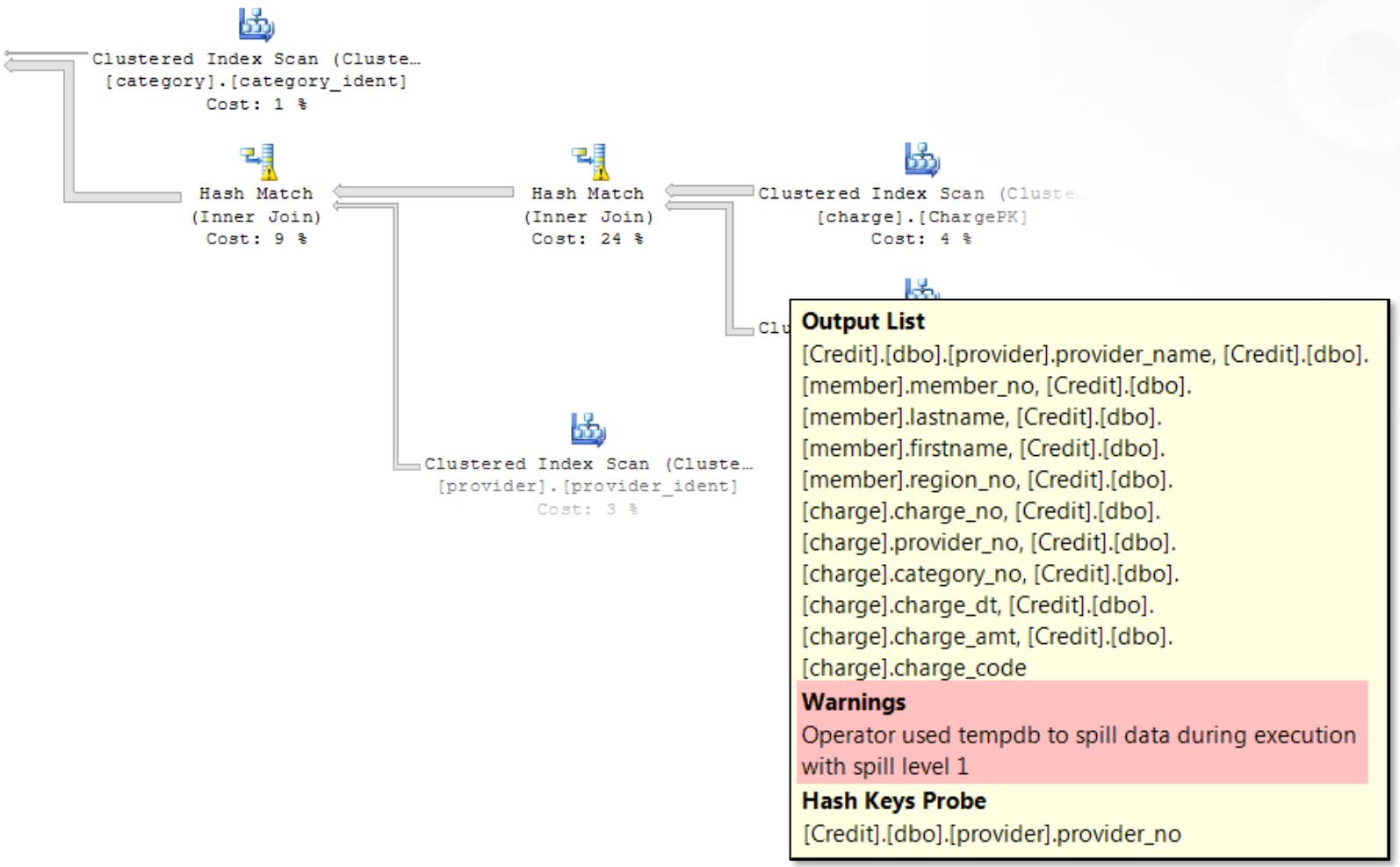
Hash Match Join Performance Characteristics

- 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)

