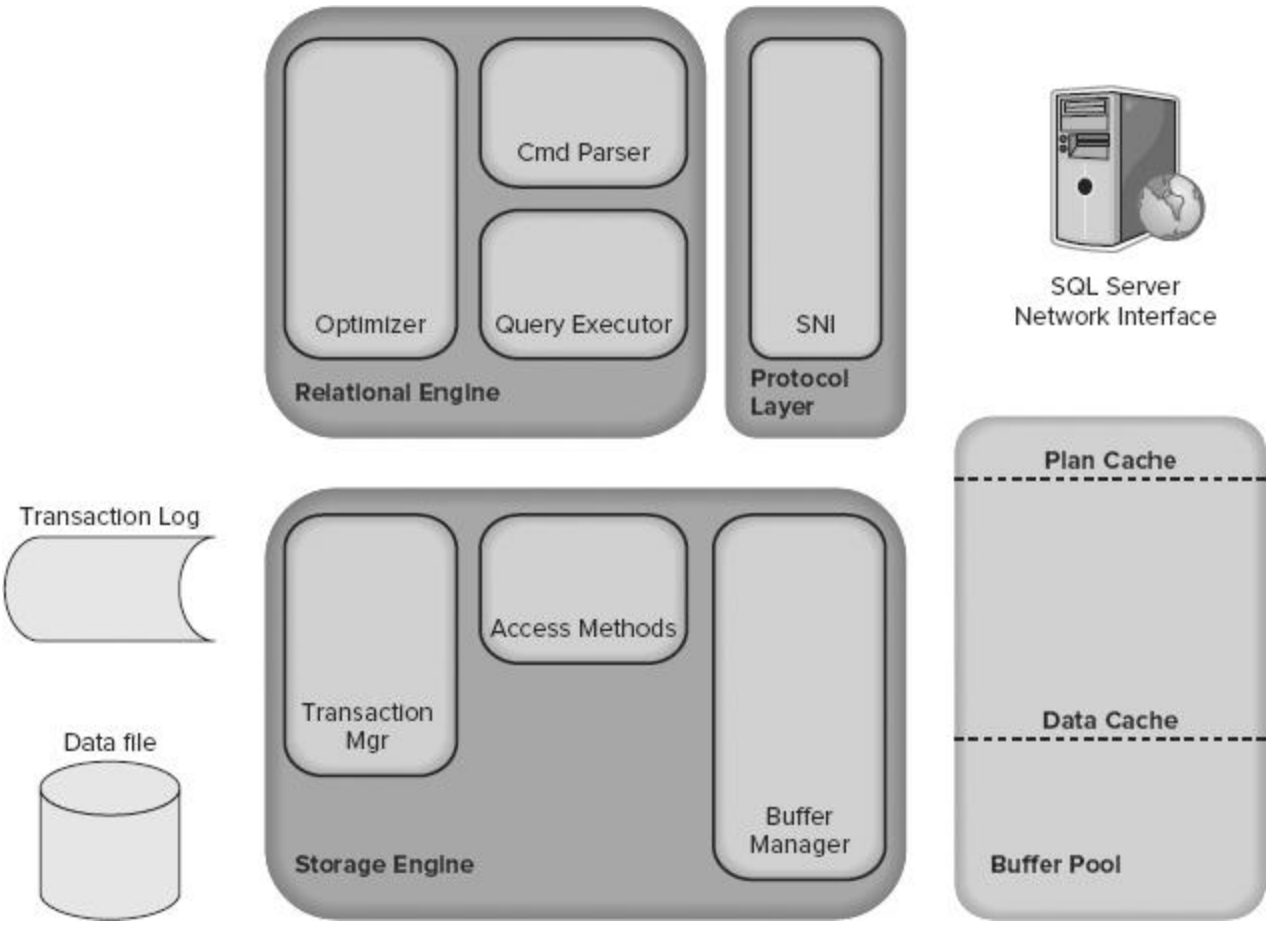
Architecture

Architecture of SQL Server ? Select Query Example



Brief Overview:

As shown in Figure 1, SQL Server is divided into two main engines: the Relational Engine and the Storage Engine.

