COVERS SQL SERVER 2012

Join the discussion @ p2p.wrox.com

# Professional SQL Server® 2012 Internals and Troubleshooting

Christian Bolton, Justin Langford, Glenn Berry, Gavin Payne, Amit Banerjee, Rob Farley

# CONTENTS

# PART I

# Internals

- **CHAPTER 1:** SQL Server Architecture
- **CHAPTER 2:** Demystifying Hardware
- **CHAPTER 3:** Understanding Memory
- **CHAPTER 4:** Storage Systems
- **CHAPTER 5:** Query Processing and Execution
- **CHAPTER 6:** Locking and Concurrency
- **CHAPTER 7:** Latches and Spinlocks
- **CHAPTER 8:** Knowing Tempdb

# Chapter 1

# SQL Server Architecture

### WHAT'S IN THIS CHAPTER?

- Understanding database transactions and the ACID properties
- Architectural components used to fulfill a read request
- Architectural components used to fulfill an update request
- Database recovery and the transaction log
- Dirty pages, checkpoints, and the lazy writer
- Where the SQLOS fits in and why it's needed

## WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at www.wrox.com/remtitle.cgi?isbn=1118177657 on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.

# INTRODUCTION

A basic grasp of SQL Server's database engine architecture is fundamental to intelligently approach troubleshooting a problem, but selecting the important bits to learn about can be challenging, as SQL Server is such a complex piece of software. This chapter distills the core architecture of SQL Server, putting the most important components into the

context of executing a simple query to help you understand the fundamentals of the core engine.

You will learn how SQL Server deals with your network connection, unravels what you're asking it to do, decides how it will execute your request, and finally how data is retrieved and modified on your behalf.

You will also discover when the transaction log is used and how it's affected by the configured recovery model; what happens when a checkpoint occurs and how you can influence the frequency; and what the lazy writer does.

The chapter starts by defining a "transaction" and outlining the database system's requirements to reliably process them. You'll then look at the life cycle of a simple query that reads data, looking at the components employed to return a result set, before examining how the process differs when data needs to be modified.

Finally, you'll learn about the components and terminology that support the recovery process in SQL Server, and the SQLOS "framework" that consolidates a lot of the low-level functions required by many SQL Server components.

> **NOTE**
>
> *Coverage of some areas of the life cycle described in this chapter is intentionally shallow in order to keep the flow manageable; where that's the case, you are directed to the chapter or chapters that cover the topic in more depth.*

# DATABASE TRANSACTIONS

A *transaction* is a unit of work in a database that typically contains several commands that read from and write to the database. The most well-known feature of a transaction is that it must complete all the commands in their entirety or none of them. This feature, called *atomicity*, is just one of four properties defined in the early days of database theory

as requirements for a database transaction, collectively known as ACID properties.

# ACID Properties

The four required properties of a database transaction are atomicity, consistency, isolation, and durability.

## Atomicity

Atomicity means that *all* the effects of the transaction must complete successfully or the changes are rolled back. A classic example of an atomic transaction is a withdrawal from an ATM machine; the machine must both dispense the cash *and* debit your bank account. Either of those actions completing independently would cause a problem for either you or the bank.

## Consistency

The consistency requirement ensures that the transaction cannot break the integrity rules of the database; it must leave the database in a consistent state. For example, your system might require that stock levels cannot be a negative value, a spare part cannot exist without a parent object, or the data in a sex field must be male or female. In order to be consistent, a transaction must not break any of the constraints or rules defined for the data.

## Isolation

Isolation refers to keeping the changes of incomplete transactions running at the same time separate from one another. Each transaction must be entirely self-contained, and changes it makes must not be readable by any other transaction, although SQL Server does allow you to control

the degree of isolation in order to find a balance between business and performance requirements.

## Durability

Once a transaction is committed, it must persist even if there is a system failure — that is, it must be durable. In SQL Server, the information needed to replay changes made in a transaction is written to the transaction log before the transaction is considered to be committed.

# SQL Server Transactions

There are two types of transactions in SQL Server, *implicit* and *explicit*, and they are differentiated only by the way they are created.

*Implicit* transactions are used automatically by SQL Server to guarantee the ACID properties of single commands. For example, if you wrote an update statement that modified 10 rows, SQL Server would run it as an implicit transaction so that the ACID properties would apply, and all 10 rows would be updated or none of them would.

*Explicit* transactions are started by using the `BEGIN TRANSACTION` T-SQL command and are stopped by using the `COMMIT TRANSACTION` or `ROLLBACK TRANSACTION` commands.

Committing a transaction effectively means making the changes within the transaction permanent, whereas rolling back a transaction means undoing all the changes that were made within the transaction. Explicit transactions are used to group together changes to which you want to apply the ACID properties as a whole, which also enables you to roll back the changes at any point if your business logic determines that you should cancel the change.