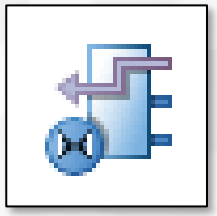
Hash Joins: Performance Variations

- **SQL Server can also do "role reversal"**
  - >= 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

Hash Joins: Plan Warnings in SQL Server 2012+



Batch Mode Adaptive Join



- **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**
- **Batch mode is used because a columnstore index is used, or a table with a columnstore index is referenced**
- **Introduced in SQL Server 2017, requires compatibility mode 140 or higher**

Demo

Join operators

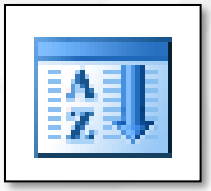
Query Memory

- **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 Match
  - 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!**

Stop-and-Go Operators

- **Stop-and-go operators must read ALL rows from the child operator before it can pass rows to the parent or perform specific actions**
- **Examples of stop-and-go operators include:**
  - Hash Join
    - Outer (top) table in "blocking input", not inner
    - Must read and process the entire build phase before the probe phase can start
  - Sort
  - Eager Spool
  - Hash Aggregate
- **Examples of operators that can keep streaming one row for every row that is read:**
  - Nested Loop
  - Merge Join
  - Compute Scalar

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

Operator Memory

- **Each type of operator requires varying amounts of memory in order to perform the associated operation**
- **Some operators require *more* memory because they cache rows**
  - More rows = more memory required
    - SQL Server performs estimates of the required memory and tries to reserve the memory grant prior to execution
    - This is where cardinality estimation is critical

Memory Grant Information (1)

- SQL Server 2012 expanded on memory grant information
- Provides estimates vs. actual
- Serial required/desired memory attributes estimated during query compile time for serial execution
  - Unit of measurement = KB
- Other attributes provide estimates that include parallelism

```
<MemoryGrantInfo
SerialRequiredMemory="6144"
SerialDesiredMemory="6760"
RequiredMemory="51464"
DesiredMemory="52104"
RequestedMemory="52104"
GrantWaitTime="0"
GrantedMemory="52104"
MaxUsedMemory="4680" />
```

Memory Grant Information (2)

- Additional information available in SQL Server 2014 SP2 and SQL Server 2016 SP1
- Expose maximum memory values for a single query
  - MaxCompileMemory
  - MaxQueryMemory
  - https://support.microsoft.com/en-us/kb/3170112
- If memory grant use is not efficient, e.g. only a fraction of allocated memory grant is used, the MemoryGrantWarning attribute is added to the plan
  - https://support.microsoft.com/en-us/kb/3172997