The Relational Engine is also sometimes called the query processor because its primary function is query optimization and execution. It contains a Command Parser to check query syntax and prepare query trees; a Query Optimizer that is arguably the crown jewel of any database system; and a Query Executor responsible for execution.

The Storage Engine is responsible for managing all I/O to the data, and it contains the Access Methods code, which handles I/O requests for rows, indexes, pages, allocations and row versions; and a Buffer Manager, which deals with SQL Server's main memory consumer, the

buffer pool. It also contains a Transaction Manager, which handles the locking of data to maintain isolation (ACID properties) and manages the transaction log.

The other major component you need to know about before getting into the query life cycle is the buffer pool, which is the largest consumer of memory in SQL Server. The buffer pool contains all the different caches in SQL Server, including the plan cache and the data cache, which is covered as the sections follow the query through its life cycle.

SQL Server Network Interface

The SQL Server Network Interface (SNI) is a protocol layer that establishes the network connection between the client and the server. It consists of a set of APIs that are used by both the database engine and the SQL Server Native Client (SNAC). SNI replaces the net-libraries found in SQL Server 2000 and the Microsoft Data Access Components (MDAC), which are included with Windows.

SNI isn't configurable directly; you just need to configure a network protocol on the client and the server. SQL Server has support for the following protocols:

- Shared memory — Simple and fast, shared memory is the default protocol used to connect from a client running on the same computer as SQL Server. It can only be used locally, has no configurable properties, and is always tried first when connecting from the local machine.

- TCP/IP — This is the most commonly used access protocol for SQL Server. It enables you to connect to SQL Server by specifying an IP address and a port number. Typically, this happens automatically when you specify an instance to connect to. Your internal name resolution system resolves the hostname part of the instance name to an IP address, and either you connect to the default TCP port number 1433 for default instances or the SQL Browser service will find the right port for a named instance using UDP port 1434.

- Named Pipes — TCP/IP and Named Pipes are comparable protocols in the architectures in which they can be used. Named Pipes was developed for local area networks (LANs) but it can be inefficient across slower networks such as wide area networks (WANs).

Regardless of the network protocol used, once the connection is established SNI creates a secure connection to a TDS endpoint (described next) on the server, which is then used to send requests and receive data. For the purpose here of following a query through its life cycle, you're sending the SELECT statement and waiting to receive the result set.

Tabular Data Stream (TDS) Endpoints

TDS is a Microsoft-proprietary protocol originally designed by Sybase that is used to interact with a database server. Once a connection has been made using a network protocol such as TCP/IP, a link is established to the relevant TDS endpoint that then acts as the communication point between the client and the server.

There is one TDS endpoint for each network protocol and an additional one reserved for use by the dedicated administrator connection (DAC). Once connectivity is established, TDS messages are used to communicate between the client and the server.

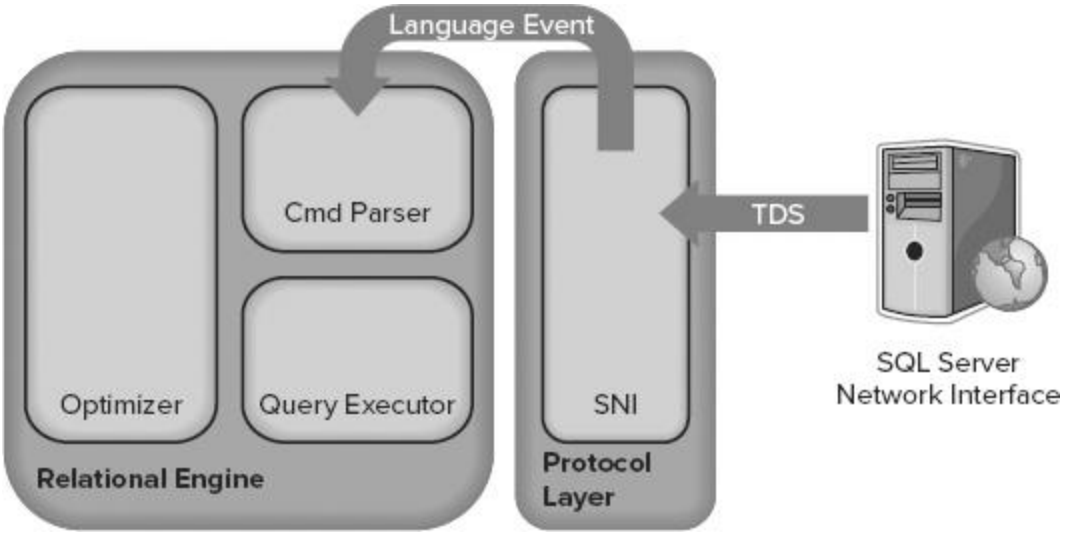
The SELECT statement is sent to the SQL Server as a TDS message across a TCP/IP connection (TCP/IP is the default protocol).

