SQL intersection

Mastering SQL Server Execution Plan Analysis

Preconference Workshop #17

SQLintersection Fall 2014

Workshop: Sunday, November 9, 2014

written and presented by
Paul White

SQLintersection

Pre-Conference Workshop

Mastering Execution Plan Analysis

Paul White
SQLkiwi@gmail.com

SQL intersection

Paul White

Page Free Space
SQLblog.com

SQLperformance.com

Database Administrators
dba.stackexchange.com

@SQL_Kiwi

SQLkiwi@gmail.com

SQL intersection

2

Outline

- **The Query Optimizer**
  - Overview
  - Detailed architecture and behaviors
  - Strengths and weaknesses
  - Designing and coding for best results
- **Execution Plan Analysis**
  - Execution engine architecture
  - Basic plan analysis
  - Query plan operators and properties
  - Advanced plan analysis
    - Common patterns
    - Changing data

SQL intersection

Mastering Execution Plan Analysis

The SQL Server Query Optimizer

SQL intersection

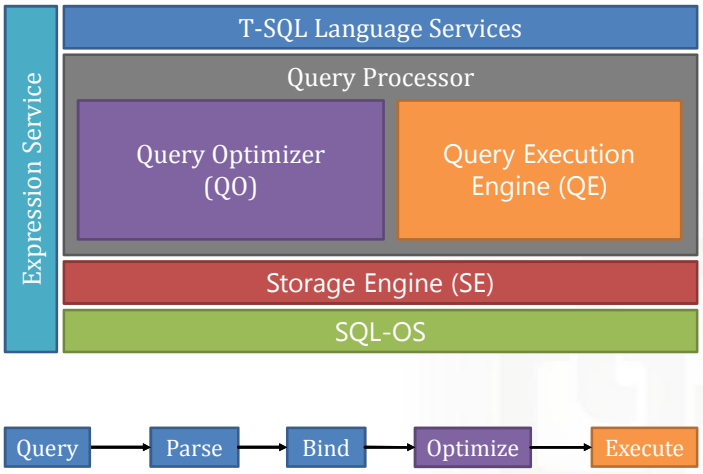
Concepts

- **SQL**
  - Origins in set theory & relational algebra
  - Bags not sets and other quirks
  - Declarative not procedural
  - Binding order
- **Execution Engines**
  - Row mode: a demand-driven pipeline of iterators
  - Batch mode execution available from SQL Server 2012
  - In-memory OLTP engine from SQL Server 2014
- **Query Optimizer**
  - The bridge between SQL and the execution engines
  - Same logical results but greater efficiency
  - Different goals than optimizing compilers in other languages

```
FROM
ON
JOIN
WHERE
GROUP BY
CUBE / ROLLUP
HAVING
SELECT
DISTINCT
ORDER BY
TOP
```

SQL intersection

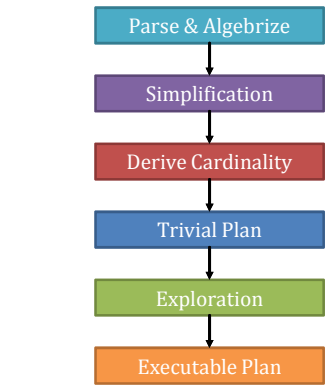
SQL Server Architecture



SQL intersection

6

Query Optimization Pipeline



More Detail

Parse
Algebrize
Expand views
NNF convert
Simplification
Automatic statistics processing
Parameterization
Cardinality derivation
Project normalization
Trivial plan
Cost-based exploration
Convert to executable plan

Parse & Algebrize

- **Parse**
  - Check syntax and semantics
  - Covert to a "parse tree" of logical operations
  - Identify constants for server-side parameterization
- **Algebrize**
  - Name resolution
  - Bind object references to metadata
  - Derive expression types
  - Bind aggregates to grouping expressions
  - Constant folding
  - Parameter embedding
  - Output is a bound tree of logical operators
  - Bound trees may be cached for e.g. views, constraints, and defaults

Constant Folding & Parameter Embedding

- **Constant folding**
  - Early evaluation of constant expressions to improve performance
  - Arithmetic & logical expressions
  - Some *deterministic* built-in functions
  - Additional cases for cardinality estimation only
- **Parameter embedding**
  - Requires OPTION (RECOMPILE)
  - Not WITH RECOMPILE
  - From SQL Server 2008 SP1 CU5 (build 10.00.2746) onward
  - Parameter values and some T-SQL expressions fully evaluated
  - Significant tree rewrites are possible
  - Occurs before cost-based optimization
  - Not applied when assigning to a variable; other minor exceptions

SQL intersection

Demo

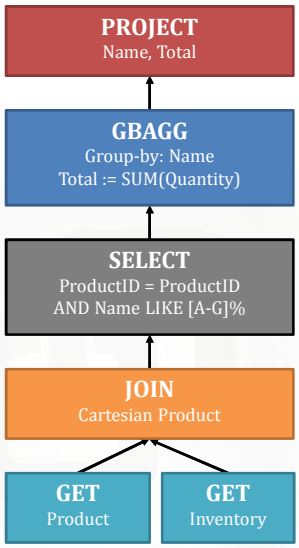
Demo Q1
Constant Folding & Parameter Embedding

SQL intersection

Bound Logical Tree

```
SELECT
    P.Name,
    Total = SUM(INV.Quantity)
FROM Production.Product AS P
JOIN Production.ProductInventory AS INV
    ON INV.ProductID = P.ProductID
WHERE
    P.Name LIKE N'[A-G]%'
GROUP BY
    P.Name;
```

Bound Logical Tree



SQL intersection

11

Demo

Demo Q2

Bound Logical Tree

SQL intersection

The Simplification Phase

- **Predicate & projection pushdown**
  - Eliminate rows and columns as early as possible
  - Index & computed column matching
  - Automatic statistics generation
- **Rewrite logical subqueries**
  - As inner join
  - As outer join (with group by and projection as necessary)
  - As an apply
- **Simplify joins**
  - Full and outer join to inner join
  - Remove redundancies
- **Remove empty expressions**
- **Originally aimed at generated T-SQL**
  - All or nothing & performed only once

