

SQLskills Immersion Event IE2: Performance Tuning

Module 7: Query Plan Analysis

Joe Sack
Joe@SQLskills.com



Overview

- Why look at query plans?
- How to capture query plans
- How to analyze query plans
- Understanding common operators
- Know which plan patterns to watch for and investigate further



2

© SQLskills, All rights reserved.
<http://www.SQLskills.com>



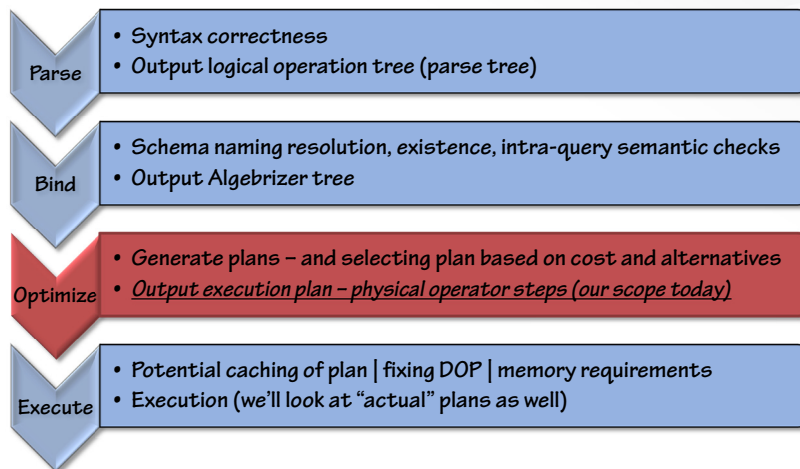
Why Do We Care About QPs?

- T-SQL is a declarative language: you tell it **WHAT** to do but not **HOW** to do it
 - There are exceptions such as hints, plan guides, and new features such as Columnstore Indexes are sensitive to query construction
- Use query plans to find out **HOW** the results are being retrieved and can potentially be optimized
- Query plans help you triage and address the most impactful areas needing improvement, prioritizing across statements within a batch and also operations within a specific statement
- Reveals key traits such as costs, indexes that are accessed, operators (a.k.a. iterators), rows accessed per operator, executions per operator, order of joins, and much more...

What Doesn't a Query Plan Tell Us?

- Doesn't tell us about locks acquired
- Doesn't show wait statistics
- Doesn't tell us if data is in the cache or not
- Doesn't tell us actual cost vs. estimated cost and I'll explain why later
- Some of the operations are hidden or surfaced in unexpected ways
 - I'll call these out as we go along
- Not a replacement of SET STATISTICS IO
- Not a replacement of SET STATISTICS TIME

Query Processing and Execution



"Capturing" a Plan

- **Estimated plan (compile):**
 - SET SHOWPLAN_ALL (on deprecation path)
 - SET SHOWPLAN_TEXT (on deprecation path)
 - SET SHOWPLAN_XML
 - Graphical Showplan
 - sys.dm_exec_cached_plans & sys.dm_exec_query_plan
- **Actual plan (runtime):**
 - SET STATISTICS XML
 - SET STATISTICS PROFILE (on deprecation path)
 - Graphical Showplan

Why Deprecate?

- 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
 - Meta-descriptions and flexible expansion across updates and versions (add attributes/elements)
- Can capture plan for plan guide use
- 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)
- Rich array of data to mine, and I'll call out opportunities as we go along

XML Plans and Showplan Schema

- XML Schema describes the structure of an XML document
 - <http://schemas.microsoft.com/sqlserver/2004/07/showplan/>
- Understand what to expect within the XML of a plan
- Can be used as a key for "parsing" via XQuery
- Much of what you can find in the graphical plan you can see in XML, so the primary benefit is in parsing and also finding key elements and attributes programmatically