Protocol Layer

When the protocol layer in SQL Server receives your TDS packet, it has to reverse the work of the SNI at the client and unwrap the packet to find out what request it contains. The protocol layer is also responsible for packaging results and status messages to send back to the client as TDS messages.

Our SELECT statement is marked in the TDS packet as a message of type "SQL Command," so it's passed on to the next component, the Query Parser, to begin the path toward execution.

Figure 2 shows where our query has gone so far. At the client, the statement was wrapped in a TDS packet by the SQL Server Network Interface and sent to the protocol layer on the SQL Server where it was unwrapped, identified as a SQL Command, and the code sent to the Command Parser by the SNI.



Command Parser

The Command Parser's role is to handle T-SQL language events. It first checks the syntax and returns any errors back to the protocol layer to send to the client.

If the syntax is valid, then the next step is to generate a query plan or find an existing plan. A query plan contains the details about how SQL Server is going to execute a piece of code. It is commonly referred to as an execution plan.

To check for a query plan, the Command Parser generates a hash of the T-SQL and checks it against the plan cache to determine whether a suitable plan already exists. The plan cache is an

area in the buffer pool used to cache query plans. If it finds a match, then the plan is read from cache and passed on to the Query Executor for execution.

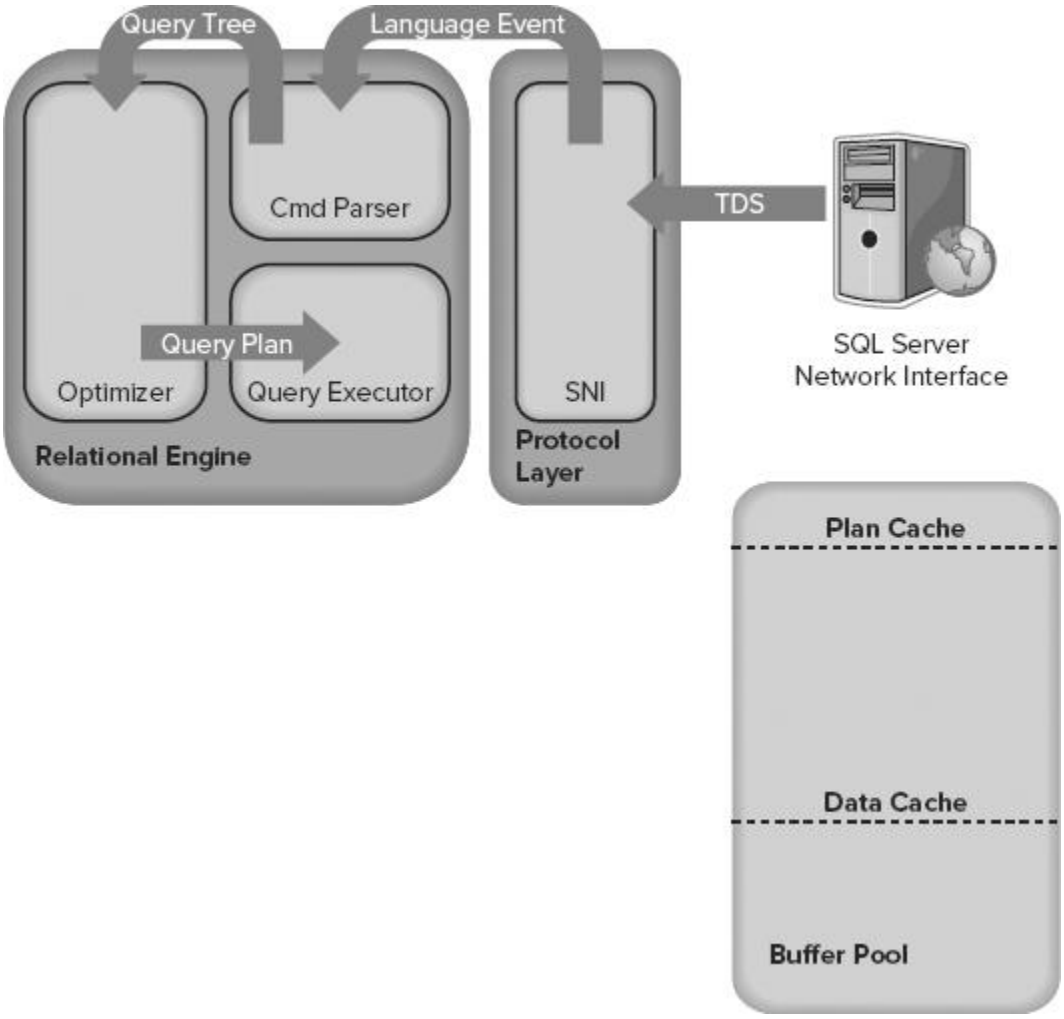
The following section explains what happens if it doesn't find a query plan.