SQL intersection

Demo

Demo Q3

The Simplification Phase

SQL intersection

Cardinality Estimation

- **Estimate the size of intermediate & final results**
- **Inputs**
  - Base relation cardinality information
  - Statistics histograms & density
  - Trie trees for short strings
- **Derived statistics**
  - Histogram & density recomputed after each logical operation
  - New alternatives may generate new operations
  - No general guarantee of consistency
  - Many modelling assumptions
- **Cardinality estimates influence initial join order heuristics**
  - In all but the simplest of cases

CE Modelling Assumptions

- **Uniformity**
  - Distinct values have equal frequency
  - Values are distributed evenly
- **Independence**
  - Column values are considered to be independent
  - Unless correlation information is available
  - Very limited multi-column statistical support
- **Inclusion**
  - Comparisons always yield a match
  - Users look for data because it exists
- **Containment**
  - The smaller number of distinct values all exist in the larger set
  - Base containment – only applies to base tables

CE Model Support

- **Specific model support for:**
  - Filter
  - Joins
  - Apply
  - Project
  - Group By
  - Union
  - Top
- **For best results**
  - Use simpler relational constructions
  - Avoid complex expressions and opaque operations
- **Not everything can be modelled**
- **Not everything that could be modelled has been**

SQL Server 2014 CE

- **New cardinality estimation module**
  - Determined by *context database* compatibility level
  - Trace flag 2312 new CE override
  - Trace flag 9481 original CE override
- **Major features**
  - Main goals are simplicity, consistency, and supportability
  - Groundwork for possible future improvements
  - Exponential backoff
  - Ascending key and general out-of-histogram logic
- **Expectations**
  - Likely overall improvement for most workloads
  - Regressions *will* occur
  - Real testing is required

Some Useful CE Trace Flags

- **Original CE**
  - 2301 Enable advanced modelling extensions
  - 2389 & 2390 The ascending key problem
  - 4137 Minimum selectivity (for conjunctive predicates only)
  - 4138 Disable row goals
  - 4139 Apply ascending key logic even when stationary
- **New CE**
  - 9471 Minimum selectivity for conjunctive *and* disjunctive predicates
  - 9472 Independence instead of exponential backoff
  - 2363 Debug output
  - Many other undocumented model variations in the 94xx range
  - Only *documented* flags are supported for production use

Demo

Demo Q4
Cardinality Estimation

The Trivial Plan Stage

- **For queries with no cost-based choices**
  - A fast path to avoid the overhead of cost-based optimization
  - Details change over time
  - Occasionally chosen even if plan choices exist
- **Things that may prevent a trivial plan**
  - Subqueries and joins
  - Some query hints
  - Cost threshold for parallelism
- **If a trivial plan is chosen:**
  - Simple parameterization evaluated for safety
  - Query optimization may end here

Demo

Demo Q5
The Trivial Plan Stage

The Cost-Based Optimizer

- **Uses the Cascades framework**
  - Many public research papers
- **Initialization**
  - Copy logical tree nodes into a "memo" structure
  - Initially one memo group per tree node
- **Operation**
  - Explore logical alternatives
  - Implement using physical operations
  - Calculate expected costs
  - Rinse & repeat
  - Choose the cheapest combination
- **Goals**
  - Find a reasonable plan quickly
  - Not an exhaustive search

Demo

Demo Q6

Cost-Based Optimizer Input

Memo Groups

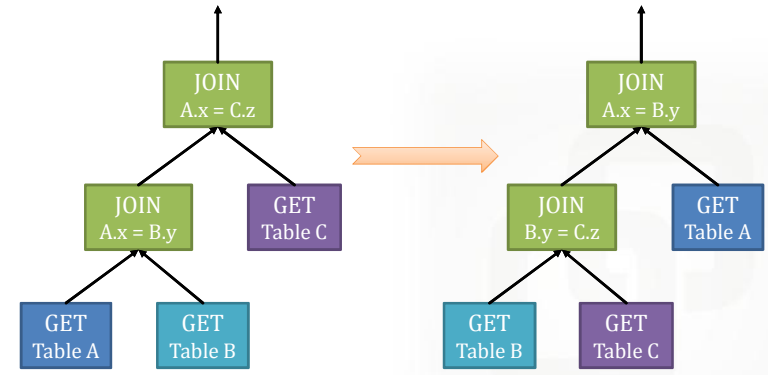
- **Logical properties**
  - Input & output column list
  - Data types and nullability
  - Keys & functional dependencies
  - Column value ranges
  - *Constant per memo group*
- **Physical properties**
  - Sort order
  - Cardinality estimate
  - Parallelism
  - Execution mode
  - Halloween protection
  - *Vary per memo group*

Four Types of Optimizer Rules

- **Simplification**
- **Exploration**
  - Generate new logical alternatives
  - For a single group or a subtree
  - Mostly relatively simple operations
  - Explorations may combine to produce more complex effects
- **Implementation**
  - Physical implementation of a logical operation
  - Generate estimated operator cost
- **Enforcers**
  - Required sort orders
  - Execution modes
- **395 rules in SQL Server 2014**