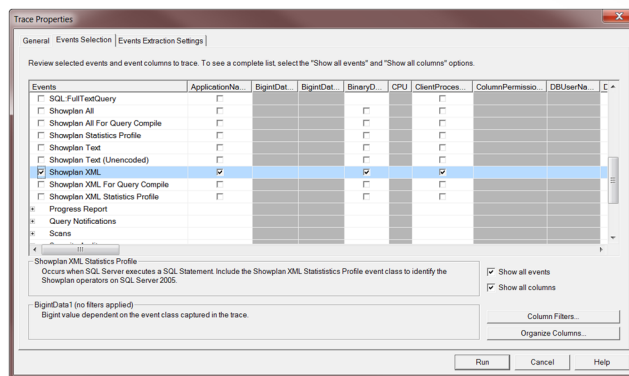
DMOs

- **sys.dm_exec_cached_plans**
 - One row for each query plan currently in memory
 - plan_handle is the plan identifier
 - objtype tells us what kind of cached object it is (e.g. ad hoc query, stored procedure, prepared statement)
- **sys.dm_exec_query_plan**
 - Provided a plan_handle, it returns the plan in XML format
 - Limit 128 levels of nested elements
 - Can be a cached plan or from currently executing request
- **sys.dm_exec_text_query_plan**
 - Output is in text format
 - No size limit
 - Can be used to specify a statement within a batch

Covering in more depth in later modules...

SQL Trace

- **Plans available to capture via SQL Trace, but be careful of the overhead!**
 - This is not a preferred method
 - Even Extended Events query plan event capture has non-trivial overhead, so disregard the emerging myth



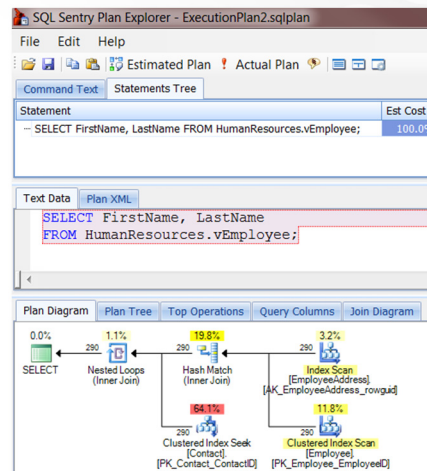
Graphical Showplan

- **Within SSMS**
 - “Display Estimated Execution Plan”
 - Parsed but not executed
 - “Include Actual Execution Plan”
 - Parsed and executed
- **Tree of icons (operators) connected by arrows**
- **Can still be useful for very large plans unless the cost distribution is extremely distributed (difficult to identify hot spots)**
 - This is where third-party options come in



Third-Party Options

- **SQL Sentry Plan Explorer**
 - Free tool
- **Good for plans from small to very large**
- **Pulls out key information in compact way**



Estimated vs. Actual

- **Estimated plan**

- Query not actually executed
- Plan represents how the statements will be executed
 - Pulling plan from cache shows estimated not actual
 - Parallel plan could become serial
- Ideally the row estimates are accurate, but when they aren't, this is one of the red flags which we'll discuss how to identify later

- **Actual plan**

- Includes estimated AND actual number of rows by operator and executions by operators
 - Key for detecting cardinality estimate issues

Demo

Comparing Showplan methods

Iterators / Operators

- The query plan is a tree of operators
 - Also called iterators
- SQL Server 2012 has 100+ operators
 - Logical and physical
- Think of them as extensible building blocks for a query with each implementing its own functionality
- The tree of operators is your query plan
- Specific operators are not “good” or “bad”
 - Some can consume more resources than other or have overhead that you should be aware of
 - Some are more appropriate in given contexts
 - We’ll discuss several types of operators later

Logical vs. Physical

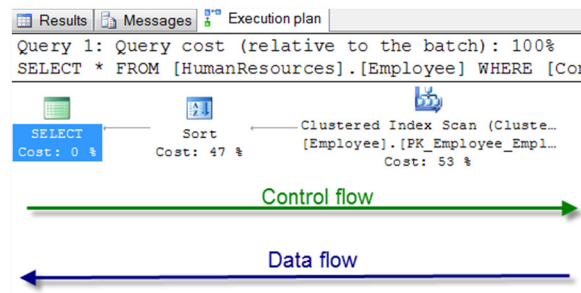
- Logical vs. physical operators
 - One-to-many mapping of logical to physical
 - As in: a logical operator could be implemented via Query Optimizer rules by multiple physical operators
 - INNER JOIN -> Loop/Hash/Merge
 - Physical operators implement the operation (e.g. Nested Loop Join)
 - You can see logical-to-physical mapping in the plan