Plan Cache

Creating execution plans can be time consuming and resource intensive, so it makes sense that if SQL Server has already found a good way to execute a piece of code that it should try to reuse it for subsequent requests.

If no cached plan is found, then the Command Parser generates a query tree based on the T-SQL. A query tree is an internal structure whereby each node in the tree represents an operation in the query that needs to be performed. This tree is then passed to the Query Optimizer to process. Our basic query didn't have an existing plan so a query tree was created and passed to the Query Optimizer.

Figure 3 shows the plan cache added to the diagram, which is checked by the Command Parser for an existing query plan. Also added is the query tree output from the Command Parser being passed to the optimizer because nothing was found in cache for our query.



Query Optimizer

The Query Optimizer is the most prized possession of the SQL Server team and one of the most complex and secretive parts of the product. Fortunately, it's only the low-level algorithms and source code that are so well protected (even within Microsoft), and research and observation can reveal how the Optimizer works.

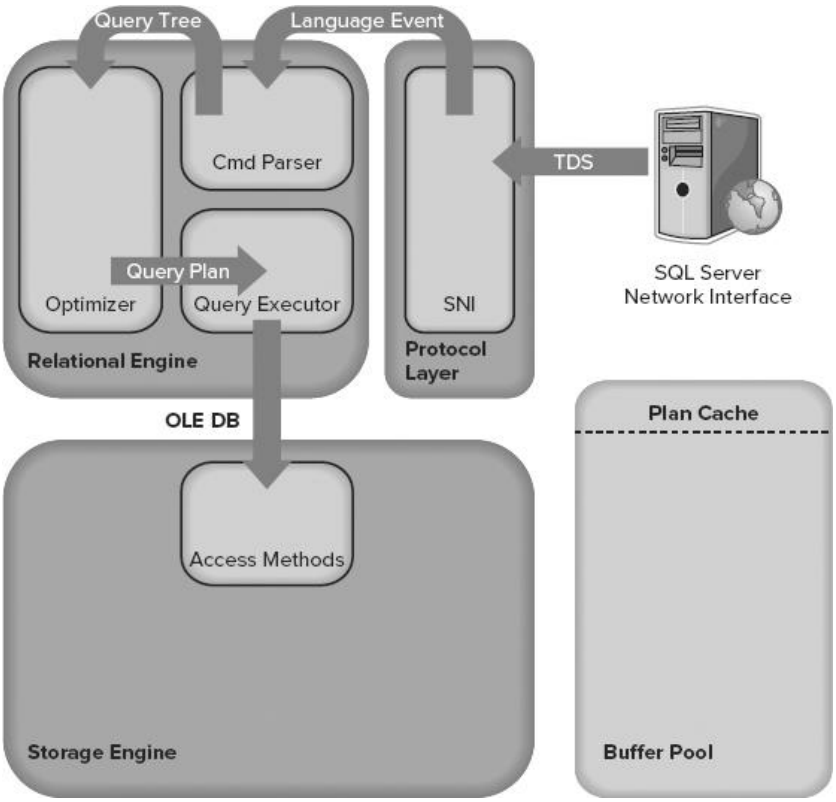
It is what's known as a "cost-based" optimizer, which means that it evaluates multiple ways to execute a query and then picks the method that it deems will have the lowest cost to execute. This "method" of executing is implemented as a query plan and is the output from the Query Optimizer.

