



*The World's Largest Community  
of SQL Server Professionals*

# Introduction to Graphical Execution Plans in SQL Server 2005/2008

Brad M McGehee

Director of DBA Education

[Red Gate Software](#)

[www.bradmcgehee.com](http://www.bradmcgehee.com)

# My Assumptions About You

- You are a SQL Server DBA or developer with novice to intermediate knowledge of SQL Server.
- You have a basic understanding of indexing and Transact-SQL.
- You want to learn the basics of how to read and interpret execution plans so that you can:
  - Better understand how your queries are executing
  - And to help figure out ways of how to optimize them

The difference between a *run-of-the-mill* DBA/Developer from an *exceptional* DBA/Developer is that the exceptional DBA/Developer thoroughly understands how to use their available tools to their full potential.

# Our Focus Today

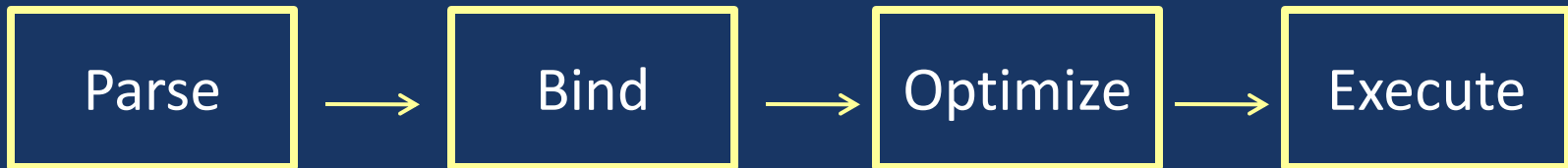
- The study of execution plans is a huge topic, one that we can't cover entirely in a short session.
- The goal of this session is to provide an **easy introduction** to:
  - Why execution plans are important
  - How executions plans are created
  - Learn about different type of execution plans
  - Learn the difference between estimated and actual execution plans
  - Learn different ways to capture execution plans
  - Learn the basics of how to read & interpret graphical execution plans
- After today's session, I highly recommend you read Grant Fritchey's free eBook on *SQL Server Execution Plans*.

# Why are Execution Plans Important?

- A execution plan, simply put, **describes the data retrieval/storage methods** chosen by the Query Optimizer to execute a specific query.
- So why is this important to know?
- Because you want to answer these questions:
  - Why are my queries running slow, and what can I do to boost their performance?
    - Has the query optimizer made a mistake?
    - Does the query optimizer have bad data (statistics)?
    - Are there missing indexes?
    - Is the query poorly written?

# How Execution Plans are Created

1. A “new” query is sent to the query optimizer .
2. The query is **parsed** to check if it is written correctly.
3. The query then goes through a **binding** process, where validation steps occur, which resolves all the names of the various objects, tables and columns, among other tasks.
4. The query then goes through the **optimization** process, where different execution plans are explored, and then one is selected based on the lowest cost (generally).
5. Then the query is cached in the plan cache, and **executed** by the query execution engine.



# More on the Execution Plan Process

- DDL code is not optimized, because there is only one way to perform such tasks.
- Some queries have such an easy and obvious execution plan that a **trivial plan** is created, and the query is never really “optimized”.
- The creation of an “optimized” execution plan takes time.
  - In some cases, all potential execution plans are analyzed, and the least cost one is selected. Called a **full optimization**.
  - In other cases, full optimization begins, but only a **good enough** plan may be selected if the optimization process may take too long (times out).
- The Query Optimizer uses index and column statistics to help it determine the cost of potential execution plans. **Out of date or bad statistics** can cause the Query Optimizer to produce bad execution plans. The Query Optimizer can force statistics updates to occur.

# Execution Plans Are Cached

- Once an execution plan has been created for a query, it is stored in the **plan cache**, where it can hopefully be reused.
- **Reusing a cached execution plan can save time** because a new execution plan does not have to be recreated each time the same query is re-executed. [This, of course, is why SQL Server caches execution plans in the first place.]
- **Plan caching and recompilation** can significantly affect the performance of a server, and needs serious consideration of its own, but this is an entire subject of its own.

