

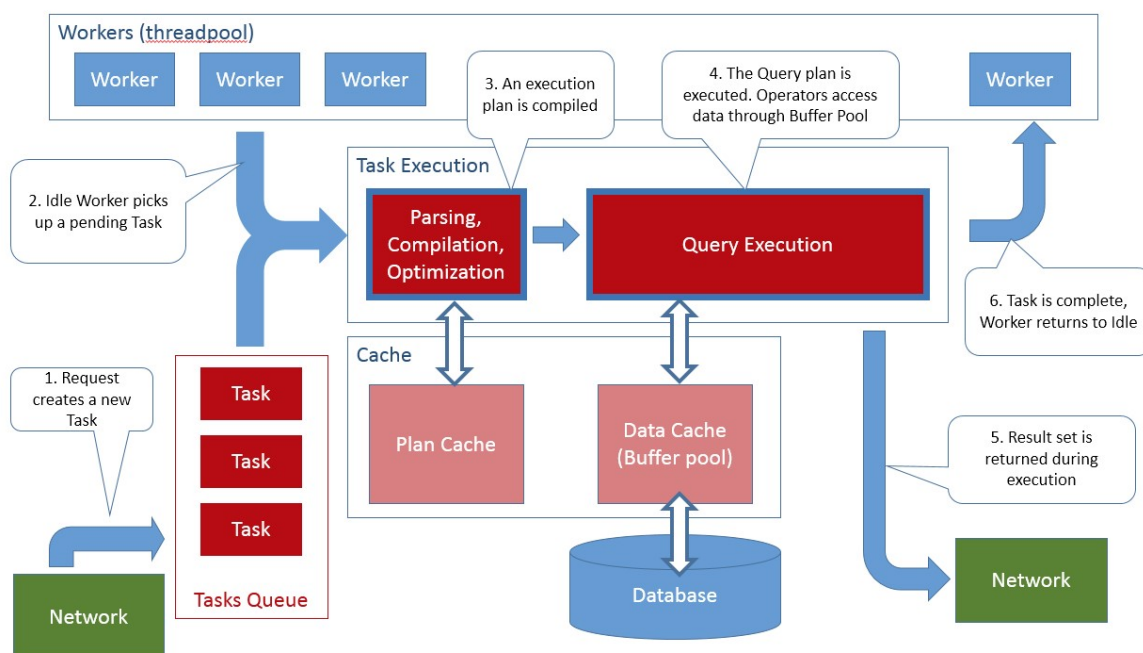
rusanu.com

- [About](#)
- [Links](#)
- [Articles](#)
- [Blog](#)

Understanding how SQL Server executes a query

August 1st, 2013

If you are a developer writing applications that use SQL Server and you are wondering what exactly happens when you 'run' a query from your application, I hope this article will help you write better database code and will help you get started when you have to investigate performance problems.



Requests

SQL Server is a client-server platform. The only way to interact with the back-end database is by sending requests that contain commands for the database. The protocol used to communicate between your application and the database is called TDS (Tabular Data Stream) and is described on MSDN in the Technical Document [\[MS-TDS\]: Tabular Data Stream Protocol](#). The application can use one of the several client-side implementations of the protocol: the CLR managed `SqlClient`, `OleDb`, `Odbc`, `Jdbc`, `PHP Driver for SQL Server` or the open source `FreeTDS` implementation. The gist of it is that when your application wants the database to do *anything* it will send a request over the TDS protocol. The request itself can take several forms:

Batch Request

This request type contains just T-SQL text for a batch to be executed. This type of requests do not have parameters, but obviously the T-SQL batch itself can contain local variables declarations. This is the type of request `SqlClient` sends if you invoke any of the [`SqlCommand.ExecuteReader\(\)`](#), [`ExecuteNonQuery\(\)`](#), [`ExecuteScalar\(\)`](#), [`ExecuteXmlReader\(\)`](#) (or they respective asynchronous equivalents) on a `SqlCommand` object with an empty [Parameters](#) list. If you monitor with SQL

Profiler you will see an [SQL:BatchStarting Event Class](#)

[Remote Procedure Call Request](#)

This request type contains a [procedure identifier](#) to execute, along with any number of parameters. Of special interest is when the procedure id will be 12, ie. [sp_execute](#). In this case the first parameter is the T-SQL text to execute, and this is the request that what your application will send if you execute a SqlCommand object with a non-empty Parameters list. If you monitor using SQL Profiler you will see an [RPC:Starting Event Class](#).

[Bulk Load Request](#)

Bulk Load is a special type request used by bulk insert operations, like the [bcp.exe utility](#), the [IRowsetFastLoad](#) Oledb interface or by the [SqlBulkcopy](#) managed class. Bulk Load is different from the other requests because is the only request that starts execution before the request is complete on the TDS protocol. this allows it to start executing and then start consuming the [stream](#) of data to insert.

After a complete TDS request reaches the database engine SQL Server will create a task to handle the request. The list of requests in the server can be queried from [sys.dm_exec_requests](#).

Tasks

The above mentioned task created to handle the request will represent the request from beginning till completion. For example if the request is a SQL Batch type request the task will represent the entire batch, not individual statements. Individual statements inside the SQL Batch will not create new tasks. Certain individual statements inside the batch may execute with parallelism (often referred to as DOP, Degree Of Parallelism) and in their case the task will spawn new sub-tasks for executing in parallel. If the request returns a result the batch is complete when the result is completely consumed by the client (eg. when you dispose the [SqlDataReader](#)). You can see the list of tasks in the server by querying [sys.dm_os_tasks](#).