Query Executor

The Query Executor's job is self-explanatory; it executes the query. To be more specific, it executes the query plan by working through each step it contains and interacting with the Storage Engine to retrieve or modify data.

The SELECT query needs to retrieve data, so the request is passed to the Storage Engine through an OLE DB (an API designed by Microsoft, allows accessing data from a variety of sources in a uniform manner.) interface to the Access Methods.

Below figure shows the addition of the query plan as the output from the Optimizer being passed to the Query Executor. Also introduced is the Storage Engine, which is interfaced by the Query Executor via OLE DB to the Access Methods (coming up next).



Access Methods

Access Methods is a collection of code that provides the storage structures for your data and indexes, as well as the interface through which data is retrieved and modified. It contains all the code to retrieve data but it doesn't actually perform the operation itself; it passes the request to the Buffer Manager.

Suppose our SELECT statement needs to read just a few rows that are all on a single page. The Access Methods code will ask the Buffer Manager to retrieve the page so that it can prepare an OLE DB rowset to pass back to the Relational Engine.

Buffer Manager

The Buffer Manager, as its name suggests, manages the buffer pool, which represents the majority of SQL Server's memory usage. If you need to read some rows from a page (you'll look at writes when we look at an UPDATE query), the Buffer Manager checks the data cache in the buffer pool to see if it already has the page cached in memory. If the page is already cached, then the results are passed back to the Access Methods.

If the page isn't already in cache, then the Buffer Manager gets the page from the database on disk, puts it in the data cache, and passes the results to the Access Methods.

The key point to take away from this is that you only ever work with data in memory. Every new data read that you request is first read from disk and then written to memory (the data cache) before being returned as a result set.

This is why SQL Server needs to maintain a minimum level of free pages in memory; you wouldn't be able to read any new data if there were no space in cache to put it first.

The Access Methods code determines the amount of pages needed by the SELECT query , so it asked the Buffer Manager to get it. The Buffer Manager checked whether it already had it in the data cache, and then loaded it from disk into the cache when it couldn't find it.

Data Cache

The data cache is usually the largest part of the buffer pool; therefore, it's the largest memory consumer within SQL Server. It is here that every data page that is read from disk is written to before being used.

The **sys.dm_os_buffer_descriptors** DMV contains one row for every data page currently held in cache. You can use this script to see how much space each database is using in the data cache:

```
SELECT count(*)*8/1024 AS 'Cached Size (MB)'
  ,CASE database_id
      WHEN 32767 THEN 'ResourceDb'
      ELSE db_name(database_id)
      END AS 'Database'
```

```
FROM sys.dm_os_buffer_descriptors
GROUP BY db_name(database_id),database_id
ORDER BY 'Cached Size (MB)' DESC
```

The output will look something like this (with your own databases, obviously):

Cached Size (MB)	Database
3287	People
34	tempdb
12	ResourceDb
4	msdb

In this example, the People database has 3,287MB of data pages in the data cache.

The amount of time that pages stay in data cache is determined by a least recently used (LRU) policy.