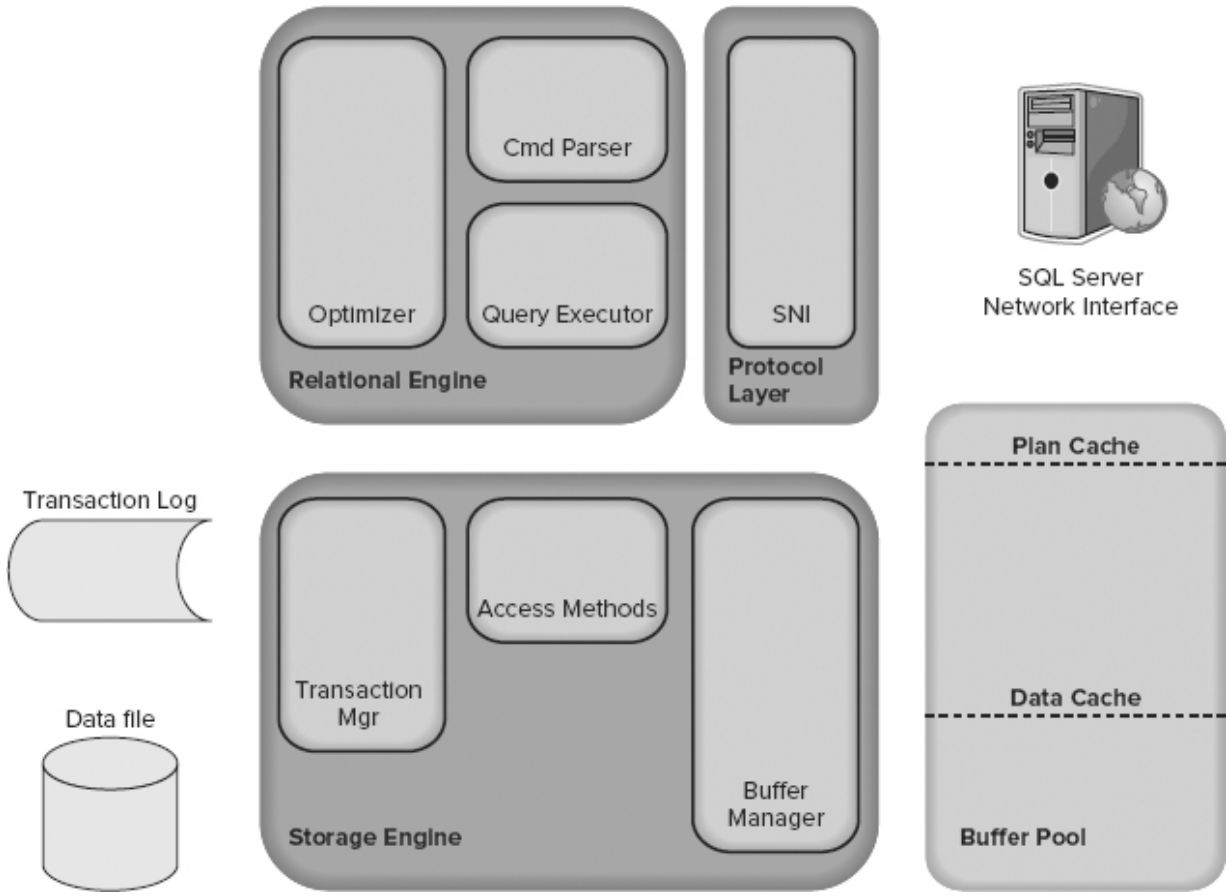
# THE LIFE CYCLE OF A QUERY

To introduce the high-level components of SQL Server's architecture, this section uses the example of a query's life cycle to put each component into context to foster your understanding and create a foundation for the rest of the book.

It looks at a basic `SELECT` query first in order to reduce the scope to that of a `READ` operation, and then introduces the additional processes involved for a query that performs an `UPDATE` operation. Finally, you'll read about the terminology and processes that SQL Server uses to implement recovery while optimizing performance.

Figure 1-1 shows the high-level components that are used within the chapter to illustrate the life cycle of a query.

**FIGURE 1-1**




# The Relational and Storage Engines

As shown in Figure 1–1, 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 Buffer Pool

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.

> **NOTE**
>
> *The buffer pool is covered in detail in Chapter 3.*

# A Basic SELECT Query

The details of the query used in this example aren't important — it's a simple `SELECT` statement with no joins, so you're just issuing a basic read request. It begins at the client, where the first component you touch is the SQL Server Network Interface (SNI).

## 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).

  To use Named Pipes you first need to enable it in SQL Server Configuration Manager (if you'll be connecting remotely) and then create a SQL Server alias, which connects to the server using Named Pipes as the protocol.