When a new request reaches the server and the task is created to handle that request, in PENDING state. At this stage the server has no idea yet what the request actually is. The task has to start executing first, and for this the engine must assign a worker to it.

Workers

Workers are the thread pool of SQL Server. A number of workers is created initially at server start up and more can be created on-demand up to the configured [max worker threads](#). Only workers execute code. Workers are waiting for PENDING tasks to become available (from requests coming into the server) and then each worker takes exactly one task and executes it. The worker is busy (occupied) until the task finishes completely. Tasks that are PENDING when there are no more available workers will have to wait until one of the executing (running) task completes and the worker that executed that task becomes available to execute another pending task. For a SQL batch request the worker that picks up that task will execute the entire SQL batch (every statement). This should settle the often asked question whether statements in a SQL batch (=> request => task => worker) can execute in parallel: no, as they are executed on a single thread (=> worker) then each statement must complete before the next one starts. For statements that internally use parallelism (DOP > 1) and create sub-tasks, each sub-task goes through exactly the same cycle: it is created as PENDING and a worker must pick it up and execute it (a different worker from the SQL batch worker, that is by definition occupied!). The lists and state of workers inside SQL Server can be seen by querying [sys.dm_os_workers](#).

Parsing and Compilation

Once a task started executing a request the first thing it needs to do is to understand the content of the request. At this stage SQL Server will behave much like an interpreted language VM: the T-SQL text inside the request will be parsed and an [abstract syntax tree](#) will be created to represent the request. The entire request (batch) is parsed and compiled. If an error occurs at this stage, the requests terminates with a compilation error (the request is then complete, the task is done and the worker is free to pick up another pending task). SQL, and T-SQL, is a high end declarative language with extremely complex statements (think SELECT with several JOINS). Compilation of T-SQL batches does not result in executable code similar to native CPU instructions and not even similar to [CLI instructions](#) or [JVM bytecode](#), but instead results primarily in [data access plans](#) (or query plans). These plans describe the way to open the tables and indexes, search and locate the rows of interest, and do any data manipulation as requested in the SQL batch. For instance a query plan will describe an access path like `'open index idx1 on table t, locate the row with the key 'k' and return the columns a and b'`. As a side note: a common mistake done by developers is trying to come up with a single T-SQL query that cover many alternatives, usually by using clever expressions in the WHERE clause, often having many OR alternatives (eg. `(COLUMN = @parameter OR @parameter IS NULL)`). For developers trying to keep things [DRY](#) and avoiding repetition are good practices, for SQL queries they are plain bad. The compilation has to come up with an access path that work for *any* value of the input parameters and the result is most often sub-optimal. I cannot close this side without urging you to read [Dynamic Search Conditions in T-SQL](#) if you want to learn more about this subject.

Optimization

Speaking of choosing an optimal data access path, this is the next stage in the lifetime of the request: optimization. In SQL, and in T-SQL, optimization means choosing the best the data access path from all the possible alternatives. Consider that if you have a simple query with join between two tables and each table has an additional index there are already 4 possible ways to access the data and the number of possibilities grows exponentially as the query complexity increases *and* more alternative access paths are available (basically, more indexes). Add to this that the JOIN can be done using various strategies (nested loop, hash, merge) and you'll see why optimization is such an important concept in SQL. SQL Server uses a cost based optimizer, meaning that it will consider all (or at least many) of the possible alternatives, try to make an educated guess about the cost of each alternative, and then choose the one with the lowest cost. Cost is calculated primarily by considering the size of data that would have to be read by each alternative. In order to come up with these costs SQL Server needs to know the size of each table and the distribution of column values, which is are available from the statistics associated with the data. Other factors considered are the CPU consumption and the memory required for each plan alternative. Using formulas tuned over many years of experience all these factors are synthesized into a single cost value for each alternative and then the alternative with the lowest cost is chosen as the query plan to be used.

Exploring all these alternatives can be time consuming and this is why once a query plan is created is also cached for future reuse. Future similar requests can skip the optimization phase if they can find an already compiled and optimized query plan in the SQL Server internal cache. For a lengthier discussion see [Execution Plan Caching and Reuse](#).

Execution

Once a query plan is chosen by the Optimizer the request can start executing. The query plan gets translated into an actual execution tree. Each node in this tree is an operator. All operators implement an abstract interface with 3 methods: `open()`, `next()`, `close()`. The execution loop consists in calling `open()` on the operator that is at the root of the tree, then calling `next()` repeatedly until it returns false, and finally calling `close()`. The operator at the root of the tree will in turn call the same operation on

each of its children operators, and these in turn call the same operations on their child operators and so on. At the leaf the trees there are usually physical access operators that actually retrieve data from tables and indexes. At intermediate levels there are operators that implement various data operations like filtering data, performing JOINS or sorting the rows. Queries that use parallelism use a special operator called an Exchange operator. The Exchange operator launches multiple threads (tasks => workers) into execution and asks each thread to execute a sub-tree of the query plan. It then aggregates the output from these operators, using a typical [multiple-producers-one-consumer](#) pattern. An excellent description of this execution model can be found in the [Volcano-An Extensible and Parallel Query Evaluation System](#).