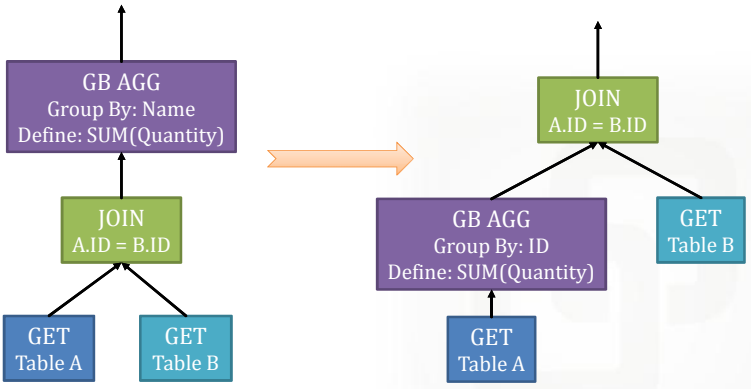
Exploration Rule Example 1

- **Inner Join Switch**



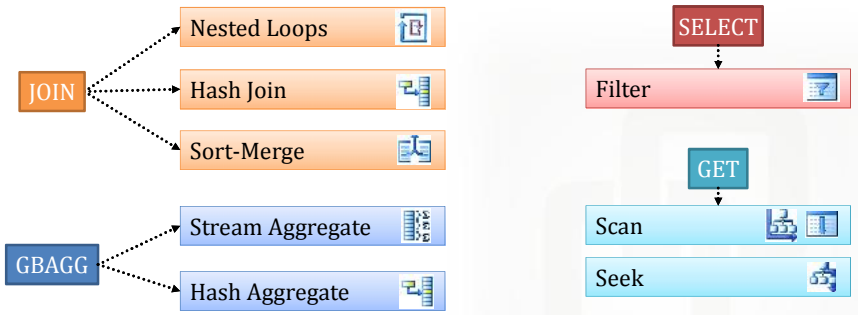
Exploration Rule Example 2

- **Push Aggregate Below Join (GbAggBeforeJoin)**



Implementation Rules

- **Logical to physical**



Disabling Rules

- **Join hints**
  - LOOP
  - HASH
  - MERGE
- **Query hints**
  - HASH | ORDER GROUP
  - CONCAT | HASH | MERGE UNION
  - FORCE ORDER
- **Undocumented**
  - DBCC RULEON | RULEOFF | SHOWRULES
  - OPTION (QUERYRULEOFF)

Demo

Demo Q7
Optimizer Rules

SQL intersection

Cost-Based Optimization Phases

- **Search 0 – Transaction Processing**
  - Minimum of 3 table references
  - Primarily indexed nested loops join
  - Limited set of exploration rules available
- **Search 1 – Quick Plan**
  - More explorations available
  - May be run a second time with parallelism required
- **Search 2 – Full Optimization**
  - Most comprehensive search
  - Minimum of 5 table references
  - Serial or parallel as decided in search 1
  - Entered directly when USE PLAN hint specified

SQL intersection

Operator Costing

- **Inputs**
  - Cardinality & distribution information
  - Sequential & random I/O
  - CPU worker time
  - Memory
  - Parallelism
- **Many Model Assumptions**
  - Cold cache
  - Indexed nested loops inner side seeks randomly distributed
  - Largely independent of available hardware
- **The Numbers**
  - Little relation to modern realities
  - Estimated cost and batch cost percentage

Plan Freezing

- **Plan Guides**
  - May not reduce compilation time
  - XML is verbose and expensive to parse
  - Plan shape guides the search space
  - Replaces cost-based pruning & discarding heuristics
  - May time out
  - Expressions not considered
  - Filter position cannot be forced
  - Always runs search 2

Demo

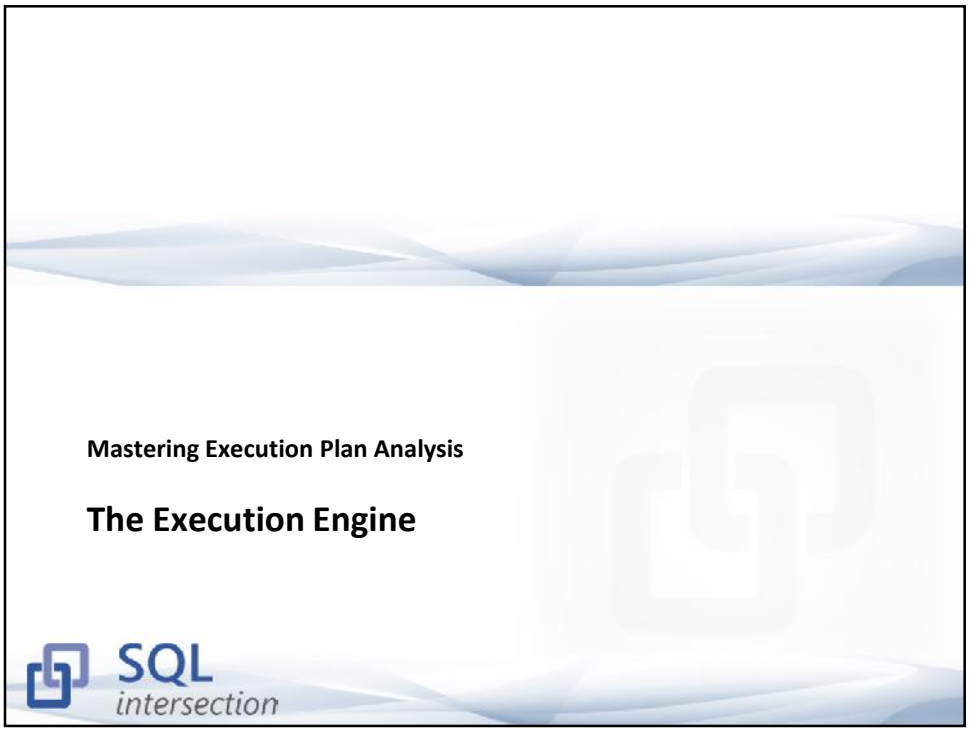
Demo Q8
The Memo and Optimization Phases

For Best Results…

- Be *simple, clear, and relational*
- Be explicit about types
- Define and enforce constraints
- Create filtered, multi-column, and computed statistics as necessary
- Keep statistics up to date
- Think like the optimizer but also consider the whole plan
- Use your knowledge of the data and business process
- Remember new features often have limited optimizer support
- Avoid deep trees where cardinality estimates degrade quickly
- Consider intermediate materialization
- Use hints if necessary, and review regularly
- Learn about the execution engine…