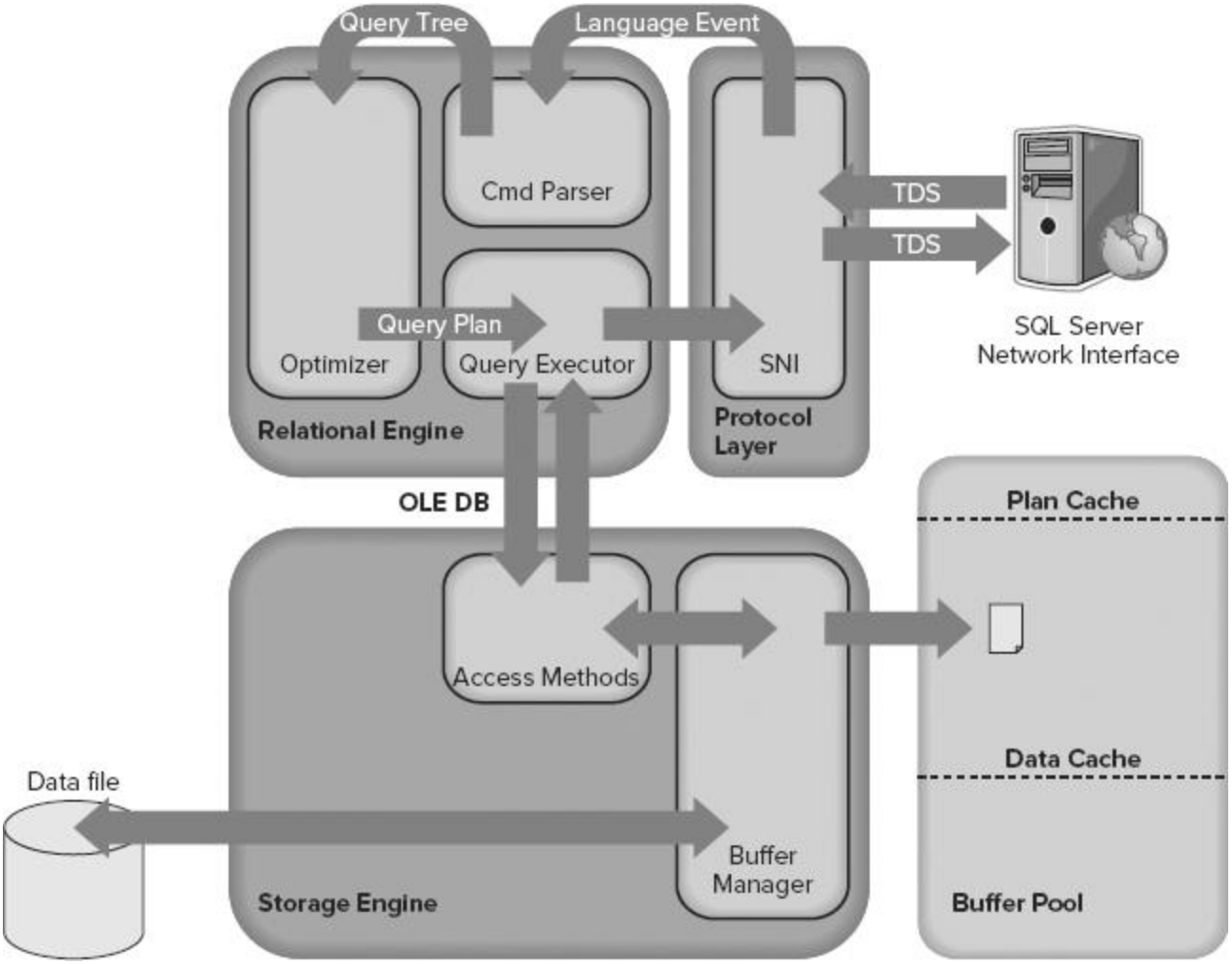
The header of each page in cache stores details about the last two times it was accessed, and a periodic scan through the cache examines these values. A counter is maintained that is decremented if the page hasn't been accessed for a while; and when SQL Server needs to free up some cache, the pages with the lowest counter are flushed first.

The process of "aging out" pages from cache and maintaining an available amount of free cache pages for subsequent use can be done by any worker thread after scheduling its own I/O or by the lazy writer process, covered later in the section "Lazy Writer."

You can view how long SQL Server expects to be able to keep a page in cache by looking at the MSSQL$<instance>:Buffer Manager\Page Life Expectancy counter in Performance Monitor. Page life expectancy (PLE) is the amount of time, in seconds, that SQL Server expects to be able to keep a page in cache.

