

Module 7

Query Plan Analysis



Overview

- 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

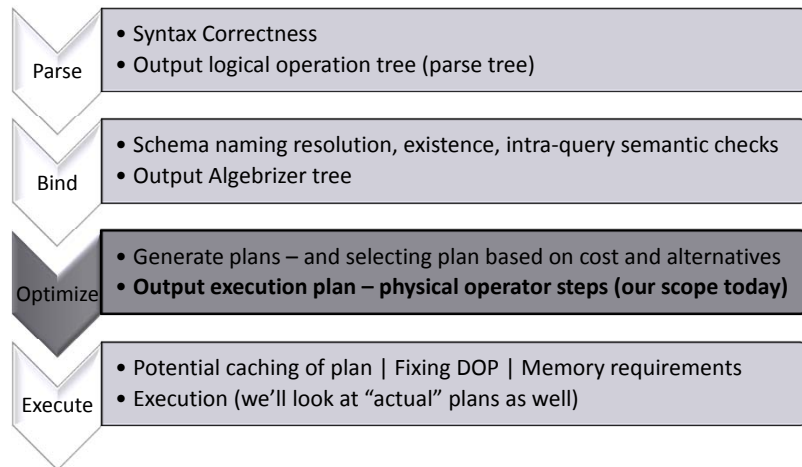
Why do we care about QPs?

- T-SQL is a declarative language; you tell it WHAT to do but not HOW to do it
 - There are exceptions, e.g. hints, plan guides
- 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 QP 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* versus 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 & 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
 - We'll dig in to examples later

©SQLskills.com
SQLskills Immersion Event

7

SQL

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 (for example – ad hoc query, stored procedure, prepared statement)
- `sys.dm_exec_query_plan`
 - Provided a `plan_handle` – 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 specific statements within a batch
 - `statement_start_offset` (in bytes)
 - `statement_end_offset` (in bytes)

Covering in more depth in later modules...

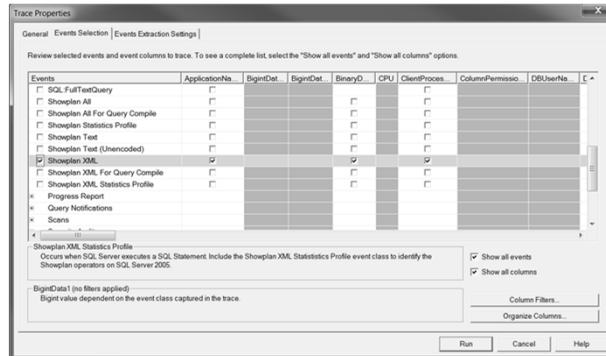
©SQLskills.com
SQLskills Immersion Event

8

SQL

SQL Trace

- Plans available via SQL Trace – **but be careful** of the overhead! This is *not* a preferred method
 - Example of the CPU pegging in production



©SQLskills.com
SQLskills Immersion Event

9

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

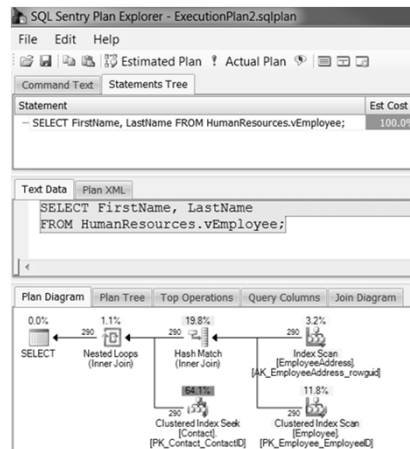


©SQLskills.com
SQLskills Immersion Event

10

Third Party Options

- SQL Sentry Plan Explorer (free tool)
- Good for plans small to very large (compacts data)
- Pulls out key information in compact way



©SQLskills.com
SQLskills Immersion Event

11

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
 - Ideally the row estimates are accurate – but when they aren't, this is one of the red flags we'll discuss how to identify later
- Actual plan
 - Includes estimated AND actual number of rows by operator and executions by operators