This execution model applies not only to queries, but also to data modification (insert, delete, update). There are operators that handle inserting a row, operators that handle deleting a row and operators that handle updating a row. Some requests create trivial plans (eg. a `INSERT INTO ... VALUES ...`) while other creates extremely complex plans, but the execution is identical for all of the and occurs just as I described: the execution tree is iterated calling `next()` until its done.

Some operators are very simple, consider for example the TOP(N) operator: when `next()` is called on it, all it just has to call `next()` on its children and keep a count. After N times being called, it simply return false w/o calling the children anymore, thus terminating the iteration of that particular sub-tree.

Other operators have more complex behavior, consider what a nested loop operator has to do: it needs to keep track of the loop iteration position on both the outer child and inner child, call `next()` on the outer child, rewind the inner child and call `next()` on the inner child until the join predicate is satisfied (see [Nested Loop Joins](#) for a more thorough discussion).

Certain operators have a stop-and-go behavior, meaning that they cannot produce any output until they consumed all the input from their own children operators. Examples of such operators is SORT: the very first call to `next()` does not return until all the rows created by the children operators are retrieved and sorted.

An operator like HASH JOIN will be both complex and stop-and-go behavior: to build the hash table it has to call `next()` on the build side child until that operator returns false. It then calls `next()` on the probe side child operator until a match is found in the hash table, then return. Subsequent calls continue to call `next()` on the probe side child operator and return on hash table match, until the probe side child operator `next()` returns false (see [Hash Join](#) for a more thorough discussion).

Results

Results are returned back to the client program as the execution proceeds. As rows 'bubble' up the execution tree, the top operator is usually tasked with writing these rows into network buffers and sending them to back to the client. The result is not created first into some intermediate storage (memory or disk) and then sent back to the client, instead it is sent back *as* is being created (as the query executes). Sending the result back to the client is, of course, subject to the network flow control protocol. If the client is not actively consuming the result (eg. by calling `SqlDataReader.Read()`) then eventually the flow control will have to block the sending side (the query that is being executed) and this in turn will suspend the execution of the query. The query resumes and produces more results (continue iterating the execution plan) as soon as the network flow control relieves the required network resources.

An interesting case is OUTPUT parameters associated with the request. To return the output value back to the client the value has to be inserted into the network stream of data that flows from the query execution back to the client. The value can only be written back to the client at the end of execution, as the request finishes. This is why output parameter values can only be checked after all results were consumed.

Query Execution Memory Grant

Some operators need significant amounts of memory to do their work. Sort operator requires to store all the input in order to sort it. Hash join and hash aggregates have to build large hash tables to do their work. The execution plan knows in the rough how much memory will be required by these operators, based on operator type, the estimated number of rows the operator has to process (“known” from the statistics) and the column sizes. The total amount of memory required by an execution plan for these operators is often referred to as the **memory grant**. When many concurrent queries are launched into execution they can run out of memory because of high number of memory expensive operators requesting large memory grants at the same time. To prevent this situation SQL Server uses a so called “resource semaphore”. This ensures that the sum of all memory grants allocated to executing queries does not exceed the total memory of the server. When the sum of memory grants is exhausting the available memory on the server then queries that are requesting further grants will have to wait for queries that are executing to finish and release the grants they have. The current status of memory grants (requested, allocated etc) can be queried from the DMV [sys.dm_exec_query_memory_grants](#). When a query has to wait for a memory grant and event is generated: [Execution Warnings Event Class](#):

The Execution Warnings event class indicates memory grant warnings that occurred during the execution of a SQL Server statement or stored procedure. This event class can be monitored to determine if queries had to wait one second or more for memory before proceeding, or if the initial attempt to get memory failed.

As always, things are a bit more complex when you dig into the details. Queries are not necessarily granted the entire memory amount requested, they can be launched into execution with a smaller memory grant than requested and this is fine. The operators are informed that a smaller amount of memory was granted and they adapt their run time execution to conform to the allotted memory grant. Another situation that can occur is that the memory estimate turns out to be insufficient (usually due to outdated statistics), in which case the operator is forced to **spill** to tempdb. Spilling to disk (writing to disk and reading back, instead of just doing the entire operation in memory) is slow and there are special warning events for them:

- [Exchange Spill Event Class](#).
- [Sort Warnings Event Class](#).
- [Hash Warning Event Class](#). Strictly speaking things are a bit more complex for hashes (they bail out, not spill), for more details see [Hash Warning SQL Profiler Event](#).

For more details about memory grants read [Understanding SQL server memory grant](#).

The memory granted to a query is a *reservation*, not an allocation. During execution the query requests actual memory allocations from this reserved grant and this is when memory is really consumed by the query. Is possible for a query to consume (allocate) less memory than granted (the estimates are usually pessimistic and account for worst cases). Memory that was granted but not consumed is used for data caching (buffer pool). However large estimates (grants) that are not used have a negative effect because they block other queries from even starting execution because of the resource semaphore limit.

A somehow related concept is the query compile resource semaphore. This is a similar gate as the execution gate, but it applies to query **compilation**, not to query execution. Normally this should never be an issue because compilations (should) occur rarely. A large number of requests blocked at the query compile gate indicates an issue with query plan reuse, see [Diagnosing Plan Cache Related Performance Problems and Suggested Solutions](#).