# Execution Plans Come in Different Forms

- Text (Deprecated)
  - Harder for some people to read, but some DBAs prefer it.
  - Because it is text, large plans can be easily search for “text” strings.
- XML (A Storage Format)
  - Executions plans are now stored as XML code
  - XML code is not really designed to be read directly by DBAs
  - Can be saved and shared with others (makes execution plans portable)
  - Is generally displayed in graphical format (more below)
- Graphical (A Display Format)
  - Is the graphical (visual) form of the XML execution plan (see above)
  - Generally easier to read and understand by beginners that text or XML
  - Our focus today
- Demo

# Actual vs. Estimated Execution Plans

- Actual Execution Plan
  - Produced after a query actually runs.
  - Displays a combination of estimated and actual results.
- Estimated Execution Plan
  - Produced without running the query.
  - Can't be used if the query creates objects it needs to use: i.e. temp table.
  - Displays estimated data only (based on index and column statistics).
  - Not guaranteed to represent actual query plan. For example, the optimizer might force a statistics update, which might change the actual plan.
  - Can save time and resources when testing long running queries.
  - Allows you to explore data modification queries without modifying data.
- Demo

# How to Capture/Produce Execution Plans

- SSMS (our choice for today)
- Profiler (Showplan XML Event)
- SQL Server 2005 Performance Dashboard
- SQL Server 2008 Data Collector
- DMVs (requires some fancy queries)
- SET commands (shown in previous demo)

# How to Read Graphical Execution Plans
















- Demo
  - Overview of Graphical Execution Plan Screen
  - Using the Zoom Button (Around, In, Out, Fit)
  - Learning to Read from *Right to Left* and *Top to Bottom*
  - Learning about Operators
  - Learning about Tool Tips (and Properties)
  - Learning about Arrows
  - Executing Multiple Queries


# Execution Plans are Made Up of Operators

- Each operator receives rows, performs some operation on them, then sends the rows to another operator. For example:
  - Scanning data from a table
  - Seeking data in a table
  - Sorting
  - Joining
- In total, there are 79 different **operators** that can be included in an execution plan.































# Graphical Execution Plan Operators

	<a href="#">Arithmetic Expression</a>
	<a href="#">Assert</a>
	<a href="#">Bitmap</a>
	<a href="#">Bookmark Lookup</a>
	<a href="#">Clustered Index Delete</a>
	<a href="#">Clustered Index Insert</a>
	<a href="#">Clustered Index Scan</a>
	<a href="#">Clustered Index Seek</a>
	<a href="#">Clustered Index Update</a>
	<a href="#">Collapse</a>
	<a href="#">Compute Scalar</a>
	<a href="#">Concatenation</a>
	<a href="#">Constant Scan</a>
	<a href="#">Delete</a>
	<a href="#">Deleted Scan</a>









	<a href="#">Eager Spool</a>
	<a href="#">Filter</a>
	<a href="#">Hash Match</a>
	<a href="#">Hash Match Root</a>
	<a href="#">Hash Match Team</a>
	<a href="#">Insert</a>
	<a href="#">Inserted Scan</a>
	<a href="#">Iterator Catchall</a>
	<a href="#">Lazy Spool</a>
	<a href="#">Log Row Scan</a>
	<a href="#">Merge Interval</a>
	<a href="#">Merge Join</a>
	<a href="#">Nested Loops</a>
	<a href="#">Nonclustered Index Delete</a>
	<a href="#">Nonclustered Index Insert</a>








# Graphical Execution Plan Operators




	<a href="#">Nonclustered Index Scan</a>
	<a href="#">Nonclustered Index Seek</a>
	<a href="#">Nonclustered Index Spool</a>
	<a href="#">Nonclustered Index Update</a>
	<a href="#">Online Index Insert</a>
	<a href="#">Parameter Table Scan</a>
	<a href="#">Remote Delete</a>
	<a href="#">Remote Insert</a>
	<a href="#">Remote Query</a>
	<a href="#">Remote Scan</a>
	<a href="#">Remote Update</a>
	<a href="#">RID Lookup</a>
	<a href="#">Row Count Spool</a>
	<a href="#">Segment</a>
	<a href="#">Sequence</a>