  Named Pipes uses TCP port 445, so ensure that the port is open on any firewalls between the two computers, including the Windows Firewall.
- **VIA —** Virtual Interface Adapter is a protocol that enables high-performance communications between two systems. It requires specialized hardware at both ends and a dedicated connection.

> Like Named Pipes, to use the VIA protocol you first need to enable it in SQL Server Configuration Manager and then create a SQL Server alias that connects to the server using VIA as the protocol. While SQL Server 2012 still supports the VIA protocol, it will be removed from a future version so new installations using this protocol should be avoided.

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

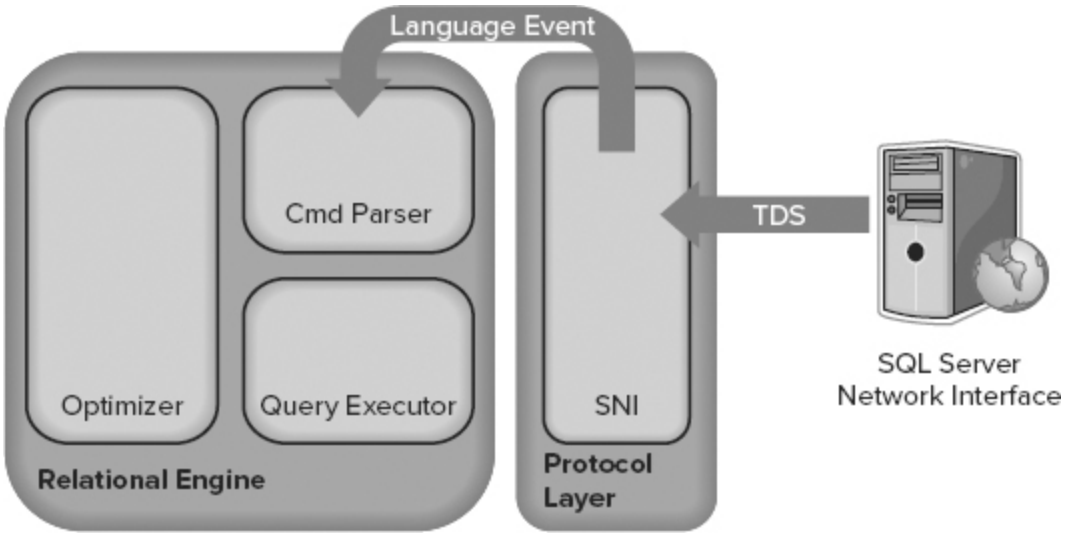
When the protocol layer in SQL Server receives your TDS packet, it has to reverse the work of the SNI at the client

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 1-2 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.

**FIGURE 1-2**




# 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 match.)

## Plan Cache

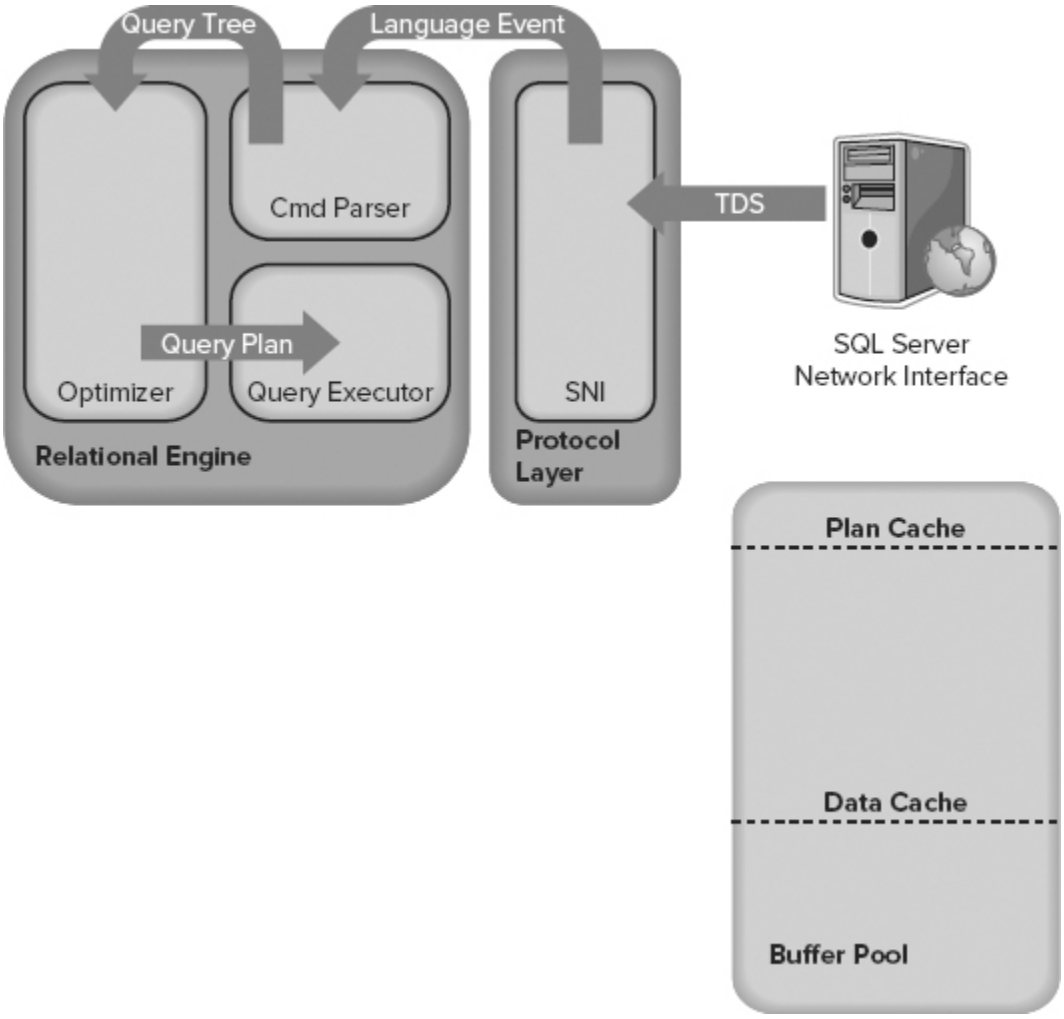
Creating execution plans can be time consuming and resource intensive, so it makes sense that if SQL Server has already found a good way to execute a piece of code that it should try to reuse it for subsequent requests.