And a last note on the memory grants: remember that not all queries require a memory grant. Memory

grants are needed only by complex queries involving sorting, large scans (parallelism) and hash joins or aggregates (ie. unsorted input, impossible to use streaming aggregates). If you see memory grant issues on systems expected to have low latency (eg. a web site) then is time to reconsider your data model design. Query memory grants are valid in analytic scenarios (large queries where high latency is expected and tolerated).

Data Organization

At this moment I feel is necessary to introduce the way data is organized in SQL Server, because understanding of the Data Access topic depends on understanding the data organization. Data in SQL Server can be organized one of three ways:

Heaps

A heap is the table w/o any order defined on it. The heap contains all the columns of a table. If a table is organized as a heap then whenever you are talking about the 'table' then heap **is** the 'table'. If you did not declare a `PRIMARY KEY` clause when you created the table then the table is a heap. A table created by a `SELECT... INTO ... FROM ...` statement will be a heap. For more details on how heaps are organized see [Heap Structures](#).

Clustered Indexes

A clustered index is a table with an order defined on it. The clustered index contains all the columns of a table and if a table is organized as a clustered index then whenever you are talking about the 'table' the clustered index **is** the 'table'. Clustered indexes are [B-Trees](#). If you declare a primary key declared on a table then the same key is also used for the clustered index, unless the primary key is explicitly declared as `NONCLUSTERED`. For more details on how clustered index are organized see [Clustered Index Structure](#).

Nonclustered Indexes

A nonclustered index is a copy of a subset of a table data with a specific order defined on it. A nonclustered index contains one or more columns from the table. When talking about "indexes" on a table, most often the discussion refers to nonclustered indexes. Nonclustered indexes are [B-Trees](#). For more details on how nonclustered indexes are organized see [Nonclustered Index Structures](#).

With SQL Server 2012 there is another mode of organizing a data, namely [Nonclustered Columnstores](#) and in the next version of SQL Server there will also be a [Clustered Columnstore](#) mode. If you're interested in them read the articles linked.

Data Access

At the leaf extremities of the execution tree there are operators that implement the access to the data. The operators will return an actual row of data from a table (or from an index) when the `next()` method is called on them. There are three possible data access operators:

Scan Operator

The Scan operator will iterate through all the rows in its source. A scan can never locate a particular row, it always has to scan the entire data set (hence it's name...). If you inspect an execution plan you'll possibly see any of the operators for [Clustered Index Scan](#), [Nonclustered Index Scan](#), [Table Scan](#), [Remote Index Scan](#) and [Remote Scan](#). They are distinct operators because they apply to different data sources (indexes, tables, remote linked servers) but they all have in common the end-to-end scan behavior. As these operators have to read the entire data, always, they are always expensive. Only data warehousing queries should resort to these type of scans.

Seek Operator

The seek operator can locate a row directly based on a key. Seek can only operate on B-Tree

organized data sources, so it can only be applied to Clustered and Nonclustered indexes. If an index has a complex key (multiple columns) then the Seek operator can only operate if values for the leftmost keys in the index definition are provided. To give an example, if an index has the key columns (A, B, C) then the Seek can locate the first row where A='a', or the first row where A='a' AND B='b' or the first row where A='a' AND B='b' AND C='c'. However, on such an index, Seek cannot locate a row where B='b' or a row where C='c'. Seek operator is also capable of implementing ranges. Given the same index definition on (A, B, C) a Seek operator can iterate all rows where A > 'a' AND A < 'z' or all rows where A = 'a' AND B > 'b' AND B < 'z', but it cannot iterate rows where B > 'b' AND B < 'z'. If you inspect an execution plan you will possibly see any of the operators [Clustered Index Seek](#), or [Remote Index Seek](#). They are distinct operators because they apply to different data sources, but they all have in common the capability to efficiently locate a row based on a key value or to iterate efficiently over a range of key values. Obviously there are no heap seek operators, as heaps, being unordered, do not have the capability to locate a row efficiently based on a key. Seek should be the preferred data access method in almost every situation.

Bookmark Lookup Operator

Bookmark lookup is a special data access operator that can efficiently locate a row based on a special type of value, called a bookmark. You cannot provide bookmarks, only the engine internally can retrieve bookmarks for later lookup. And this is the gist of this special operator: it is never the primary data access operator, it will always be an operator used to look up a row that was previously accessed one of the other operators, a Scan or a Seek. Bookmarks can be looked up on any data organization mode, both on heaps and on B-Trees. In an execution plan you may see an [Bookmark Lookup](#), a [Row ID Lookup](#) (Heap specific look up) or a [Key Value Lookup](#) (B-Tree specific look up).

- Scans read all the data.
- Seek read only the minimum required data.
- Heaps only support scans.
- B-Trees can perform seeks only if the leftmost index key(s) are specified.

Strictly speaking all the operators used to insert, delete or update rows are also data access operators. The [Inserted Scan](#) and the [Deleted Scan](#) operators do access data to iterate over the trigger **inserted** and **deleted** pseudo-tables. [Log Row Scan](#) is a more esoteric data access operator (it reads rows from the log, not from the data tables). But going into this level of detail to explain how these work would derail the purpose of this very article.

Frequently you will see the concept of Range Scan being mentioned. This refers to a Seek operator that uses a seek to quickly locate a row by key and then it continues by iterating the rows from that position forward, sometimes up to a specific second key value. The Seek operator is performing a scan within the range defined by the start and end keys, hence the term Range Scan being used.