Under memory pressure, data pages are flushed from cache far more frequently. Microsoft has a long standing recommendation for a minimum of 300 seconds for PLE but a good value is generally considered to be 1000s of seconds these days. Exactly what your acceptable threshold should be is variable depending on your data usage, but more often than not, you'll find servers with either 1000s of seconds PLE or a lot less than 300, so it's usually easy to spot a problem.

The database page read to serve the result set for our SELECT query is now in the data cache in the buffer pool and will have an entry in the sys.dm_os_buffer_descriptors DMV. Now that the Buffer Manager has the result set, it's passed back to the Access Methods to make its way to the client.



1. The SQL Server Network Interface (SNI) on the client established a connection to the SNI on the SQL Server using a network protocol such as TCP/IP. It then created a connection to a TDS endpoint over the TCP/IP connection and sent the SELECT statement to SQL Server as a TDS message.
2. The SNI on the SQL Server unpacked the TDS message, read the SELECT statement, and passed a "SQL Command" to the Command Parser.
3. The Command Parser checked the plan cache in the buffer pool for an existing, usable query plan that matched the statement received. When it didn't find one, it created a query tree based on the SELECT statement and passed it to the Optimizer to generate a query plan.
4. The Optimizer generated a "zero cost" or "trivial" plan in the pre-optimization phase because the statement was so simple. The query plan created was then passed to the Query Executor for execution.
5. At execution time, the Query Executor determined that data needed to be read to complete the query plan so it passed the request to the Access Methods in the Storage Engine via an OLE DB interface.
6. The Access Methods needed to read a page from the database to complete the request from the Query Executor and asked the Buffer Manager to provision the data page.

7. The Buffer Manager checked the data cache to see if it already had the page in cache. It wasn't in cache so it pulled the page from disk, put it in cache, and passed it back to the Access Methods.
8. Finally, the Access Methods passed the result set back to the Relational Engine to send to the client.

Architecture of SQL Server ? Update Query Example

This example takes a look at a simple UPDATE query that modifies the data that was read in the previous example.

The good news is that the process is exactly the same as the process for the SELECT statement you just looked at until you get to the Access Methods.

The Access Methods need to make a data modification this time, so before the I/O request is passed on, the details of the change need to be persisted to disk. That is the job of the Transaction Manager.

Transaction Manager

The Transaction Manager has two components that are of interest here: a Lock Manager and a Log Manager. The Lock Manager is responsible for providing concurrency to the data, and it delivers the configured level of isolation by using locks.

NOTE
The Lock Manager is also employed during the SELECT query life cycle covered earlier, but it would have been a distraction; it is mentioned here because it's part of the Transaction Manager.
The real item of interest here is actually the Log Manager. The Access Methods code requests that the changes it wants to make are logged, and the Log Manager writes the changes to the transaction log. This is called write-ahead logging (WAL).