Operator Methods

- Physical operators can use 3 method calls:
 - Init()
 - Causes operator to initialize itself and ready data structures
 - GetNext()
 - Operator retrieves first or subsequent row of data
 - # of calls translate to number of rows
 - Provides rows to the iterator that made the GetNext() call
 - Close()
 - Clean-up and shut down
- Context of calls depend on the operator and isn't fully documented

Query Tree

- Operators are combined into trees
- Operator reads rows from leaf-level data source OR from child operators and return rows to parent
- Control flow starts at root (left-to-right)
- Data flow starts at leaf level (right-to-left)



Operator Cost (1)

- Cost used to equate to elapsed time in seconds required to run on a specific Microsoft employee's machine (7.0 days)
- So really, "cost" today in the context of query plans is a unit-less measure
 - Cost <> time <> milliseconds
- 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)

Operator Cost (2)

- 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 add to this for specific operations
- Sub-tree cost = cost of specific operator + descendants
- Total cost found in root operator
- Estimated costs remain so on "actual" plans
- Costs between extremely different hardware configurations are static

Operator Cost (3)

- 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`

Demo

Exploring cost

Cost Sensitivity to Row and Page Count

- Estimated costs are sensitive to specific pieces of data, for example – the number of anticipated data pages and rows
- Let's take the Clustered Index Scan operator (for example)
 - 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)
 - 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.

Demo

Row and Page Count Influence on Estimated CPU and I/O Cost

Skewed Costs

- You may sometimes see total estimated operator cost exceed 100%
- Why? Some examples:
 - Some operators, like Concatenation, may not expect to pull both inputs, but the physical operator is still costed individually
 - Sometimes it's a graphical Showplan rendering bug
 - Sometimes it's a bug in costing itself (bad calculations)
 - Check connect.microsoft.com
 - It may or may not cause a bad plan selection

Operator Memory (1)

- 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

Query Memory (2)

- One area on the “watch list” are heavy memory consuming operators
 - Hash Match
 - Sort
- When available memory is insufficient, high memory requiring queries may wait to execute
- Under-estimating memory (due to CE issues) can cause spills to tempdb (I/O)
- Over-estimating memory can reduce concurrency!