If we look back at how query execution occurs we can now understand how the data access operators drive the entire query plan iteration. When the `next()` method is called on the operator at the root of the query tree the call percolates down from parent operator to child operator until it reaches a data access operators. These operators implement `next()` by actually reading the data from the underlying source (heap or B-Tree) and returning the row read. They remember the position of the row returned and subsequent call to `next()` reads the next row and return this next row. Data access operators do not have more child operators, they sit at the leaves of the query plan tree. Operators higher on the tree implement functionality like filtering row, joining tables, sorting results, computing aggregates and so on atop the data returned by the data access operators.

Reading Data

The data access operators always read data from a cache, never from disk. This cache is called the Buffer Pool. If data is not present in the cache the data access operator must request it from the disk (issue a disk IO read) and wait until it is fetched into the cache. Data in the cache (in the Buffer Pool) is shared between all queries so once fetched subsequent data access operators that have to read the same data will benefit from finding the data in the cache. SQL Server will read as much data into this cache as is possible, growing the process allocated private memory until all the memory on the system is allocated to SQL Server (subject to a max limit configured by changing the [max server memory](#)). The Buffer Pool cache, as well as the IO read/write requests do not handle individual rows but instead they always operate on a 8Kb page.

Lets consider how a data access operator like a Scan would read data from an unordered heap:

- On first `next()` call the operator has to find the first row and return it. SQL Server stores metadata about tables that describe exactly which pages belong to a table, for more details I recommend reading [Managing Space Used by Objects](#) and [Inside the Storage Engine: IAM pages, IAM chains, and allocation units](#). The data access operator will request a reference from the Buffer Pool for this page which will return a pointer to the in-memory copy of the requested page. If the page is not in memory, the request blocks until the page is read from disk. The page contains an array of individual data records. A record is not necessarily an entire row, because large values and variable length values may be stored on another page. I recommend reading [Inside the Storage Engine: Anatomy of a record](#) for more details on how rows are arranged on the page. The data access operator will locate the first row on the page, copy out the requested field values and return. The data access operator keeps internal state that allows it to efficiently return to this position (same page and row).
- Parent operators consume the first row returned by the data access operator.
- When `next()` is called again on the data access operator it will use the previously established context state to position itself quickly on the current page and row, advance the row position by one, copy out the requested field values and return.
- Parent operators consume the next row returned by the data access operator.
- When `next()` is called again on the data access operator and all rows on the page have been consumed the operator will request from the Buffer Pool a reference for the next page. Which is the 'next' page is determined from the table metadata, and I refer you again to [Paul Randal's IAM article](#) for more details. Once it has the next page the operator returns the first row on this new page (it copies out the request fields and returns).
- Parent operators consume this row returned by the data access operator.
- This continues until the last row on the last page belonging to the table is returned. After this the operator has reached end-of-table, it's internal state is positioned 'beyond end of table' and cannot return any more rows.
- When the data access operator cannot return any more rows the parent operator does whatever action it has to do when the child operators are 'done'. For example a SORT operator can now start returning rows. A hash JOIN can start iterating the probe side and return rows. A SELECT operator can itself return false and cause further operators up the tree to finish, ultimately causing the query to complete.
- Data access operators can be rewind. For example if our heap scan operator would be the inner side child of a nested loop operator, when it is complete the parent nested loop would request the next row from the outer side child, then rewind our data access operator and iterate it again. Rewinding a data access operator causes it to reset its internal position state and will cause it to start again from the first row on the first page.

For comparison, here is how a data access operator would operate on a sorted B-Tree:

- On first `next()` call the operator has to Seek (find) the first row *that has the key requested* and

return it. SQL Server stores metadata about B-Trees that describe exactly which pages belong to an index but, unlike the Heap case when it has to go to the first page, the operator has to navigate to a specific key. From the metadata it retrieves the root page id and then request a reference from the Buffer Pool for this page. Using the searched key value, the data access operator navigates the B-Tree too reach the leaf page that contains the first row equal or after the searched key. At each step down the B-Tree the data access operator must request the relevant page from the Buffer Pool and possibly wait for it to be read from disk. On the leaf page the data access operator will have to search the rows in the page to locate the one with the desired key, copy out the desired column values and then return. It is possible that the data access operator does not find a row with the key value searched for. The B-Tree access can request an exact key value for the row, it can request the first row *after* the key value or it can request the first row equal or after the key value. A B-Tree can be searched in both direction (ascending or descending) so the definition of a row ‘after’ the key depends on the direction of the search. Not that this is not the same thing as having the index defined ascending or descending, which means change the actual order of rows in the B-Tree.

- Parent operators consume the first row returned by the data access operator.
- *If* the operator is used as a range scan, then `next()` will be called again asking to retrieve the row after the previously returned one. The B-Tree access operator would store the key value it returned previously and position itself on this key value, using the same procedure to navigate the B-Tree as described above, and then move to the next row. If there are no more rows on the page then the operator would return the first row on the next page (again, asking for the next page from the Buffer Pool and possibly having to wait for it to be read from disk). In B-Tree organized indexes the pages are linked, each page has on it a pointer (the page id) of the next page.
- Parent operators consume the next row returned by the data access operator.
- Range scans can contain an end range key value. In such case the call to `next()` may return false (not return a row) if the row located by moving to the next row from the current position is greater than the range end value. The term ‘greater’ is relative as the range scan can traverse the B-Tree in both ascending or descending order *and* the B-Tree itself may be organized ascending or descending. The operator can end the scan either when it reaches a key positioned after the end of range value, or when it reaches the last row on the last page of the B-Tree.
- In addition to being rewind, B-Tree operators can also be rebind. A rewind resets the operator state to start again the seek/scan with the same key/range parameters. A rebind will change the actual key values. See [Showplan Logical and Physical Operators Reference](#) for more details.