Controlling Memory Grants

- **The Resource Semaphore is what grants memory for queries, and ensures that the total amount of memory granted is within the server limit**
- **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
  - 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);
```

Memory Grant Feedback

- **To better manage query memory grants, a feedback mechanism was introduced in SQL Server 2017 for batch mode operators**
  - Reduces wasted memory from excessive grants
  - Reduces spills to disk from underestimates
  - Row mode memory grant feedback was added in SQL Server 2019
- **The memory required for a query can be recalculated after query execution, and the grant numbers are updated in the cached plan**
  - Recalculation occurs when a query wastes > 50% of memory allocated
  - A spill to disk will also force a recalculation
- **Memory grants respect the limits set by Resource Governor or query hints**
- **Feedback loop can be disabled after multiple executions with variable memory requirements**
  - Monitor with Extended Events

Finding Spill Information

- **Extended Events and Trace both include events for tracking Sort and Hash warnings**
  - Extended Events provides more detailed information, particularly in SQL Server 2016
  - Additional event fields which include DOP, memory, and tempdb IO, were ported back to SQL Server 2012 SP3 and SQL Server 2014 SP2
    - https://support.microsoft.com/en-us/kb/3107172
- **Additional information available in SQL Server 2012 SP3, SQL Server 2014 SP2, and SQL Server 2016 related to tempdb spills from a sort or hash**
  - https://support.microsoft.com/en-us/kb/3107400
- **Spill diagnostics added to module stats DMVs and statement events in SQL Server 2017**
  - https://support.microsoft.com/en-us/kb/4041814/

Demo

Operator memory

Overview

- Capturing changes in query performance
- Capturing and analyzing plans
- Common operators
- Essential information in a plan

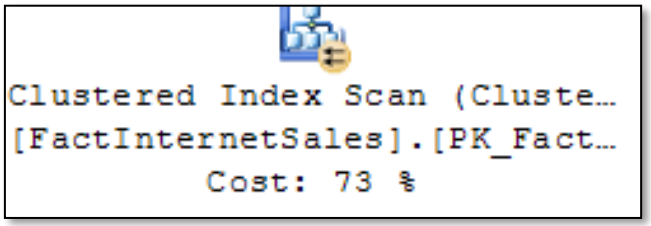
Parallelism

- **Query optimization can generate either a serial or parallel plan**
  - First determine cost of a serial plan, and if that cost is greater than CTP, consider parallel plans
  - The Query Optimizer ultimately caches one of them, not both
  - Parallel plans can then be used as intended, or run in serial with parallelism operators removed
- **The MAXDOP server setting limits:**
  - The maximum number of threads that can be used per operator
  - It does not limit the total number of threads for the plan

Identifying Parallelism in the Plan

- When parallelism occurs in query execution, you will see exchange operators in the plan
  - Distribute Streams
  - Repartition Streams
  - Gather Streams

- You will also see the parallelism icon in the graphic for other operators that can run in parallel or serial modes



- Not all operators are parallel-aware
- Within the XML Showplan you'll see RelOp:
  - Parallel="true"

Exchange Operators

- **Distribute Streams**
  - Takes one input thread and produces multiple output data streams

- **Repartition Streams**
  - Takes in multiple threads and then produces multiple streams out

- **Gather Streams**
  - Takes in multiple threads and produces a single stream out

NonParallelPlanReason

- **Introduced in SQL Server 2012**
  - Identify why a plan didn't run in parallel
  - Not all-encompassing, but a step in the right direction
  - Examples:
    - MaxDOPSetToOne (hint or max degree of parallelism)
    - EstimatedDOPIsOne (processor affinity)
  - Post by Simon Sabin that lists reasons for 2012:
    - http://sqlblogcasts.com/blogs/simons/archive/2015/04/26/non-parallelizable-operations-in-sql-server.aspx

Parallelism Performance Aspects

- **Not inherently bad or good**
  - Context matters, for example OLTP vs. DW
- **Indicates higher cost query, so that should attract attention**
- **Some objects and operators inhibit parallelism**
  - UDFs and TVFs (CLR, scalar, multi-statement), built-in functions (like OBJECT_ID), TOP
- **Watch for data skew across threads**
  - You can check this in XML or Properties window
  - Remember to also check for CXPACKET waits on non-zero thread IDs
- **Watch for memory pressure**
  - Increased memory grant requirements for parallel operations

Operator Execution Statistics (1)

- In SQL Server 2016 SP1, time and wait type information is added to the plan
- Within the QueryTimeStats attribute you can see CpuTime and ElapsedTime
- The top 10 wait types are also included within the WaitStats element
  - https://support.microsoft.com/en-us/help/3201552

```
<QueryTimeStats CpuTime="4" ElapsedTime="4" />
```

```
<WaitStats>
  <Wait WaitType="PAGEIOLATCH_SH" WaitTimeMs="4" WaitCount="31" />
  <Wait WaitType="SOS_SCHEDULER_YIELD" WaitTimeMs="5" WaitCount="843" />
  <Wait WaitType="ASYNC_NETWORK_IO" WaitTimeMs="72" WaitCount="6" />
  <Wait WaitType="MEMORY_ALLOCATION_EXT" WaitTimeMs="496" WaitCount="535073" />
</WaitStats>
```

Operator Execution Statistics (2)

- **In SQL Server 2014 SP2 and SQL Server 2016 per-operator query execution statistics are available in the XML**
  - RunTimeCountersPerThread
  - Includes CPU time, elapsed time, IO information
  - https://support.microsoft.com/en-us/kb/3170113