©SQLskills.com
SQLskills Immersion Event

12

XML Plans and Showplan Schema

- XML Schema describes the structure of an XML document
<http://schemas.microsoft.com/sqlserver/2004/07/showplan/>
- Allows us to understand what to expect within the XML of a plan
- Can be used as a key for “parsing” via XQuery
- I’ll call out various XML nodes where appropriate, but for more detail I’ve included the schema in the Appendix of this presentation
- 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

Demo

Comparing Showplan Methods

Building Block of the QP

- The query plan is a tree of **operators**
 - Also called **iterators**
- SQL Server 2008 R2 – 100+ operators (logical and physical)
- Think of them as building blocks for a query – each implementing its own functionality
- The tree of operators *is* your query plan
- Specific operators are not “good” or “bad”
 - Some consume more resources than other or have overhead that you should be cognizant of
 - Some are more appropriate in given contexts
 - We’ll discuss several types of operators later

Logical vs. Physical

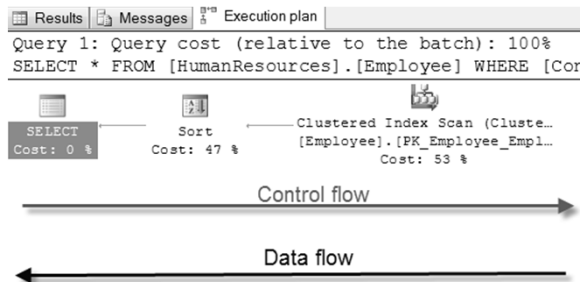
- Logical vs. physical operators
 - One-to-many mapping of logical to physical
 - As in – a logical operator could be implemented via QO rules by multiple physical operators
 - INNER JOIN -> Loop/Hash/Merge
 - Some cases – one physical operator can map to multiple logical operators
 - Often logical op name = physical op name
 - Physical operators implement the operation (for example – Nested Loop Join)
 - You can see L-P mapping in the plan (for example – Tooltip in SSMS over an operator)

Operator underlying 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
- Control flows from the root to the leaves

Query Tree

- Operators are combined into trees
- Operator reads rows from data source OR from child operators and return rows to parent
- Control flow starts at root (L2R)
- Data flow starts at leaf level (R2L)



Key concept – 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)

Key concept – Operator Cost ⁽²⁾

- Operator cost =
 - I/O cost + CPU cost
 - I/O cost assumes physical I/O required (cold cache)
- Cost calculation varies by operator
 - Some have I/O and CPU costs
 - Some just CPU costs
 - # of executes 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

Key concept – 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

Edge Case – 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 bug on how graphical Showplan renders it
 - Sometimes it's a bug in costing itself (bad calculations) – see 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 – more on this later

Query Memory (2)

- One area on the “watch list” are heavy memory consuming operators (more on them later)
 - Hash Match
 - Sort
- When available memory is insufficient, high memory requiring queries may wait to execute
- Underestimating memory (due to CE issues) can cause spills to tempdb (I/O)
- I’ll point this out later in the presentation as we go through different plans

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 – 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?



Key Lookup



- AKA “Bookmark Lookups”
 - SQL Server 2000 -> Bookmark Lookup
 - Early versions of 2005 – “Clustered Index Seek” + keyword LOOKUP
 - 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 – 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 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

SQL
skills

Join Considerations

- Beware of advice telling you that specific join types (or operators, for that matter) are “good” or “bad”
- Key factors:
 - Tables joined, order of joins, algorithm used, cardinality, selectivity
- Join hints and/or forcing order = red flag
 - Generally, “edge” cases warrant their use
 - Otherwise, ask questions and find out why this is happening

Outer/Inner terminology

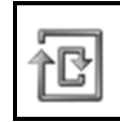
- When it comes to joins, you may hear the “outer” versus “inner” table terminology
 - But its NOT related to the order you write them in your query – unless you’re forcing it
 - Outer = top = left
 - Nested Loop – for each row in outer – find all rows in inner
 - Merge Join – inner/outer – not as important (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
 - Memory requirements are lower compared to other join types
 - NLs be associated with inflated random I/Os when the row estimates end up being way off