Read Ahead

From the description of how scan operators operate you can see that every time they finish reading all the rows on a page they have to ‘fix’ the page into the buffer pool and this potentially can stall the operator as it has to wait for the page to be fetched from disk into memory. If the operator has to stall at every page, performance plummets. The solution is to *read ahead* pages that are not referenced by the operator now, but will be referenced soon. SQL Server does this and issues asynchronous read ahead requests for pages that *will* be required by the scan operator before the operator actually reaches that page to read the rows. Luckily by the time the operator reaches that page the page is already in the buffer pool and ‘fixing’ the page is nearly instantaneous. For more details see [Reading Pages](#), [Sequential Read Ahead](#). There exists also a special kind of read ahead for reading random pages for a nested loop, see [Random Prefetching](#).

Latches: page concurrent access protection

For a detailed discussion about latches read the [Diagnosing and Resolving Latch Contention on SQL Server](#) whitepaper.

Before going on to how data writes are executed, it is necessary to give a brief description to the mechanisms that exist in place to ensure that always correct data is read from a page. Under multi-threading it is always necessary to ensure that readers do not read an incomplete write. All programmers are familiar with the primitive structures used to enforce such protection: mutexes, semaphores, critical sections. In database nomenclature though the established term is a **latch**. Latches are data structures that protect resources for concurrent access (eg. each page in the buffer pool has a latch to protect it). Latches support multiple acquire modes: shared, exclusive, update, keep, destroy. The following table shows the latch mode compatibility:

	KP	SH	UP	EX	DT
KP	Y	Y	Y	Y	N
SH	Y	Y	Y	N	N
UP	Y	Y	N	N	N
EX	Y	N	N	N	N
DT	N	N	N	N	N

Whenever a query operator needs to access a page (eg. to compare a key, or to read a row) it must acquire the page latch in SH mode. Multiple operators can read the same page concurrently as multiple threads can acquire the page latch in SH mode. Any operator that needs to modify the page must acquire the page latch in EX mode. Only one operator can modify the page at a time, and no other operators can read the same page until the modification is complete. These two types of requests show up in the wait stats as `PAGELATCH_SH` and `PAGELATCH_EX` wait types. The [Diagnosing and Resolving Latch Contention on SQL Server](#) whitepaper has more details, including on how the page latch also protects IO requests (so that the page is only read once from disk even if multiple concurrent threads need to fetch it from disk) and also how high end multi-CPU systems use **superlatches** to avoid processor cache invalidation.

It is important not to confuse the physical page protection offered by latches with the logical protection of locking. Locks can be controlled by the client (eg. by choosing a transaction isolation level) but latches are **always** required.

Locks

While latches offer physical protection for concurrent access between threads, locks offer logical protection for concurrent access between transactions:

Logical vs. Physical

Locks describe the entity being locked, they're not the actual entity. Locks are 6 byte strings that describe what is being locked. Latches are the actual object in memory being locked, to acquire a latch the code must somehow obtain a reference (the pointer, the address) to the latch object. To obtain a lock the code has just to construct the proper 6 byte string describing the object being locked and then request the lock from the Lock Manager.

Threads vs. Transactions

Latches are acquired and released by the threads executing inside the process. Locks are requested and released by transactions. This is a very important distinction, as transactions can span several requests and execute on multiple threads (either concurrently, in case of parallel execution, or sequentially in case of subsequent requests being picked up by different workers).

Locks are more complex than latches, not least because there are 22 lock modes documented in the [Lock Compatibility MSDN chapter for SQL Server 2008 R2](#). Add cryptic names such as `SCH-S`, `SIX` or `RX-U` and it is no wonder they seem such an arcane feature. Let's try to clear up a little bit what these various lock

modes mean.

Schema Stability

While a query plan is being executed it is desirable that the objects referenced by this plan (tables, indexes) do not change structure (eg. drop a column from a table) nor vanish altogether. To prevent such changes all queries obtain upfront at the start of the execution a so called “schema stability lock” on all the objects referenced by the plan. This is the **SCH-S** lock mode. DDL statements will obtain the **SCH-M** lock (the schema modification) which is the most powerful (most restrictive, least compatible) lock mode there is, ensuring that no other statement can even reference this object. In practice most queries not acquire **SCH-S** mode but instead they acquire the equivalent data intent lock, like **IS**.

Shared, Update, Exclusive

The basic lock modes are **S** (shared), **U** (update) and **X** (exclusive). **S** and **X** are self explanatory. The **U** mode is acquired when inspecting a row that *might* be later updated or deleted (hence the lock may be upgraded to **X**). Acquiring **U** locks do not block reads, but blocks other queries from also inspecting the row for a potential update/delete. Without an **U** mode two queries attempting to update the same row would deadlock as they would both attempt to escalate the **S** mode to **X** mode. And having the update queries acquire directly **X** mode on all rows, even those that turn out not to qualify for the update, would result in unnecessary blocking of reads.