Memory Grant Plan Info

- SQL Server 2012 expands 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" />
```

So What Do You Look For?

- The next few slides will detail key areas
- Don't believe statements that specific operators or behaviors translate to an absolute problem
 - I'll call out 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 and a few "edge" cases thrown in

Table and Index Scans

- **Table scan:** indicating a retrieval of ALL rows from a table
 - Red flag? Indicates a heap table
 - Red flag for larger tables (I/O)
- **Clustered Index scan:** indicating a retrieval of all rows
 - 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 so 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
- **To differentiate singleton or range scan operations, check the properties or XML <SeekPredicates>**
- **You can also validate**
 - sys.dm_db_index_operational_stats
 - range_scan_count
 - singleton_lookup_count

Demo

Index Seek: singleton or range scan?

Columnstore Index Operators



- Columnstore Index Scan
- Build Hash: build of a batch hash table for a columnstore index
- Hooks for SegmentReads and SegmentSkips
 - Not functional as of 11.0.2316, so use trace flag 646 to identify segment elimination
- 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

Key Lookup



- AKA "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 "Cluster 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

Key Lookup



- **Why do we care?**
 - For each row in the NCI, an associated Clustered Index I/O (random) is required
 - 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



- **Is just a bookmark lookup to a heap (using the RID)**
 - Just like with Key Lookups, you'll only see this with Nested Loop Joins
- **Why do you care?**
 - Same reasons as for Key Lookup
 - You also may care because you're going against a heap, warranting further questions/investigation

Demo

Lookups

Join Considerations

- Beware of advice telling you that specific join types (or operators, for that matter) are “good” or “bad”
- Key factors:
 - Tables to be joined, order of joins, algorithm used, cardinality, selectivity
- 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

Outer/Inner Terminology

- **When it comes to joins, you may hear the “outer” vs. “inner” table terminology**
 - But its NOT related to the order you write them in your query, unless you’re forcing it
 - Outer = top = left
 - Nested Loop: for each row in outer, find all rows in inner
 - Merge Join: inner/outer – not as important (will discuss why)
 - Hash Join: outer table is the “build” hash table
 - Inner = bottom = right
 - Hash Match: inner table is probe table

Nested Loop



- **Supports:**
 - Inner join, left outer join, left semi join, left anti semi join, cross join, cross apply, outer apply
 - Semi join returns columns from one of the tables where matches exist for both joined tables
 - Anti semi join returns columns from one of the joined tables where a match does NOT exist
- **Does not support:**
 - Right join, right semi-join, right anti-semi-join, full outer join (algorithm uses outer table to drive loop)
- **Only join type that permits inequality predicates and dynamic cursors**
 - Forcing does “Query processor could not produce a query plan because of the hints defined in this query. Resubmit the query without specifying any hints and without using SET FORCEPLAN.”

Nested Loop



- **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

Join Performance Characteristics

- **For a Nested Loop join:**

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

Demo

Nested loop

Merge Join



- **Logical operations:**
 - Requires one equijoin predicate
 - Except for full outer join transformation in many-to-many scenario
- **Joins two inputs (sorted on joining columns)**
 - Pre-existing sorting (via index) is ideal, but sorts can be automatically added (noteworthy if you see this)
- **Algorithm:**
 - Retrieve row from outer and inner tables
 - If a match: return the row
 - If no match: get a new row from the smaller input and iterate

Join Performance Characteristics

- **For a Merge Join:**
 - Memory requirements also lower (generally)
 - Hopefully rare... many-to-many joins have overhead (worktables)
 - Look for ManyToMany attribute
 - Outer (top) / inner (bottom) requires sort on join key
 - 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)

Demo

Merge

Hash Match Join



- **Joins of two unsorted inputs**
 - Can be used for de-duplication
 - Grouping aggregates
- **Requires one equijoin predicate**

Hash Match Join



- **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)
- **Effectiveness of hash-bucket distribution depends on hash key and hash function**