Demo

Nested Loop



Merge Join



- Logical operations:
 - Requires one equijoin predicate
 - Except for full outer joins
- Joins two inputs (sorted on joining columns)
 - Pre-existing sorting (via index) is ideal, but sorts can be automatically added (red flag)
- 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)
 - Many-to-Many joins have overhead (worktables)
 - Look for ManyToMany **attribute**
 - Limited parallelism capability compared to Hash Joins
 - Outer (top) / inner (bottom) requires sort on join key
 - If the sort is “injected” – take note of it
 - Injected sorts have a risk of spilling to disk (tempdb)

Demo

Merge

Hash Match Join



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

Hash Match Join



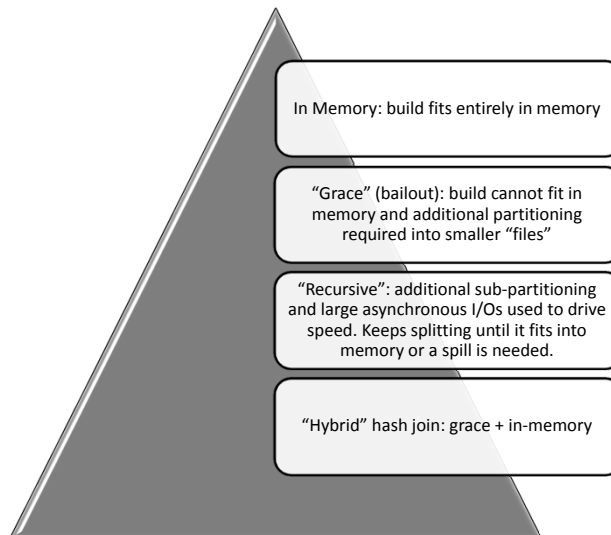
- Algorithm:
 - Build a hash table (hash buckets) via computed hash key values for each row of the “build” input (top/outer table)
 - For each probe row (bottom/inner table), compute a hash key value and evaluate for matches in the “build” hash table (buckets)
 - Output matches (or output based on logical operation)
- Effectiveness of hash bucket distribution depends on hash key and hash function
 - Too many or too few buckets = non-optimal
 - Function choice varies & is proprietary

Demo

Hash Match



Hash Join – Types



Hash Joins – performance variations

- SQL Server can also do “role reversal”
 - $\geq$ one spill, build/probe roles can be switched
- Hash bailouts (grace or recursive) can be found in Hash Warning events (trace)
- Reasons included skewed data distributions, cardinality estimate issues, missing or stale statistics, inappropriate join selection or memory pressure issues
 - Estimates based on build cardinality and average row size

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 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 reqs)
 - Duplicates can cause overhead – impacting bucket size (cannot be further divided)

Hash Join Trees

- Left Deep
 - Hash join output is build input for parent
 - Stop-and-Go puts hold on the parent operator build until the child build ends and child probe begins
- Right Deep
 - Hash join output is probe input for parent
 - Joins not held up by stop-and-go (builds run independently) and probes can start operating
- Bushy (rare in SQL Server – but you can force)
 - Children of a join are both themselves joins

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
- Operators that can keep streaming one row for every row that is read – examples:
 - Nested loop
 - Compute scalar