```
<RunTimeInformation>
  <RunTimeCountersPerThread Thread="0" ActualRebinds="1" ActualRewinds="0" ActualRows="18097" ActualEndOfScans="1"
                            ActualExecutions="1" ActualElapsedms="11" ActualCPUms="11" ActualScans="0" ActualLogicalReads="0"
                            ActualPhysicalReads="0" ActualReadAheads="0" ActualLobLogicalReads="0" ActualLobPhysicalReads="0"
                            ActualLobReadAheads="0" />
</RunTimeInformation>
```

```
 <RunTimeInformation>
   <RunTimeCountersPerThread Thread="4" ActualRows="5574" ActualRowsRead="5574" Batches="0" ActualEndOfScans="1" ActualExecut
   <RunTimeCountersPerThread Thread="3" ActualRows="7401" ActualRowsRead="7401" Batches="0" ActualEndOfScans="1" ActualExecut
   <RunTimeCountersPerThread Thread="1" ActualRows="4312" ActualRowsRead="4312" Batches="0" ActualEndOfScans="1" ActualExecut
   <RunTimeCountersPerThread Thread="2" ActualRows="14178" ActualRowsRead="14178" Batches="0" ActualEndOfScans="1" ActualExec
   <RunTimeCountersPerThread Thread="0" ActualRows="0" ActualEndOfScans="0" ActualExecutions="0" ActualElapsedms="0" ActualCP
 </RunTimeInformation>
```

Query Parameters

- The plan contains valuable information on compiled vs. runtime value
  - ParameterCompiledValue
  - ParameterRuntimeValue
- New attribute, ParameterDataType, added in 2016 SP1
  - https://support.microsoft.com/en-us/help/3190761/
- Some queries are very sensitive to different parameters
- Test different values using recompile (or a cold cache) to see if different plans are created for different values

Trace Flags Enabled

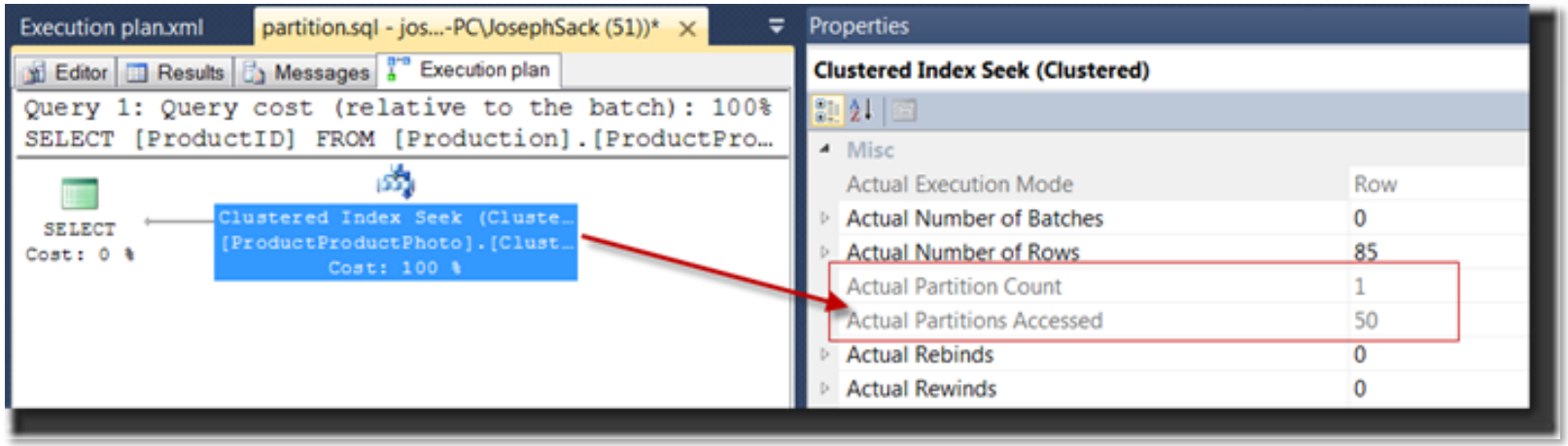
- The TraceFlags element is available in SQL Server 2014 SP2 which lists the trace flags enabled (global or session) at the time of query compilation (IsCompileTime = true) and execution (IsCompileTime = false)
  - https://support.microsoft.com/en-us/kb/3170115