 - Too many or too few buckets = non-optimal
 - Function choice is proprietary

Demo

Hash match

Hash Join: Types

In Memory: build fits entirely in memory

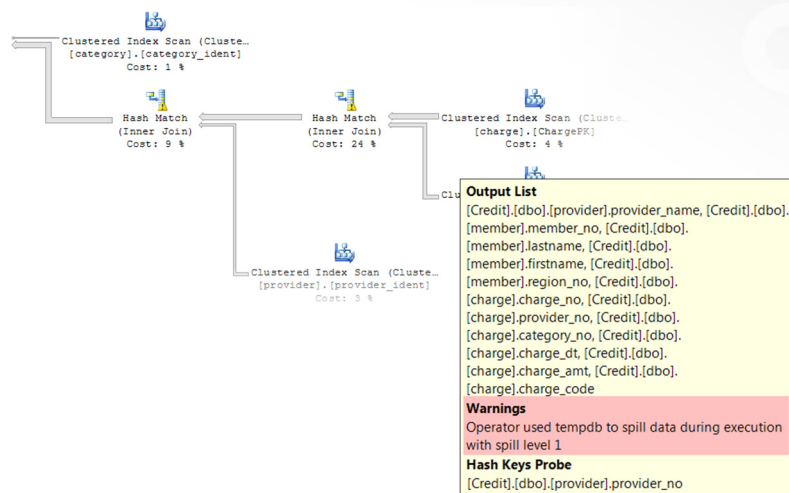
"Grace" (bailout): build cannot fit in memory and additional partitioning required into smaller "files"

"Recursive": additional sub-partitioning and large asynchronous I/Os used to drive speed. Keeps splitting until it fits into memory or a spill is needed.

Hash Joins: Performance Variations

- SQL Server can also do “role reversal”
 - $\geq$ one spill, build/probe roles can be switched
 - Not visible to us and should be rare
- Hash Warning events can be found in SQL Trace and in SQL Server 2012 – spill notifications within the plan
- Reasons included skewed data distributions, cardinality estimate issues, missing or stale statistics, inappropriate join selection or memory pressure
 - Estimates based on build cardinality and average row size

Hash Joins: Plan Warnings in SQL Server 2012



Join Performance Characteristics

- **For a Hash Join (Match):**
 - Typical case is that the smaller table is the “build table” (red flag if you see otherwise)
 - Hash table must be generated FIRST before the probe begins, so a smaller table would ideally reduce the latency between phases
 - Doesn’t require ordering of inner/outer
 - Hash build or probe can spill to disk if there is insufficient memory (higher memory requirements)
 - Duplicates can cause overhead, impacting bucket size (cannot be further divided)

Stop-and-Go Operators

- **Stop-and-go operators must read ALL rows from the child operator before it can pass rows to parent or perform specific actions**
- **Examples of stop-and-go operators include:**
 - Sort
 - Eager Spool
 - Hash Join
 - Outer (top) table in “blocking input”, not inner
 - Hash Aggregate
- **Examples of operators that can keep streaming one row for every row that is read:**
 - Nested loop
 - Compute scalar

Filter



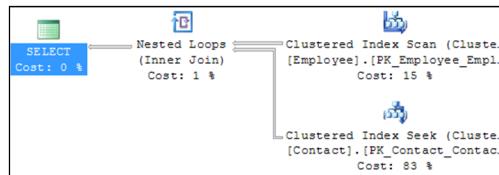
- Predicates can be evaluated within operators that read data from table/indexes
- Query Optimizer aims (when possible) to “push” filter down the tree (leaf level) to reduce rows moved
- If a predicate is high in cost or complexity a separate Filter operator may be used
- When you see these, take note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

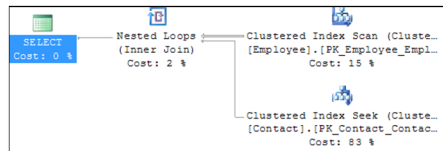
```
SELECT
    e.[EmployeeID],
    c.[Title],
    c.[FirstName],
    c.[MiddleName],
    c.[LastName],
    c.[Suffix]
FROM [HumanResources].[Employee] e
INNER JOIN [Person].[Contact] c
ON c.[ContactID] = e.[ContactID];
```



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])	

Seek Predicate and Residual

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



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	200 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3
Predicate	
[AdventureWorks].[Person].[Contact].(LastName) as [c].	
[LastName]='Elliott'	
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])	

Demo

Filters, predicates, residuals

Stream Aggregate



- **Groups rows by one or more columns**
 - Calculates one or more aggregate expressions
 - Does this one group at a time
 - Can do:
 - Scalar aggregates (no GROUP BY) e.g. SUM
 - Group aggregates (have GROUP BY)
 - Requires ordered (sorted) input for grouping columns
 - If unordered then Query Optimizer can add a Sort operator
- **Non blocking (streams)**
- **Minimal memory**

Hash Match (Aggregate)



- **Several aspects of hash join apply to hash match**
 - Requires memory for hash table
 - Unsorted inputs are okay
 - Is stop-and-go on the build table
- **For hash match, hash table generated for groupings of rows**
 - Hash table values based on grouping columns
 - 1) Generate hash
 - 2) Check for existing row in hash table
 - 3) Generate row if no match or update matching row

Hash Match (Aggregate)



- **Ideal for very large inputs**
- **Memory estimated is based on CE**
- **Memory needed = # groups / # rows**
- **Why do we care?**
 - If memory is limited, we have risk of spills (tempdb)
- **Unlike hash joins, hash aggregate incoming duplicates are not an issue**
 - They are de-duped into individual groups

Sort



- As named: orders rows received from input
- Variation is “Distinct Sort” to remove duplicates
- 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/IO/Memory)
 - May not be needed if you have supporting indexes or unnecessary ORDER BY

Spools



- **Eager Spool**
 - Takes entire input, placing row(s) in hidden tempdb work file
 - When spool's parent operator asks for first row, spool grabs all rows from the input and stores them in tempdb
- **Lazy Spool**
 - When the lazy spool's parent requests a row, the spool grabs the row from the input operator and stores it in the tempdb spool table