Filter



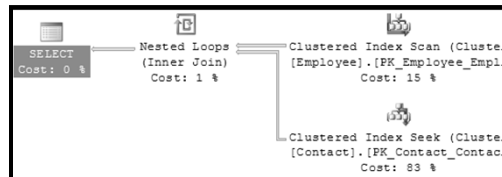
- Predicates can be evaluated *within* operators that read data from table/indexes
- QO aims (when possible) to place filter down the tree (leaf level) to reduce rows moved
- If predicate is high in cost or complexity a separate Filter operator may be used
- When you see these – take note of where they 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 (hence residual)
 - 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 – 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];
```



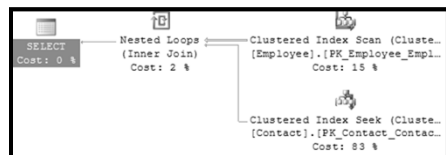
Clustered Index Seek (Clustered)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	290
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	290
Estimated Number of 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])	

©SQLskills.com
SQLskills Immersion Event

53

Seek Predicate + 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 = '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.com
SQLskills Immersion Event

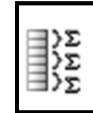
54

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 – QO 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 aggregates – duplicates are not an issue (are not retained and are de-duped into individual groups)

Sort



- As named – orders rows received from input
- Variation – “Distinct Sort” to remove duplicates
- 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

Eager \ Lazy Spool



- 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

Row Count Spool



- Scans input
 - Counts rows
 - Returns count without associated data
- Used for row “existence” checks
 - E.g. -> a left semi join operation (only outer rows needed, but inner rows evaluated in predicate)

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 re-updates to
- 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 used – like Sort

Rebinds and Rewinds

- INNER and OUTER table – one row at a time from OUTER table
- Parameter value from outer row to lookup from inner row
- Rewind = row in outer table maintains the same value as previous lookup in inner table (less expensive)
- Rebind = lookup value changes from outer table (more expensive cost estimate)
- Sort / Table Spool / NCI Spool / Row Count Spool TVF / Assert (if Startup Express is true) / Filter (if Startup Expression is true)

Spool overhead

- Might be an optimization when used
- 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

Demo

Spools

SQL
skills

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)

Assert



- Verifies existence of a specific condition
 - Check constraints
 - Referential integrity
 - Enforce return of one-row for scalar subquery
 - 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



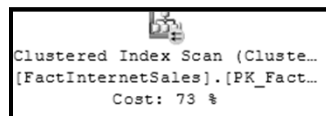
- Evaluates an expression, producing a scalar value (e.g. GETDATE() value)
 - Also seen with partitioning
- SET STATISTICS XML “gotcha”
 - May not have RunTimeInformation element – missing Actual Rows / Rebinds / Rewinds

Parallelism

- Query Optimization can generate both a serial and parallel plan
 - But QO only caches *one* of them – not both
 - 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 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
- 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

Common Distribution Algorithms

- Applies to Distribute/Repartition (not all types listed):
 - <Parallelism PartitioningType="Hash">
 - Hash function on >=1 column determines thread routing (push)
 - <Parallelism PartitioningType="Broadcast">
 - All rows to all threads (push)
 - <Parallelism PartitioningType="RoundRobin">
 - Sequential send of row packets to next thread (push)

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
- Watch for memory pressure (increased for parallel operations)

Demo

Parallel Plan



DW Optimizations Bitmap

➤ 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

➤ StarJoinInfo

- If StarJoinInfo element exists, SQL Server has identified the plan as a star join plan
 - QO can optimize based on restrictive dimensions and very large fact table producing a seek on the fact table

Merge Interval



- Merges multiple potentially overlapping intervals used in predicates
 - Goal is to minimize redundant scans
- If you see this operator – may be a tipoff to expressions with unknown values at compile time
- Preceded by compute scalar and constant scan

Data Modifications - Narrow Plans

- 1 row update of index at a time
- One operator, and then other dependent indexes handled behind the scenes (almost)
- Can have overhead if many rows, as row-by-row done based on clustered key order (random I/O)

Data Modifications - Wide Plans

- Wide plans: 1 index at a time gets updated at a time (multiple rows)
 - 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
 - [10/11/12/13] + 1
 - Delete 10 and Insert 14

Demo

Data Modification Plans

Cursors – plan and operators

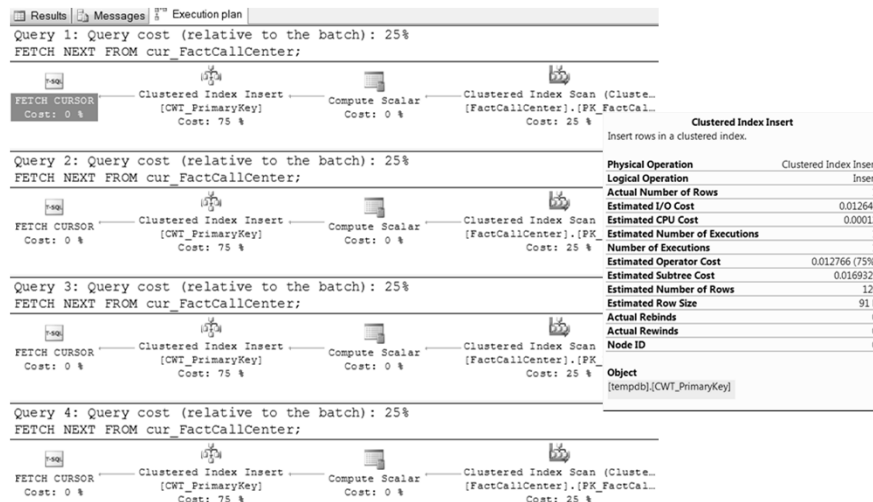
- Dynamic Cursor - reflects all data changes made to the rows as you scroll