Lock Hierarchy and Intent Locks

All the lock modes that have an **I** in the name are *intent* locks. To understand what intent locks are you have to think at the problem of lock hierarchy. Consider a query that is going scan an entire table, of a few million rows. Acquiring individual locks on the rows will result in few million row lock requests, an expensive proposition. Yes, acquiring a lock is fast, but even fast adds up when repeated ad nauseam... So the query decides instead to lock the entire table and places on **S** lock on the table object rather than locking each individual row. Another concurrent query wants to delete the a row in the table so it requests an **X** mode lock on that row. And here is the conundrum: how is the Lock Manager supposed to know that it cannot grant the **X** mode row lock because the entire table is locked in **S** mode by another transaction? Remember that locks are just strings describing what is being locked, and there is no way the Lock Manager could know that the string describing the row requested in **X** mode happens to be a row in the table on which an **S** lock mode was granted. The solution is the intent locks. The operators requesting locks know the actual hierarchy of the objects involved: the update operator know that row it tries to update belongs to that table. So is the responsibility of the operator to request an intent lock on the **parent** object(s) in this hierarchy. The update operator will first request an **IX** mode lock on the table (“intent exclusive”) which can be read as a declaration: “I am not locking the table, but I may lock a row in this table in **X** mode”. This **IX** mode will not be compatible with the **S** (shared) mode requested by the scan that wants to lock the entire table so now the problem is solved by making the hidden conflict explicit, thus blocking one of the transactions until the other completes. So when you see lock modes like **SIX** you can understand it’s meaning: “I am locking the table in **S** mode **and** I may lock a row in this table in **X** mode”.

Key-Range locks

Lock modes that contain an **R** in the name are range locks. Range locks are used to protect rows that *do not exists*. Key range locking is the mechanism used by SQL Server to obtain [serialization isolation level](#) between transactions (ie. no [Phantom Reads](#)). As serialization isolation requires that no new rows appear in a result set when a query is run repeatedly under the same transaction, the way to ensure this requirement is to lock not only the actual rows returned by a query, but also the entire range covering these rows so that concurrent INSERT cannot creep in new rows between the rows already returned by the first execution of the query. See [Key-Range Locking](#) for more details.

Writing Data

Operators that modify (write) data are very similar to the read operators presented above. When `next()` is called on a write operator it has to locate the row (or, for an insert, locate the position to insert a new row) and then do the actual modification. Then, when `next()` is called again, it does the modification on the next row. Delete and update operators are usually driven by other read operators that locate the row to be deleted or updated and these read operators pass to the write operators a bookmark that locates exactly the row to be modified. The actual inserting, deleting and modifying data though is a bit more convoluted than reading. SQL Server uses Write Ahead Logging which means that each and every modification done to the data has to be first described in the log. Roughly speaking all writes occur in the following sequence:

A detailed description of this process can be found at [SQL Server 2000 I/O Basics](#).

- The write operator is 'positioned' on the page that has to be modified (a row has to be inserted, deleted or modified on the page). This means that the page must be 'fixed' in the SQL Server cache.
- The operator must obtain an exclusive latch to the page. This guarantees that no other operator can read this page.
- The operator must generate a log record describing exactly the operation it is about to perform on the page and append this log record into the log.
- The operator can now modify the in memory cached image of the page.
- The LSN generated for the log record created before the page was modified must be written into the page header as the 'last modified LSN'.
- The page exclusive latch is released. The page can now be read by any other operator. It is important to understand that until now no disk write occurred yet, everything is just modifications done in memory.
- Before the transaction that did the modification commits it **must** generate a new log record describing the fact that this transaction is committed, this log record is appended to the log and **all** log records in memory, up to and including this commit record, must be written to disk. By definition this will include above log record that describes the modifications done to the page. Note that the page is still only modified in memory and not written to disk. At this moment the transaction is durable because if the server crashes the modification done to this page is guaranteed to be described in the log, saved on disk, and the recovery process will redo this operation if necessary.
- Periodically the modified ('dirty') pages in the SQL Server cache are written to the disk. This is called a ['checkpoint'](#).

There is a special type of write that occurs using a different sequence: a minimally logged write. Only operations that insert new data can do minimally logged operations such as `INSERT` and append of a blob field using the [.WRITE\(@value, NULL,...\)](#) syntax of `UPDATE`. Certain conditions must be met before a minimally logged operation can occur. See [Operations That Can Be Minimally Logged](#) and also read [The Data Loading Performance Guide](#). The sequence of operations done in a minimally logged operation is, roughly, the following:

- The operator doing the minimally logged operation (a bulk insert operator) allocates a new page. See [Managing Extent Allocations and Free Space](#) to understand how pages are being allocated.
- The operator 'fixes' the page in the cache and obtains an exclusive latch on it.
- A log record is generated describing the fact that the page is being used for minimally logged bulk insert. This log record is appended to the log (in memory) and the record LSN is written on the page as the last modified LSN.
- The page is added to a list of minimally logged pages associated with the transaction.
- The operator can now add as many rows as they fit on the page. It does not need to generate any log to describe each row. Only the in memory image of the page is modified, no disk write occurs.
- When the page fills up a new page is allocated and the process described above repeats.