 - It does not retrieve all rows like the eager spool
 - Lazy spools are non-blocking

Eager Spool and “Halloween Protection”

- **Identified by IBM researchers back in 1976**
 - Performing an update involved, conceptually, a read cursor and a write cursor
 - The write cursor was updating the read cursor, causing a moving target of repeated updates to already-updated rows
- **Eager spools, when needed, prevent the write cursor from impacting the read cursor**
 - Example of when its not needed?
 - If you’re NOT updating the index key itself (causing movement that could make rows scanned > 1)
 - Other blocking operators already in use (such as Sort)

Spool Overhead

- Might be an optimization when used
- Index spools may point to a need for an index
- Spools, particularly those containing small data sets, can be in the buffer pool and not be written to disk (no I/O)
- If buffers allowed for the operation are exceeded, a spill will write out to disk
- Might see them with remote tables (linked servers / distributed queries) for local access
- Recursive CTE queries WITH STACK predicate, <Spool Stack="true">

Demo

Spools

Constant Scan



- Introduces ≥ 1 constant rows that can be built upon by parent operators
 - You can see this in data modification plans as well as SELECT plans
- SQL Server 2005: seen with partitioning as well, defining applicable partition numbers (driven by predicates)
 - Nested Loop operator joining Constant Scan (outer) and partitioned table (inner)

Assert

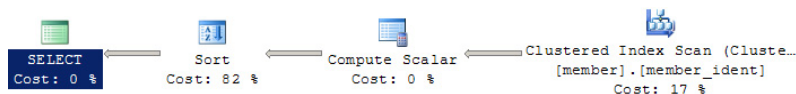


- **Verifies existence of a specific condition**
 - Check constraints
 - Referential integrity
 - Enforce return of one-row for scalar sub-query
 - Algorithm:
 - If non-NULL value, raise appropriate error, otherwise resume the query (pass through Assert)
 - CASE WHEN [Employee].[SickLeaveHours] < (0) OR [Employee].[SickLeaveHours] > (120)
 - THEN (0)
 - ELSE NULL END

Compute Scalar



- **Evaluates an expression, producing a scalar value (e.g. GETDATE() value)**
- **Estimated CPU cost is often low – for example 0.001 for an estimated 9,615 rows passing through an UPPER function**
 - May be a placeholder definition of an expression calculated in another operator – for example...
 - Compute Scalar
 - [Expr1003] = Scalar Operator(upper([Credit].[dbo].[member].[lastname]))
 - Sort
 - [Credit].[dbo].[member].member_no, [Credit].[dbo].[member].curr_balance, Expr1003

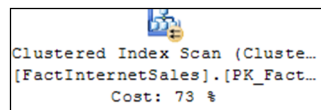


Parallelism

- **Query Optimization can generate both a serial and parallel plan**
 - But Query Optimizer ultimately caches one of them, not both (myth)
 - Parallel plans can then be used as intended, or run in serial with parallelism operators removed
- **MAX DOP limits:**
 - Max tasks an individual operator can spin up
 - Max concurrently executing tasks on schedulers
 - Maximum operators able to execute concurrently in the same plan

Identifying Parallelism in the Plan

- **Parallelism operators (Distribution/Repartition/Gather)**
- **You'll also see the parallelism icon in the graphic for operators that can run in parallel or serial modes**



- **If looking at XML Showplan you'll see RelOp:**
 - Parallel="true"

Exchange Operators

- **Distribute Streams**

- Takes one input and produces multiple output data streams
 - Routing most often determined by hashing
 - Partitioned data includes PARTITION COLUMNS in argument



- **Repartition Streams**

- Takes in multiple streams and then produces multiple streams
 - Can be used in conjunction with bitmap filter to reduce rows in output
 - May "rebalance" thread skew
 - Can be order preserving – and you'll see ORDER BY in argument
 - If partitioned data you'll see PARTITION COLUMNS in argument



- **Gather Streams**

- Operator takes in multiple streams and produces a single stream
 - Can be order preserving you'll see ORDER BY in argument



Consumer and Producer

- **Exchange operators are really made up of two independent operators:**

- **Producer**
 - Reads rows and pushes packets of rows to consumers
 - 8KB packet – which can explain uneven distribution
 - Distribute = 1 thread
 - Gather/Repartition = DOP limits threads
- **Consumer**
 - Receives packets of rows and returns to parent operator
 - Gather = 1 thread
 - Distribute/Repartition = DOP limits threads

Parallelism Branches

- Introduced in SQL Server 2012, branches provide a count of concurrent execution paths within the execution plan
 - UsedThreads: returns the maximum count of used parallel threads
 - ReservedThreads: count of parallel threads per NUMA node
 - Doesn't include coordinator thread (hidden) but you can see via sys.dm_os_workers and sys.dm_os_tasks