SQL intersection

45



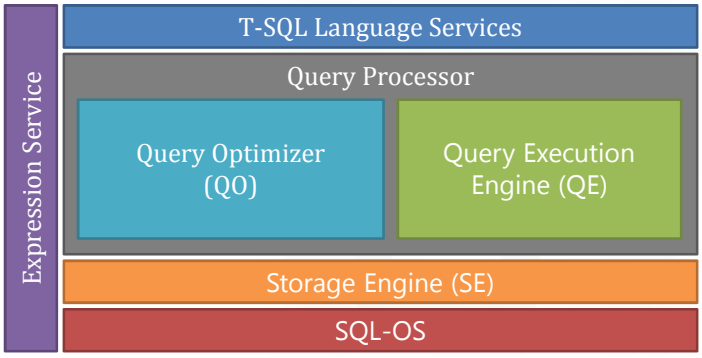
Introduction

- **Reading Execution Plans – The Basics**
  - Scans & lookups
  - Thick lines
  - Actual versus estimated cardinality
  - Expensive queries in the batch
  - Expensive operators in the plan
- **Execution Plan Analysis – Advanced Query Tuning**
  - Requires deeper knowledge of:
  - The execution engine
  - Execution plan operator behaviours and properties
  - Desirable & undesirable query plan shapes

SQL intersection

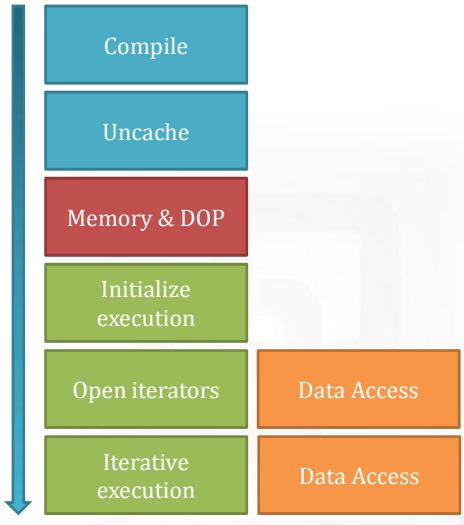
The Query Execution Engine

- Operates on plans produced by the Query Optimizer
- Stores and retrieves data via the Storage Engine
- SQL-OS provides memory and threading services



Query Execution Overview

- Compile if necessary
- Retrieve from plan cache
- Acquire memory grant
- Fix degree of parallelism
- Initialize execution
- Open plan iterators
- Iterative execution
  - New modes 2012+



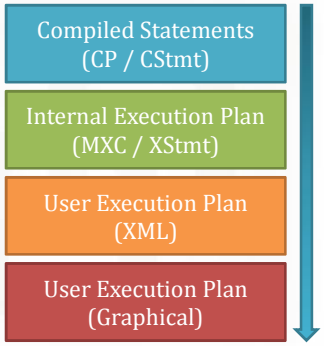
Execution Plan Architecture

- **QO produces one cached plan (CP) per *batch***
  - May contain many compiled statements (CStmts)
  - An execution context (MXC) is generated from the CP
  - CStmts converted to executable statements (XStmts) within the MXC
- **The MXC contains runtime information**
  - Parameter & local variable values
  - IDs of objects created at runtime
  - State information (e.g. currently executing XStmt)
  - Expression bindings
  - And much more
- **The MXC is referred to internally as the "execution plan"**

SQL intersection

The Internal Execution Plan

- **The MXC is not the same "execution plan" users see**
  - XML show plan is derived from the MXC
  - Not all iterators are incorporated in the XML
  - Not all internal details are exposed
  - Graphical plans are another abstraction on top



SQL intersection

Iterative Execution

- **Execution Plan**
  - A demand-driven pipeline of generic iterators
- **Generic iterators**
  - Perform a single physical data operation
  - Common interface of Open, Get Row, and Close methods
  - Combined into trees we call execution plans
- **Demand-driven**
  - Execution begins at the extreme left
  - Consumers drive producers to do work
  - Data moves from right to left as a result
  - Execution plans suck
- **Pipeline**
  - Iterators *consume* one row at a time

SQL intersection

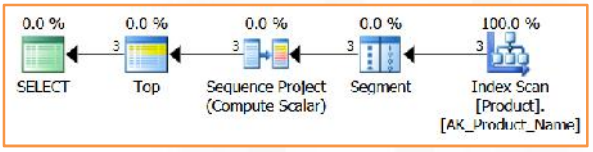
Internal & User Execution Plans Illustrated

1 Internal Execution Plan (MXC / XStmt)

CQScanRowsetNew
CQScanRangeNew
CQScanProfileNew
CQScanSegmentNew
CQScanProfileNew
CQScanSeqProjectNew
CQScanProfileNew
CQScanTopNew

2 User Execution Plan (XML / Graphical)

```
<?xml version="1.0" encoding="utf-16"?>
<ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="3" Statem
          <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true
          <QueryPlan DegreeOfParallelism="1" CachedPlanSize="16" Co
```




```
SELECT TOP (3)
    RowNum = ROW_NUMBER() OVER (ORDER BY P.Name),
    P.Name
FROM Production.Product AS P
ORDER BY RowNum;
```

SQL intersection

Demo

Demo X1
Row mode execution internals

About Data Access in Execution Plans

- **The optimizer and execution engine try to avoid copying data**
  - Preferred data access is “by reference”
  - Non-leaf plan operators may reference row data directly
  - The storage engine keeps track of the current row
  - Don’t think of complete rows being passed around
  - Some operators necessarily make a copy of the data
  - Details not exposed in show plan output

Reading Execution Plans

- **Right to Left**
  - Useful for a high-level overview
  - Shows *summary* information only
  - Be aware of the underlying mechanisms at work
- **Left to Right**
  - Detailed analysis of critical plans
  - Advanced tuning work
  - The only way to understand certain behaviours
- **Combine both techniques**

SQL intersection

Mastering Execution Plan Analysis

Execution Plan Operators

SQL intersection

Operator Types

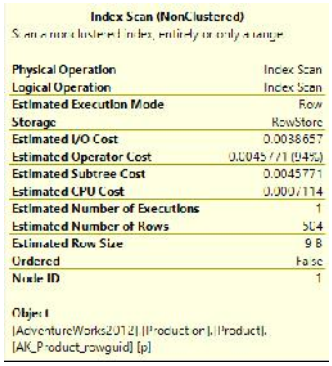
- **Pipelined**
  - Consume input & produce output in GetRow() method calls
- **Blocking**
  - Consume entire input in Open() method call
  - Often require a memory grant or other temporary storage
    - Sort
    - Eager hash aggregate and distinct
    - Hash join (build input only)
    - Eager spool, scalar aggregate, UDX
- **Batched**
  - Parallel column store only