```
<TraceFlags IsCompileTime="true">
  <TraceFlag Value="4199" Scope="Global" />
</TraceFlags>
<TraceFlags IsCompileTime="false">
  <TraceFlag Value="4199" Scope="Global" />
</TraceFlags>
```

```
<TraceFlags IsCompileTime="true">
  <TraceFlag Value="8649" Scope="Session" />
</TraceFlags>
```

Partitioning Plan Interpretation

- **Graphical vs. XML output could be misinterpreted**
  - "Actual Partition Count" = PartitionsAccessed element and PartitionCount attribute
  - "Actual Partitions Accessed" = PartitionRange




```
<PartitionsAccessed PartitionCount="1">
  <PartitionRange Start="50" End="50" />
</PartitionsAccessed>
```

Data Type Conversions

- **Explicit = CONVERT or CAST**
- **Implicit data type conversion, for example joining two table columns with different data types**
  - Data type with higher precedence "wins" and is converted
- **Can see CONVERT_IMPLICIT in plan**
  - In operator or in Compute Scalar
  - Or you can find in XML Showplan
  - You can parse plans from DMVs
  - Not always surfaced!

Summary: the "Watch List"

- High estimated cost within a query
- Scans (Index/Table)
- Lookups
- Unexpected join selection
- "Thick" lines
- Late filtering or residual predicates
- Sort operations (Query Optimizer added or not)
- Parallelism and thread skew
- Stop-and-go operators
- Missing indexes
- Warnings
- Implicit data-type conversions
- Parameter sensitivity
- Hint or plan guide usage (forcing specific plan operations)

Key Takeaways

- There are multiple sources for query performance data
  - It's important to understand what data exists where, and what should be used when
- The Query Store feature provides historical query data within SQL Server
- Where to start when analyzing a plan is a personal preference, and something that evolves over time…there is no "right way"
- Query tuning doesn't just consist of reviewing a plan in isolation, take advantage of multiple sources of information to improve performance
- Understand what behavior to "expect" for operators you see repeatedly, and take note of behaviors outside the norm (e.g. nested loops with under-estimates on the inner loop)
- Delve into plan details via the Properties window, and by mining the XML for information not easily surfaced in the UI

Additional Resources

- **Pluralsight Course**
  - SQL Server: Introduction to Query Store
    - https://bit.ly/2J42aJP
  - SQL Server: Analyzing Query Performance for Developers
    - https://bit.ly/2HKaoYo
  - SQL Server: Query Plan Analysis
    - http://bit.ly/ZyExm6
- **Whitepapers**
  - Statistics Used by the Query Optimizer in Microsoft SQL Server 2008
    - http://msdn.microsoft.com/en-us/library/dd535534.aspx
  - Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator
    - http://msdn.microsoft.com/en-us/library/dn673537.aspx
- **Books**
  - SQL Server Execution Plans by Grant Fritchey

Additional Resources

- **Blogs**
  - Craig Freedman's SQL Blog (not updated recently but very good!):
    - http://blogs.msdn.com/b/craigfr/
  - Conor Cunningham Blog(s):
    - http://blogs.msdn.com/b/conor_cunningham_msft/
    - http://www.sqlskills.com/blogs/conor/
  - Paul White's Blog
    - http://sqlblog.com/blogs/paul_white/
  - Benjamin Nevarez
    - http://www.benjaminnevarez.com/

Review

- Capturing changes in query performance
- Capturing and analyzing plans
- Common operators
- Essential information in a plan

Questions?