```
<ThreadStat Branches="3" UsedThreads="24">  
  <ThreadReservation NodeId="0" ReservedThreads="6" />  
  <ThreadReservation NodeId="1" ReservedThreads="6" />  
  <ThreadReservation NodeId="2" ReservedThreads="6" />  
  <ThreadReservation NodeId="3" ReservedThreads="6" />  
</ThreadStat>
```

NonParallelPlanReason

- Introduced in SQL Server 2012
 - Identify why a plan didn't run in parallel
 - Not all-encompassing as of SQL Server 2012, but a step in the right direction
 - Examples:
 - MaxDOPSetToOne (hint or max degree of parallelism)
 - EstimatedDOPIsOne (processor affinity)

Parallelism Performance Aspects

- **Not inherently bad or good**
 - Context matters, for example OLTP vs. DSS
- **Indicates higher cost query, so that should attract attention**
- **Some objects and operators inhibit parallelism:**
 - UDFs & 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
 - We'll demonstrate this
 - Busy concurrent tasks may cause transient skews
- **Watch for memory pressure**
 - Increased requirements for parallel operations

Demo

Parallel plan

Bitmap Operator

- **Bitmap**

- Performs bitmap filtering for parallel plans with hash or merge joins
 - Optimization that eliminates rows with key values that wouldn't produce join records
 - Reduces rows being passed to parent operators
 - Fact table ≥ 100 pages and smaller dimension tables
 - Only single column INNER JOINS with fact-to-dimension tables (but PK-to-FK not required)
 - Integer-based columns preferred



Data Modifications: Narrow Plans

- Also called a “per row” plan
- 1 row update of index at a time
- One operator handles modifications across multiple indexes
- Can have overhead if many rows involved, as row updates are based on clustered key order (random I/O)

Data Modifications: Wide Plans

- Also called a “per index” plan
- Wide plans: one index gets updated at a time (multiple rows) across multiple operators
 - Can be more efficient for many rows, but also has extra plan complexity
- Split-Sort-Collapse (for wide plans)
 - Operators that help with a phantom unique key violation
 - Split: splits UPDATE into DELETE and INSERT
 - Sort: sort operations that must occur to achieve net-change
 - Collapse: combines separate operations with more efficient operation

Demo

Data-modification plans

Partitioning Plan Interpretation

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

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

RetrievedFromCache attribute

- Added in SQL Server 2012
- Is "FALSE" when:
 - RECOMPILE hint is used
 - "optimize for ad hoc workloads" enabled (applies to first, stubbed execution)

Cardinality Estimate issues

- **Major red flag to watch for**
 - Skewed estimate vs. actual
- **Magnification and distortion as we move through the plan tree**
- **Several reasons why this could occur**
- **Other symptoms:**
 - Query performs badly or doesn't execute at all due to memory error
 - Performance may be good sometimes and bad other times

Cardinality Estimate Next Steps

- **Key areas to validate**
 - Query
 - Execution Plan
 - Statistics
- **The usual suspects:**
 - Missing indexes / missing or stale statistics
 - Table variables
 - TVF
 - Modifying in-flight variables
 - Data type conversions
 - Wrapping indexed columns in expressions

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!
 - Sometimes undetectable through keyword (but clues may exist)

PlanAffectingConvert

- Introduced in SQL Server 2012
- Warn when a convert might have a poor impact on plan choice - "Cardinality Estimate" or "Seek Plan"
 - False positives are quite possible