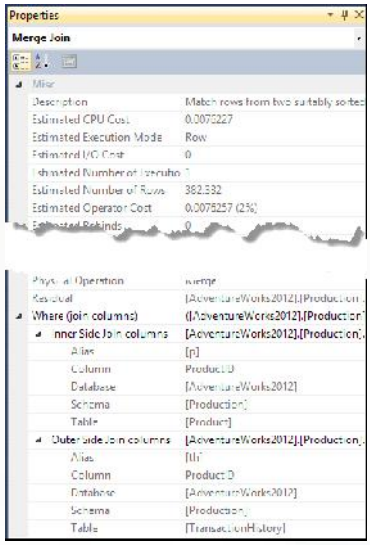
Memory Usage

- **All operators require a small amount of memory**
  - State information
  - Runtime calculations
  - This memory is cached with the plan
  - Other *small* memory needs may be allocated on demand
- **Workspace memory grant**
  - Some operations require significant workspace
    - Primarily sorting and hashing
  - Memory grant fixed just before execution begins
  - Split evenly between threads in row-mode parallel plans
  - Exception for index maintenance sorts

Operator Properties

- **SSMS tooltips**
- **Properties window (F4)**





Scans and Seeks

- **Scans**
  - May be *ordered* or *unordered*
  - Forward or backward
  - Allocation order scan possible if:
    - Unordered
    - At least 64 pages
    - NOLOCK, TABLOCK or read-only data
  - Not always a full scan
- **Seeks**
  - Singleton seek or seek with range scan
  - Always ordered, may be forward or backward
  - Multiple seeking operations possible

Key and RID Lookups

- **Not all required columns available in the nonclustered index**
  - Per-row base table lookup using clustering key or heap RID
- **Always implemented using Nested Loops Join**
  - Check operator output columns *and predicates*
- **May be the optimal choice in some cases**
  - Impossible to fully cover every query; or
  - Limited number of lookups to fetch wide data values
- **Can be very slow if random physical I/O is generated**
  - Depending on the storage subsystem
  - Read-ahead adversely affected in any case

SQL intersection

Demo

Demo X2

Scans, Seeks & Lookups

SQL intersection

Filters and Residuals

- **Filter**
  - A predicate that cannot be used to seek an index
  - May have a *start-up expression*
- **Residuals – Hidden Filters**
  - Filter predicates may be pushed into a child iterator
    - Seeks
    - Scans
    - Joins
    - Lookups
  - May have severe performance consequences
  - Often overlooked

SQL intersection

Demo

Demo X3
Filters and Residuals

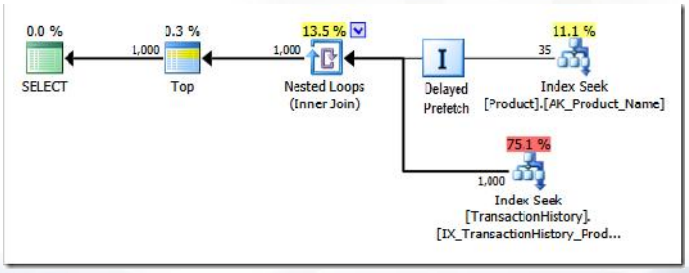
SQL intersection

Nested Loops Join

- **For each outer row, check the inner side for matches**
  - Inner side often correlated
  - Look for "outer references" to confirm
- **Best for:**
  - Small to medium outer input
  - Correlated index seek on inner side
  - May preserve outer side order
  - Very good parallel performance is possible
- **Inner side considerations**
  - Memory grant based on average correlation values
  - Aggregated execution statistics
  - Read-ahead may not be as effective

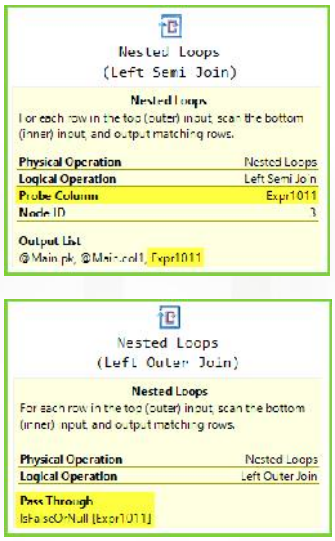
Nested Loops Prefetch

- **Outer side key look-ahead**
- **Asynchronous scatter-gather I/O**
- **Ordered or Unordered Prefetch property on join iterator**
- **Also applies to Key and RID Lookups**
- **Join iterator has an extra outer references expression**
- **Prefetch iterator not exposed in show plan XML**



Probe and Pass Thru Properties

- **Probe**
  - Indicates if an inner side row was found
  - Nested loops and merge join
  - Not shown for merge semi join
- **Pass Thru**
  - Selective inner side execution per row
  - Nested loops join *only*



Demo

Demo X4
Nested Loops Join

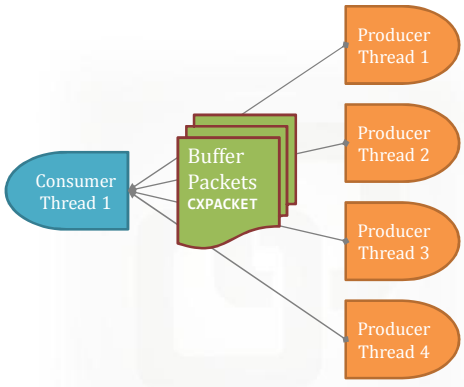
Parallel Plans – Row Mode Execution Model

- **Multiple plan branches bounded by exchanges**

- **Each branch may be serial or parallel**
- **Each parallel branch contains DOP threads**
- **Each thread runs in its own execution context (MXC)**
- **Only MAXDOP threads can be active at the same time**