	<a href="#">SequenceProject</a>
	<a href="#">Sort</a>
	<a href="#">Split</a>
	<a href="#">Spool</a>
	<a href="#">Stream Aggregate</a>
	<a href="#">Switch</a>
	<a href="#">Table Delete</a>
	<a href="#">Table Insert</a>
	<a href="#">Table Scan</a>
	<a href="#">Table Spool</a>
	<a href="#">Table Update</a>
	<a href="#">Table-valued Function</a>
	<a href="#">Top</a>
	<a href="#">UDX</a>
	<a href="#">Update</a>

# Graphical Execution Plan Operators

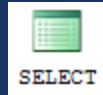
Icon	Language element
	<a href="#">Assign</a>
	<a href="#">Convert</a>
	<a href="#">Declare</a>
	<a href="#">If</a>
	<a href="#">Intrinsic</a>
	<a href="#">Language Element Catchall</a>
	<a href="#">Result</a>
	<a href="#">While</a>

Icon	Cursor physical operator
	<a href="#">Cursor Catchall</a>
	<a href="#">Dynamic</a>
	<a href="#">Fetch Query</a>
	<a href="#">Keyset</a>
	<a href="#">Population Query</a>
	<a href="#">Refresh Query</a>
	<a href="#">Snapshot</a>

Icon	Parallelism physical operator
	<a href="#">Distribute Streams</a>
	<a href="#">Repartition Streams</a>
	<a href="#">Gather Streams</a>

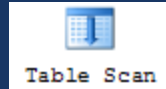
# Operators Covered Today

- SELECT
- Table Scan
- Clustered Index Scan
- Clustered Index Seek
- Non-Clustered Scan
- Non-Clustered Index Seek
- RID Lookup
- Key Lookup
- Sort
- Joins (loop, merge, hash)



# SELECT

- Represents the end results of a SELECT query.
- Good place to start when evaluating most execution plans.
- To optimize performance, the number of rows that are returned should be the minimum number of rows necessary to meet the needs of the query.



# Table Scan

- A table scan indicates there is no clustered index on the table, and the table is a heap.
- A table scan indicates that every row in the table (heap) had to be examined to see if it met the query criteria, which can mean slow performance if there are a large numbers of rows.
- Generally, heaps should be avoided.
- In most cases, you will want to add a clustered index to every table, as it has the potential of boosting the performance of the query, even if a clustered index scan is still conducted. This is because leaf nodes of the clustered index are stored together (logically, and hopefully physically). In a heap, they may or may not be, which can hurt performance, as scans work better if data is contiguous on disk.



Clustered Index Scan (Clustered)

# Clustered Index Scan

- A clustered index scan is a full or (partial scan) of all the rows of a table that have a clustered index.
- Like a table scan, clustered index scans can be slow and use up lots of server resources (for large tables).
- Clustered index scans are almost always better than table scans.
- If you see a clustered index scan, you should investigate to see why it is occurring. It is a hint that a query may be having performance problems. Perhaps the WHERE clause is not restrictive enough, or not sargable.
- On some occasions, when tables are small or many rows are returned, then a clustered index scan might be the fastest way to return data.



Clustered Index Seek (Clustered)

# Clustered Index Seek

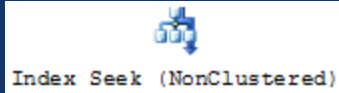
- If there is an available and useful index, and there is a sargeable WHERE clause, the query optimizer can very quickly identify the rows to be returned and return them without having to scan each row of the table.
- Generally speaking, if you see a clustered index seek in an execution plan, this is a good thing.
- One possible exception to this is if the clustered index seek is repeated over and over again. Check the operator's "Number of Executions" to find out.



Index Scan (NonClustered)

# Non-Clustered Index Scan

- All records (sometimes a range) in the table are scanned, and all rows that match the WHERE clause are returned.
- As with all scans, it can be slow and require extra I/O resources.
- Generally, seeing a non-clustered index scans should be seen as a hint of potential performance problems.
- Sometime, these scans can be turned into seeks if you modify the WHERE clause so that it is more restrictive or is sargable.



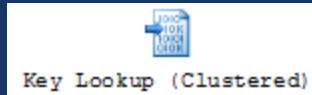
# Non-Clustered Index Seek

- A non-clustered index is used to identify the row(s) to be returned, so every row does not need to be scanned (assumes sargeable WHERE clause).
- This is generally much faster than a non-clustered index scan.
- Like clustered index seeks, non-clustered index seeks are generally a good thing.
- One exception is if bookmark lookups occur as part of the non-clustered index seek, then performance may lag if many rows are returned.