```
SELECT      CAST(p.ProductLine as varchar(max)) AS ProductLine,
            SUM(f.SalesAmount) TotalSalesAmount
FROM [dbo].[FactInternetSales] AS f
INNER JOIN [dbo].[DimProduct] AS p ON
    f.ProductKey = p.ProductKey
GROUP BY CAST(p.ProductLine as varchar(max))
ORDER BY CAST(p.ProductLine as varchar(max));

<Warnings>
<PlanAffectingConvert
  ConvertIssue="Cardinality Estimate"
  Expression="CONVERT(varchar(max), [p].[ProductLine],0)" />
</Warnings>
```

NoJoinPredicate Warning

- Warning of missing join predicates

```
SELECT      p.ProductLine,
            f.SalesAmount
FROM [dbo].[FactInternetSales] AS f,
      [dbo].[DimProduct] AS p
ORDER BY p.ProductLine;
```



```
<Warnings NoJoinPredicate="true" />
```

ColumnsWithNoStatistics Warning

- Missing column statistics
 - Check for disabled auto-create stats
 - Missing index / statistics

```
Table Scan
[DimProduct] [p]
Cost: 0 %
```

```
<Warnings>
  <ColumnsWithNoStatistics>
    <ColumnReference Database="[BigFactTable]" Schema="[dbo]" Table="[DimProduct]"
      Alias="[p]" Column="ProductKey" />
    <ColumnReference Database="[BigFactTable]" Schema="[dbo]" Table="[DimProduct]"
      Alias="[p]" Column="ProductLine" />
  </ColumnsWithNoStatistics>
</Warnings>
```

UnmatchedIndexes

- Identifies that a *filtered index* was not a match for the query due to parameterization

```
CREATE INDEX fi_PromotionKey_by_CurrencyKey
ON dbo.FactInternetSales (PromotionKey)
WHERE CurrencyKey = 6;

DECLARE @CurrencyKey int = 6;

SELECT DISTINCT PromotionKey
FROM [dbo].[FactInternetSales]
WHERE CurrencyKey = @CurrencyKey;

<UnmatchedIndexes>
  <Parameterization>
    <Object Database="[BigFactTable]" Schema="[dbo]"
      Table="[FactInternetSales]"
      Index="[fi_PromotionKey_by_CurrencyKey]" />
    </Parameterization>
  </UnmatchedIndexes>
```

Parameter Sniffing

- The plan contains valuable information on compiled vs. runtime value
 - Parameter sniffing can be good OR bad
 - If parameter sniffing is an issue, validate:
 - ParameterCompiledValue
 - ParameterRuntimeValue
 - Then test cold-cache or recompiled versions using separate values to see if optimal and sub-optimal plans are seen between the two executions

Summary: the “Watch List”

- High estimated cost within a query
- Unexpected join selection
- Scans (Index/Table)
- Sort operations (Query Optimizer added or not)
- Missing indexes
- Cardinality estimate issues
- Hint or plan guide usage (forcing specific plan operations)
- Parallelism and thread skew
- Warnings
- “Thick” lines and late filters
- Lookups
- Aggregation methods (hash vs. stream)
- Parameter sniffing
- Late filtering or residual predicates
- Implicit data-type conversions
- Stop-and-go operators

Resources

- **Whitepaper:**
 - “Statistics Used by the Query Optimizer in Microsoft SQL Server 2008” (Hanson, Angelov)
- **Books:**
 - Microsoft® SQL Server® 2008 Internals (Chapter 8)
- **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 Blog (http://sqlblog.com/blogs/paul_white/)
 - Benjamin Nevarez (<http://www.benjaminnevarez.com/>)

Review

- Why look at query plans?
- How to capture query plans
- How to analyze query plans
- Understanding common operators
- Common query plan patterns to watch for and investigate further

Questions?