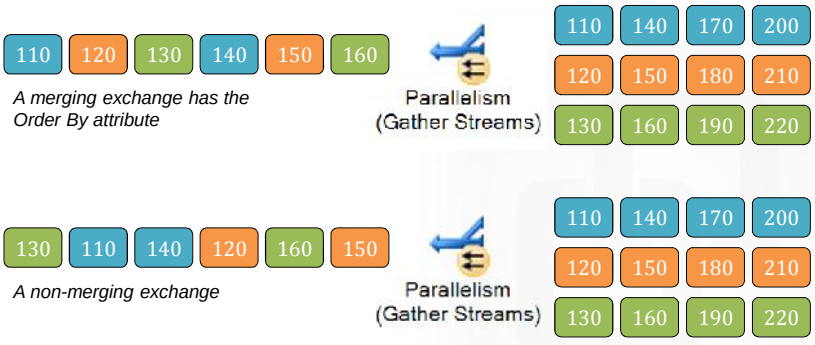
Inside An Exchange

- **Exchanges buffer rows in packets**
- **Two iterators in one**
- **Producer side fills packets**
- **Consumer side reads packets**
- **CXPACKET waits**
  - Consumer: No packet ready
  - Producer: All buffers full



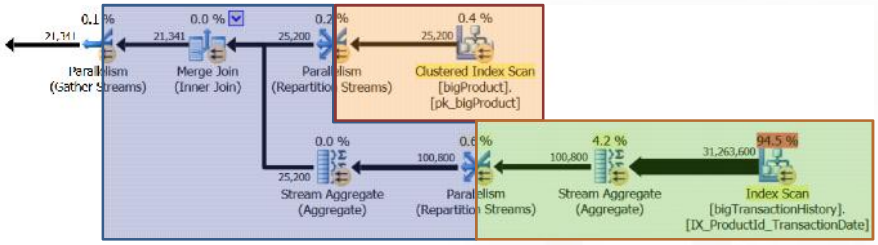
Exchange Order Preservation

- **Exchanges optionally *preserve* sort order**
  - Known as a *merging* exchange



Multiple Parallel Branches

- **Parallel merge join plan from the previous demo**
- **16-24 threads used at DOP 8**
- **Multiple independent demand-driven pipelines**

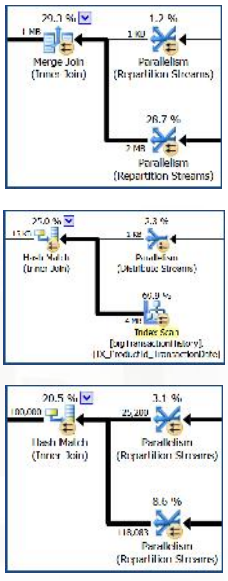


Row Distribution

- **Parallel seek/scan distributes rows across threads on demand**
- **Exchanges also route packets of rows between threads**
  - Strategy identified by the exchange Partitioning Mode:

PARTITIONING MODE	PACKET DISTRIBUTION
Hash	Hash function on partitioning column values
Round Robin	Next consumer in sequence
Broadcast	Copied to all consumers
Demand	Row at a time pulled from the producer
Range	Consumers assigned distinct partitioning column ranges

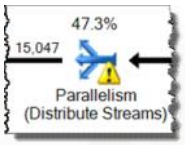
Parallel Joins



- **Merge Join**
  - Both inputs hash partitioned on join keys
  - Merging exchanges required
  - Scaling problems as DOP increases
  - Increased parallel deadlock risk
- **Hash Join**
  - Broadcast partitioning if the build input is small
    - Full hash table copy on each thread
    - No probe-side exchange
  - Hash partitioning on join keys otherwise
    - Two non-merging exchanges
    - Hash table split between threads

Parallel Execution Issues – Row Mode

- **Exchanges can be expensive**
- **Order preservation can reduce efficiency**
- **Parallel deadlocks are possible**
- **Plan costing**
- **TF 8649**
- **Thread skew**
- **Parallelism inhibitors**
  - Scalar functions, modifying a table variable, system table access…
  - TOP, window functions pre-2012, multi-statement T-SQL functions
  - Backward-ordered seeks & scans
- **Inefficient execution model for modern CPU designs**



Batch Mode Parallel Execution

- **Enterprise Edition Only**
- **Column store index access required**
- **Vector processing optimized for on-chip caches**
- **Push mode per segment**
- **Shared lock-free data access removes need for exchanges**
- **Any thread can process the next available batch**
- **Limited utility in SQL Server 2012**
- **Extensive Improvements in SQL Server 2014**
  - Updatable
  - More supported operators
  - Mixed mode plans

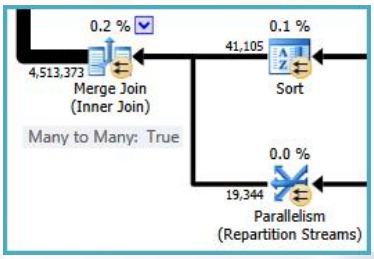
Demo

Demo X5
Parallel Execution

SQL intersection

Merge Join

- **Inputs must be sorted on the join keys**
  - Can be expensive if sorts are needed
- **Equijoin required**
  - Except full outer join (rare)
- **Many-to-many join requires a work table to handle duplicates**
  - Expensive if used
- **Preserves join key order**
- **Stops when *either* input runs out of rows**
- **Parallel merge join disfavoured**
- **Best for medium-sized serial tasks**



SQL intersection

Hash Join

- **Algorithm**
  - Build a hash table on the join keys
  - Probe hash table for matches
  - Apply residual predicates (bucket scan)
  - Requires an equijoin
- **Features**
  - Blocking on the hash table build input
  - Pipelined on the probe input
  - May spill to physical *tempdb* if hash table exceeds memory grant
  - Role reversal possible if a spill occurs
  - Does not preserve order
  - Best for small build input; medium to large probe input

SQL intersection

Bitmaps

- **Parallel plans only**
- **Early semi join reduction**
- **Row Mode**
  - Hash join or merge join with blocking iterator
  - Applied to single probe side exchange, scan, seek, or filter
  - INROW optimization for integer and bigint types
- **Batch Mode**
  - Bitmap created at batch hash table build operator in SQL 2012
  - BHTB no longer needed in 2014
  - Bitmap Creator property
  - Can be split and applied in column store scan access method