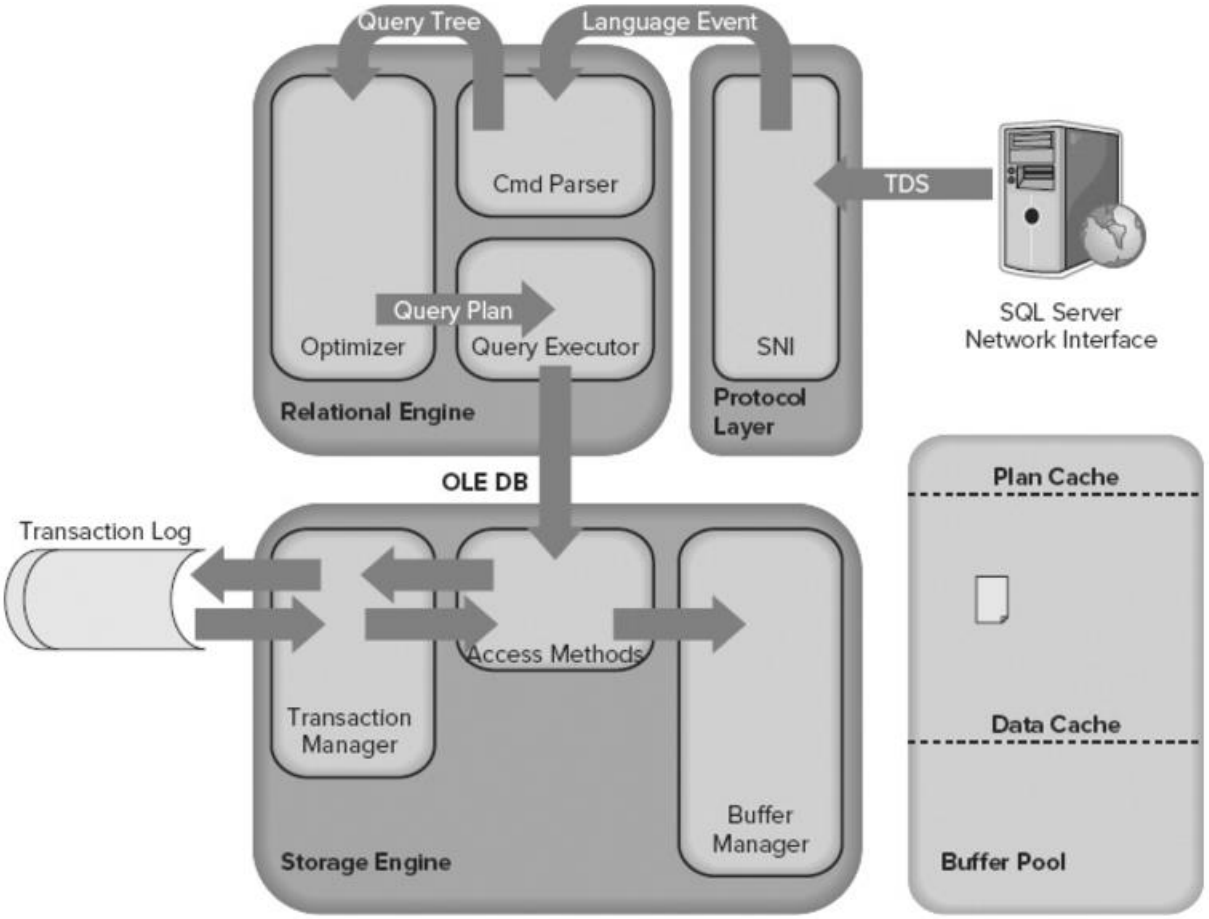
Writing to the transaction log is the only part of a data modification transaction that always needs a physical write to disk because SQL Server depends on being able to reread that change in the event of system failure (you'll learn more about this in the "Recovery" section coming up).

What's actually stored in the transaction log isn't a list of modification statements but only details of the page changes that occurred as the result of a modification statement. This is all that SQL Server needs in order to undo any change, and why it's so difficult to read the contents of a transaction log in any meaningful way, although you can buy a third-party tool to help.

Getting back to the UPDATE query life cycle, the update operation has now been logged. The actual data modification can only be performed when confirmation is received that the operation has been physically written to the transaction log. This is why transaction log performance is so crucial.

Once confirmation is received by the Access Methods, it passes the modification request on to the Buffer Manager to complete.

Below figure shows the Transaction Manager, which is called by the Access Methods and the transaction log, which is the destination for logging our update. The Buffer Manager is also in play now because the modification request is ready to be completed.



Buffer Manager

The page that needs to be modified is already in cache, so all the Buffer Manager needs to do is modify the page required by the update as requested by the Access Methods. The page is modified in the cache, and confirmation is sent back to Access Methods and ultimately to the client.

The key point here (and it's a big one) is that the UPDATE statement has changed the data in the data cache, not in the actual database file on disk. This is done for performance reasons, and the page is now what's called a dirty page because it's different in memory from what's on disk.

It doesn't compromise the durability of the modification as defined in the ACID properties because you can re-create the change using the transaction log if, for example, you suddenly lost power to the server, and therefore anything in physical RAM (i.e., the data cache). How and when the dirty page makes its way into the database file is covered in the next section.

Figure 7 shows the completed life cycle for the update. The Buffer Manager has made the modification to the page in cache and has passed confirmation back up the chain. The database data file was not accessed during the operation, as you can see in the diagram.