```
Cursor example.sql...JosephSack (52))
USE AdventureWorksDW2008R2;
GO
DECLARE cur_FactCallCenter CURSOR DYNAMIC
FOR SELECT * FROM dbo.FactCallCenter
OPEN cur_FactCallCenter

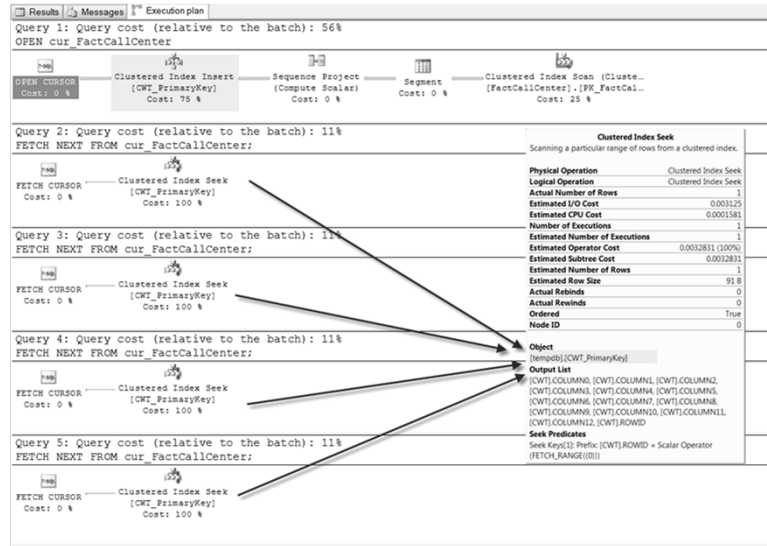
FETCH NEXT FROM cur_FactCallCenter;
FETCH NEXT FROM cur_FactCallCenter;
FETCH NEXT FROM cur_FactCallCenter;
FETCH NEXT FROM cur_FactCallCenter;

CLOSE cur_FactCallCenter;
DEALLOCATE cur_FactCallCenter;
```

Dynamic cursor - plan shape



Static cursor - plan shape



©SQLskills.com
SQLskills Immersion Event

83

Cardinality Estimate issues

- Major red flag to watch for
 - Skewed estimate vs. actual
- Can magnify and distort as it moves 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 (not predictable)

©SQLskills.com
SQLskills Immersion Event

84

Cardinality Estimate Next Steps

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

Implicit Data Type Conversions

- Explicit = CONVERT or CAST
- Implicit data type conversion – e.g. joining two table columns with different data types
 - Data type with higher precedence “wins” and is converted
- Will 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 other clues may exist)

Parameter Sniffing

- The plan contains valuable information on compiled versus 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

SQL Server 2012 Changes

- Spill warning information
- Query wait information (Memory Grant)
- Execution mode type for columnstore indexes (batch/row)
- Runtime memory grant information
- Plan-affecting type conversion warnings
- Optimizer Hardware Dependent Properties type for available memory grant, pages cached, available DOP (not documented yet)

Summary - the “watch list”

- High cost across queries
- High cost within a query
- Odd join selection
- Scans (Index/Table)
- Sort operations (injected or not)
- Missing Indexes
- Cardinality estimate issues (compiled vs. runtime values with significant plan differences)
- 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
- Cursors (evaluate if SET-based is appropriate)

©SQLskills.com
SQLskills Immersion Event

89

SQL

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

©SQLskills.com
SQLskills Immersion Event

90

SQL

Resources

- **Whitepapers:**
 - "Statistics Used by the Query Optimizer in Microsoft SQL Server 2008" (Hanson, Angelov)
 - "Hash joins and hash teams in Microsoft SQL Server" (Graefe, Bunker, Cooper)
- **Books:**
 - Microsoft® SQL Server® 2008 Internals (Chapter 8)
 - SQL Server Execution Plans, Grant Fritchey
- **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/>