SQL intersection

Demo

Demo X6
Bitmap Magic

SQL intersection

Aggregates and Distinct

- **Stream Aggregate**
  - Requires input sorted by the GROUP BY columns
  - Preserves that order
- **Hash Aggregate**
  - Blocking
  - Flow Distinct variant is pipelined
  - Preferred for larger sets with fewer groups
- **Distinct Aggregate e.g. COUNT(DISTINCT x)**
  - Implemented as separate grouping and aggregation operators
- **Sort Distinct**

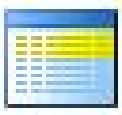
SQL intersection

Demo

Demo X7
Aggregates and Distinct

SQL intersection

Top N



- **Any N rows without an ORDER BY clause**
  - Repeatable results require a deterministic ORDER BY
  - *Only* top-level ORDER BY guarantees result set order
- **Specifying a percentage requires reading the full input set**
- **WITH TIES applies to the final group only**
- **Row goal subtree effects**

SQL intersection

Demo

Demo X8
Top and Row Goals

SQL intersection

Constant Scan & Compute Scalar

- **Constant Scan**
  - An in-memory pseudo-table of expressions or constant values
  - May have no columns
- **Compute Scalar**
  - Defines an expression
  - Evaluation is often deferred
  - May be present for internal reasons
  - Often reveals implementation details
  - Not always very tidy

SQL intersection

Sort

- **Blocking**
- **Spills to physical tempdb if memory grant is exceeded**
- **Can perform a distinct during sorting**
- **Normal algorithm is a variety of mergesort**
  - Separate algorithm for Top N Sort where N <= 100
  - In-memory standard library *quicksort* for Constant Scan input
- **Memory fractions property**
- **SQL Server 2012 show plan enhancements for:**
  - Spills
  - Detailed memory grant information
  - Memory grant wait time

SQL intersection

Demo

Demo X9

Compute Scalar and Sort

SQL intersection

Segment and Sequence Project

- **Segment**
  - Sets a flag for the start of a new group in a sorted stream
  - Window functions
  - Segment spool
  - Segment top
- **Sequence Project**
  - Compute Scalar for ordered streams (a sequence)
  - Adds a computed column e.g. row_number

SQL intersection

Demo

Demo X10

Segment and Sequence Project

SQL intersection

Spools

- **A temporary work table backed by *tempdb***
  - May not cause any physical I/O
  - Eager or lazy accumulation of rows
- **Lazy Table Spool**
  - Often seen on the inner side of a nested loops join
  - Work table truncates on rebind
  - Performance spool combines this with an outer side sort
- **Eager Index Spool**
  - Seek predicate identifies the index keys
  - Always built on a single thread

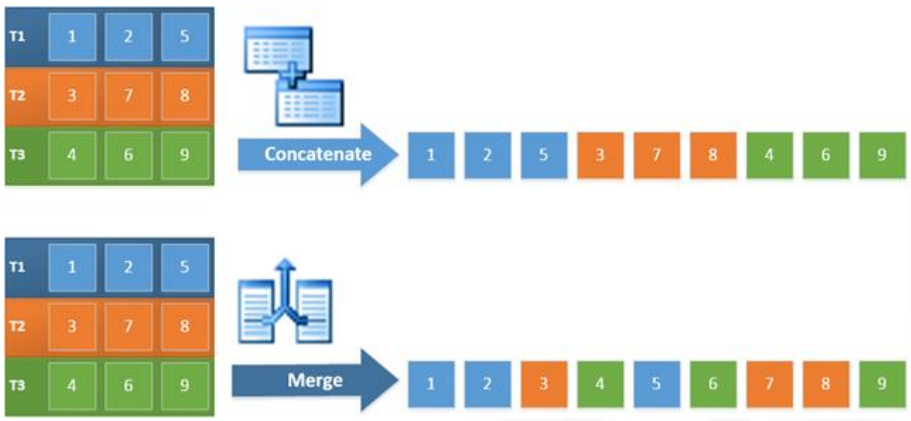
Demo

Demo X11
Eager Index Spool

Union

- **Union All**
  - Preserves duplicates
  - Concatenation or Merge Join iterators
- **Union**
  - Removes duplicates
  - Merge or Hash iterator
  - Concatenation with following distinct
- **MERGE / CONCAT UNION hint**
  - Only effective for UNION ALL from SQL Server 2012 onward

Union All Ordering



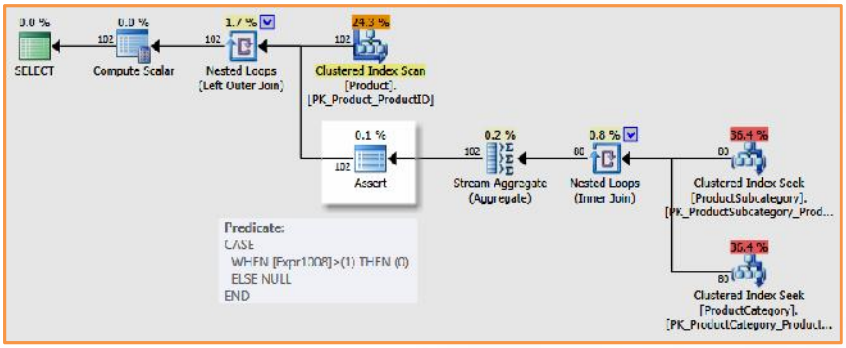
Demo

Demo X12
Preserving Order

SQL intersection

Assert

- **Verifies a required runtime condition holds**
  - Constraint checking
  - Subqueries that must return one row



SQL intersection

Updates

- **Internally, all changes are known as *updates***
  - INSERT / UPDATE / DELETE / MERGE etc.
- **One of the most complex areas of the Query Processor**
  - Constraint checking
  - Indexed view maintenance
  - OUTPUT clause
  - Composable DML
  - Non-updating updates
  - Transient key violations
  - Halloween protection
  - INSTEAD OF triggers