# RID Lookup

- A RID Lookup is a form of a bookmark lookup on a non-clustered index on a table without a clustered index (a heap).
- If you see a RID Lookup, this is a strong hint of potential performance problems.
- Generally, RID Lookups should be eliminated with the addition of an appropriate clustered index, and if necessary, a covering or included index.



# Key Lookup

- A key lookup is a bookmark lookup on a table with a clustered index.
- While key lookups are often faster than “scans,” this is not always the case. If the query returns a small number of rows, a key lookup is probably OK.
- But if many rows are returned, then it may cause a performance problem that needs correction.
- Often, key lookups can be eliminated with the addition of a covering or included index.



# Sort

- Sorts occur when you specify that returned data be ordered, or because the query optimizer needs to sort data internally to produce the desired results.
- Sorts are normal and aren't generally a problem.
- But if you return too much data, then sorts may take a lot of resources (including tempdb), and you should identify ways of reducing the number of rows returned to boost query performance.



# Joins (loop, merge, hash)

- The **nested loop** join compares each row from one table (the “outer table”) to each row from the other table (the “inner table”), looking for rows that satisfy the join predicate.
- The **merge join** works by simultaneously reading and comparing the two **sorted** inputs one row at a time. For each step, it compares the next row from each input. If the rows are equal, it outputs a joined row and continues. If the rows are not equal, it discards the lesser of the two inputs and continues.
- The **hash join** algorithm executes in two phases known as the “build” and “probe” phases. During the build phase, it reads all rows from the first input, hashes the rows on the equijoin keys, and creates or builds an in-memory hash table. During the probe phase, it reads all rows from the second input (often called the right or probe input), hashes these rows on the same equijoin keys, and looks or probes for matching rows in the hash table.



# More On Joins

- There is no “ideal join,” it all depends on the data being joined.
- From a resource perspective, a **nested loop join** generally uses fewer resources, and seeing one generally is often an indicator of good overall performance. Usually best for smaller joins. No memory, & non-blocking.
- **Merge joins** are often used for moderate to large data sets, and are most efficient if the joined columns are pre-sorted, otherwise they have to be sorted before the merge join can occur. If a sort is required, requires extra memory, but non-blocking.
- **Hash joins** are often used when very large data sets are used. Hash joins parallelize and scale better than other joins and are good at minimizing response times for OLAP queries. They use much memory and cause blocking. Seeing one may be a hint that too much data is being returned, especially in OLTP applications.
- **Demo**

# Take Aways From This Session

- This is just a small sample of the things you can do with Graphical Execution Plans.
- The learning curve for learning about Graphical Execution Plans is high, but worth the effort.
- Graphical Execution Plans are a powerful tool to help DBAs understand how a query executes.
- Based on the information provided by an execution plan, and the DBA's knowledge of how SQL Server works, many queries can often be optimized to perform better.

# Q&A

- Please ask your questions clearly and loudly.
- If you don't get your questions answered now, see me after the session, or e-mail me.

# Find Out More

## Free E-Books:

- [www.sqlservercentral.com/Books](http://www.sqlservercentral.com/Books)

## Check these out:

- [www.SQLServerCentral.com](http://www.SQLServerCentral.com)
- <http://www.simple-talk.com/sql/performance/graphical-execution-plans-for-simple-sql-queries/>

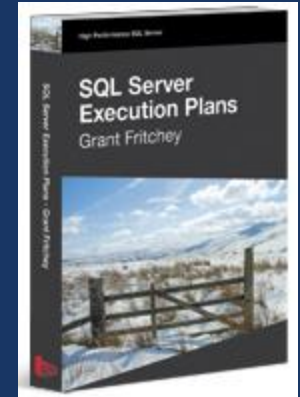
## Contact me at:

[bradmcgehee@hotmail.com](mailto:bradmcgehee@hotmail.com)

## Blogs:

[www.bradmcgehee.com](http://www.bradmcgehee.com)

[www.twitter.com/bradmcgehee](http://www.twitter.com/bradmcgehee)



[Click Here for a free 14-day trial of the Red Gate SQL Server Toolbelt](#)