- Before the transaction that did the minimally logged operation commits, all the pages modified in minimally logged mode by this transaction **must** be written to disk. After all the pages were written to disk, a new log record describing the fact that the transaction committed is generated and appended to the log (in memory). All the log up to and including this last log record must be written to disk now.
- To prevent a ‘thundering herd’ phenomenon of all pages modified by the minimally logged operation attempting to write to disk at the same time at the end of the transaction an eager write process is writing them to disk even before the transaction is committing.

Minimally logged operations are still fully transactional, consistent and durable. Up to the last moment the operation can be aborted (rolled back) or the server can crash and the database is left in a consistent state (after recovery). Just as with the fully logged operations, a detailed description of this process can be found at [SQL Server 2000 I/O Basics](#).

DDL

Not all T-SQL statements are executed as an iteration of operators in a execution plan tree. The typical example are DDL statements, like `CREATE TABLE`. To understand how DDL statements work, is important to understand that SQL Server stores all the metadata about any object in the database in internal system tables. An insert into the system tables that describes the existing tables will result in a new table being recognized by SQL Server. An insert into the system table that describe columns will add a column to a table. Deleting a row in these tables it means the table object or a column of a table gets dropped.

Everything inside the database is described by these system tables, there are about 80 system tables that cover objects, procedures, functions, schemas, users, logins, certificates, views, partitions, permissions, databases, files, literally every SQL Server concept. Therefore what DDL statements have to do is just to maintain these system tables. A statement like `CREATE TABLE` has to insert a row in the system table describing objects (tables are just one of the possible object types) and some rows in the system tables describing columns and voila, your new table was ‘created’ (Keep in mind I’m simplifying a great deal). While DDL statements do not use directly the operators that I mentioned above, they do use the same code to effectively access the system tables (read rows, write rows). When a DDL statement executes it does not call `next()` on an Seek operator but instead is using directly the code that the said Seek operator would had used to locate the row it desires in the system table. The DDL statement accomplishes its work by inserting, deleting or updating rows in these system tables. A few DDL statements have additionally do some operations outside these system tables, eg. they have to create or delete files on disk for the database, or they have to connect to Windows Clustering API to configure Availability Groups. And some DDL statements have to manipulate the data tables internally, eg. to populate the default values of a new column or validate that the existing data conforms to a newly added check constraint.

BACKUP, RESTORE and DBCC


After we sift out the query statements (including DML) and the DDL we’re left with a few special statements. BACKUP and RESTORE operates by bypassing almost *everything* we discussed so far. From a 10000ft view what BACKUP and RESTORE ultimately do is just a glorified copy from one file to another. BACKUP reads from the data and/or log files and writes into the backup file. RESTORE reads from the backup file and writes into the data and/or log files. They do need to do some housekeeping of system tables, but the bulk of work they simply read from a file and write into another. In doing so they bypass the data cache (the buffer pool). As for DBCC statements, pretty much each one does something different. For an example of how DBCC CHECKDB works I recommend reading the series of articles [CHECKDB from every angle](#).

How can I use all this information?


Developers that interact with databases have two main pain points: troubleshooting performance and troubleshooting data loss issues. This article will not help you much with the later, if you do not have a backup, nothing will save your, sorry! But I hope that understanding how things work will shed some light into the performance troubleshooting problems. Once you understand that your client is sending requests to the server and the server is creating a task for for each request, the performance conundrum can be simplified a great deal: at any moment your task is either executing (consuming CPU cycles) or is waiting. And every time it waits, and I mean *everytime*, the wait information (what was waited on and how long) will be collected by SQL Server internal wait info statistics. There is an excellent methodology on how to leverage these wait info collected statistics to troubleshoot performance bottlenecks: the [Waits and Queues](#) whitepaper. If you follow one link from the many I referenced in this article, it better be this last link.

Posted in [CodeProject](#), [Troubleshooting](#), [Tutorials](#)

4 responses to “Understanding how SQL Server executes a query”


1.  *Thomas* says:
[August 1, 2013 at 1:45 pm](#)

Thanks, that's well written and explained!

2.  *Kirchner* says:
[August 2, 2013 at 12:13 pm](#)

Awesome Remus, very good indeed.

If I can make a suggestion, I'd ask you to elaborate on how workers relate to schedulers. Can workers start executing on their own or they are always driven by a scheduler?

3.  *remus* says:
[August 3, 2013 at 12:46 am](#)

@Kirchner: a scheduler has only one worker active at any time. For another worker to run, the first has to yield the scheduler and suspend itself. There are some special case exceptions when a worker goes preemptive. I recommend watching Adam Machanic's TechEd2013 session on channel9: <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2013/DBI-B404#fbid=fNNYT8rgXoe>

4.  *Ekrem Onsoy* says:
[August 8, 2013 at 6:32 am](#)

Good job!

Interested in SQL Server monitoring and configuration management?

Sign up at DBHistory.com

Recent Posts

- [SQL Injection: casting can introduce additional single quotes](#)
- [Identifying SqlConnection objects in a dump](#)
- [Understanding SQL Server Query Store](#)
- [Introducing DBHistory.com](#)
- [The cost of a transactions that has only applocks](#)

Interested in SQL Server monitoring and configuration management?

Sign up at DBHistory.com

© RUSANU CONSULTING LLC 2007-2016. All Rights Reserved [CC-BY](#)

[Entries \(RSS\)](#)