SQL intersection

Update Iterator Properties

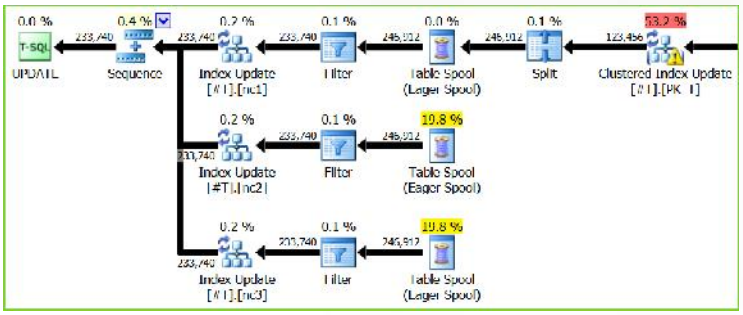
- **SSMS Object Properties**
  - Expand to see other affected indexes
- **DML Request Sort**
  - When *true*, rows must be presented in index key order
  - Required to enable insert minimal logging in SQL Server 2008+
- **Ordered or Unordered Prefetch**
  - Issues asynchronous reads
  - Reads about-to-be modified index pages into memory in advance

Halloween Protection

- **Read/write phase separation**
- **Not always an Eager Table Spool**
- **Iterators may combine to provide necessary level of protection**
- **Can explain redundant sorts**
- **Halloween Protection is required for:**
  - UPDATEs where index keys are both searched and updated
  - INSERTs where the target table is referenced in the SELECT clause
  - DELETEs with a self-join of the target table in the SELECT clause
  - MERGE similar to separate INSERT/UPDATE/DELETE operations
  - Specific “hole-filling” OLTP optimization for MERGE only

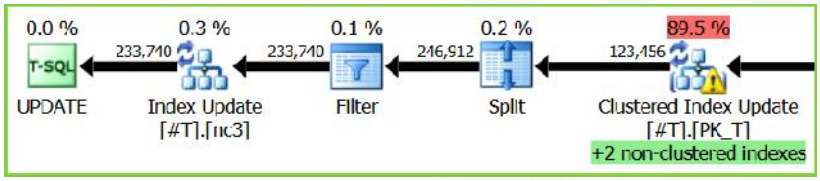
Per-Index (Wide) Update Plan Shape

- **Separate iterator branch for nonclustered index maintenance**
- **Eager Table Spool for multiple nonclustered indexes**
- **Sequence iterator processes each index fully in turn**
- **May sort keys per index to encourage sequential I/O**



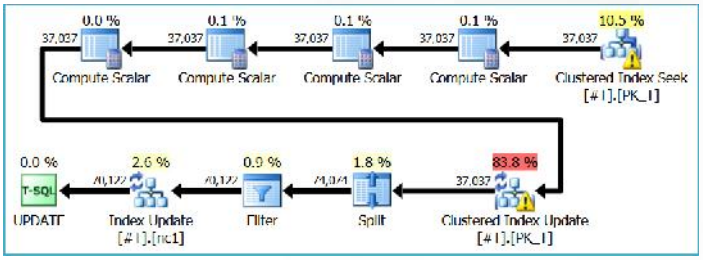
Per-Row (Narrow) Update Plan Shape

- **Multiple nonclustered indexes maintained by one iterator**
- **An optimization for smaller change sets**
- **I/O pattern may be random w.r.t each index**
- **Not available for all types of update**
- **Plans may combine per-index and per-row strategies**



Non-Updating Updates

- **Plan may be configured to skip *unchanged* nonclustered index rows**
  - Compute Scalars determine if a change in value has occurred
  - Filter eliminates any redundant updates
  - Filter not visible in per-row (narrow) plan shape
  - Clustered index changes are *never* skipped
  - Still best to explicitly filter unchanged rows in the query

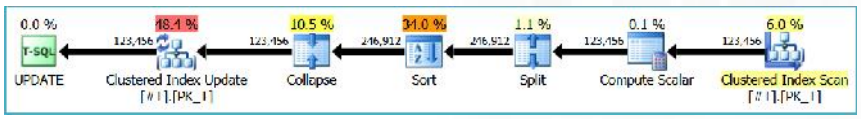


Demo

Demo X13
Changing Data

Split, Sort, Collapse Pattern

- **Required for unique index maintenance**
  - Prevents transient key violations
  - Storage Engine requires unique indexes to be unique at all times
  - Forces a per-index (wide) update plan for the index concerned
- **Algorithm**
  - Split the update into a delete and an insert
  - Sort by index key and action code
  - Collapse adjacent delete & insert pairs back to a single update

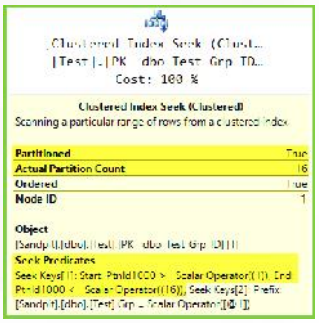


Demo

Demo X14
Split, Sort, Collapse

Partitioned Tables

- **APPLY model used in SQL Server 2005**
  - Constant Scan drives a per-partition nested loops join
- **SQL Server 2008+ uses an implied leading index key**
  - Still uses APPLY pattern for collocated join and partitioned index build
  - Except single partition OFFLINE build
  - Index order may be unexpected
  - Static or dynamic partition elimination is possible
  - Look for:
    - RangePartitionNew
    - PtnIdxxxx



Reverse Engineering an Execution Plan

- **Large plans can cause problems for optimization and execution**
  - Not always obvious how to simplify
  - QO may produce an optimal section as part of a poor plan
- **Often possible to write T-SQL directly from a plan section**
  - Check plan properties carefully
  - May require hints
- **Intermediate materialization**
  - Simpler queries generally optimize better
  - Better information for the optimizer
  - Easier to maintain and spot improvements e.g. useful indexes
  - Use common sense!

Demo

Demo X15
Reverse Engineering a Query Plan

SQL intersection

Final Thoughts

- **Common Execution Plan Problems**
  - Inaccurate estimates
  - Bad lookups
  - Residual predicates
  - Spilling hashes and sorts
  - Many to many merge
  - Avoidable sorts
  - Parallelism issues
  - Memory grant too large
  - Missing optimizer fixes
  - Inefficient whole-plan strategy
- **Plan Explorer Upload**

SQL intersection

Questions?

Don't forget to complete an online evaluation on Event Board!

Session: PRECON17

Mastering Execution Plan Analysis

Your evaluation helps organizers build better conferences and helps speakers improve their sessions.

Thank you!