The plan cache, part of SQL Server's buffer pool, is used to store execution plans in case they are needed later. You can read more about execution plans and plan cache in Chapters 3 and 5.

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 1-3 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.

**FIGURE 1-3**




# 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.

Based on that description, you would be forgiven for thinking that the Optimizer's job is to find the *best* query plan because that would seem like an obvious assumption. Its actual job, however, is to find a *good* plan in a reasonable amount of time, rather than the *best* plan. The optimizer's goal is most commonly described as finding the most *efficient* plan.

If the Optimizer tried to find the "best" plan every time, it might take longer to find the plan than it would to just execute a slower plan (some built-in heuristics actually ensure that it never takes longer to find a good plan than it does to just find a plan and execute it).

As well as being cost based, the Optimizer also performs multi-stage optimization, increasing the number of decisions available to find a good plan at each stage. When a good plan is found, optimization stops at that stage.

The first stage is known as *pre-optimization*, and queries drop out of the process at this stage when the statement is simple enough that there can only be one optimal plan, removing the need for additional costing. Basic queries with no joins are regarded as "simple," and plans produced as such have zero cost (because they haven't been costed) and are referred to as *trivial plans*.

The next stage is where optimization actually begins, and it consists of three search phases:

- **Phase 0 —** During this phase the optimizer looks at nested loop joins and won't consider parallel operators (parallel means executing across multiple processors and is covered in Chapter 5).

  The optimizer will stop here if the cost of the plan it has found is < 0.2. A plan generated at this phase is known as a *transaction processing*, or *TP*, plan.

- **Phase 1 —** Phase 1 uses a subset of the possible optimization rules and looks for common patterns for which it already has a plan.

The optimizer will stop here if the cost of the plan it has found is < 1.0. Plans generated in this phase are called *quick plans*.

- **Phase 2 —** This final phase is where the optimizer pulls out all the stops and is able to use all of its optimization rules. It also looks at parallelism and indexed views (if you're running Enterprise Edition).

  Completion of Phase 2 is a balance between the cost of the plan found versus the time spent optimizing. Plans created in this phase have an optimization level of "Full."

### HOW MUCH DOES IT COST?

The term *cost* doesn't translate into seconds or anything meaningful; it is just an arbitrary number used to assign a value representing the resource cost for a plan. However, its origin was a benchmark on a desktop computer at Microsoft early in SQL Server's life.

In a plan, each operator has a baseline cost, which is then multiplied by the size of the row and the estimated number of rows to get the cost of that operator — and the cost of the plan is the total cost of all the operators.

Because cost is created from a baseline value and isn't related to the speed of your hardware, any plan created will have the same cost on every SQL Server installation (like-for-like version).

The statistics that the optimizer uses to estimate the number of rows aren't covered here because they aren't relevant to the concepts illustrated in this chapter, but you can read about them in Chapter 5.

Because our SELECT query is very simple, it drops out of the process in the pre-optimization phase because the plan is obvious to the optimizer (a *trivial plan*). Now that there is a query plan, it's on to the Query Executor for execution.

# Query Executor

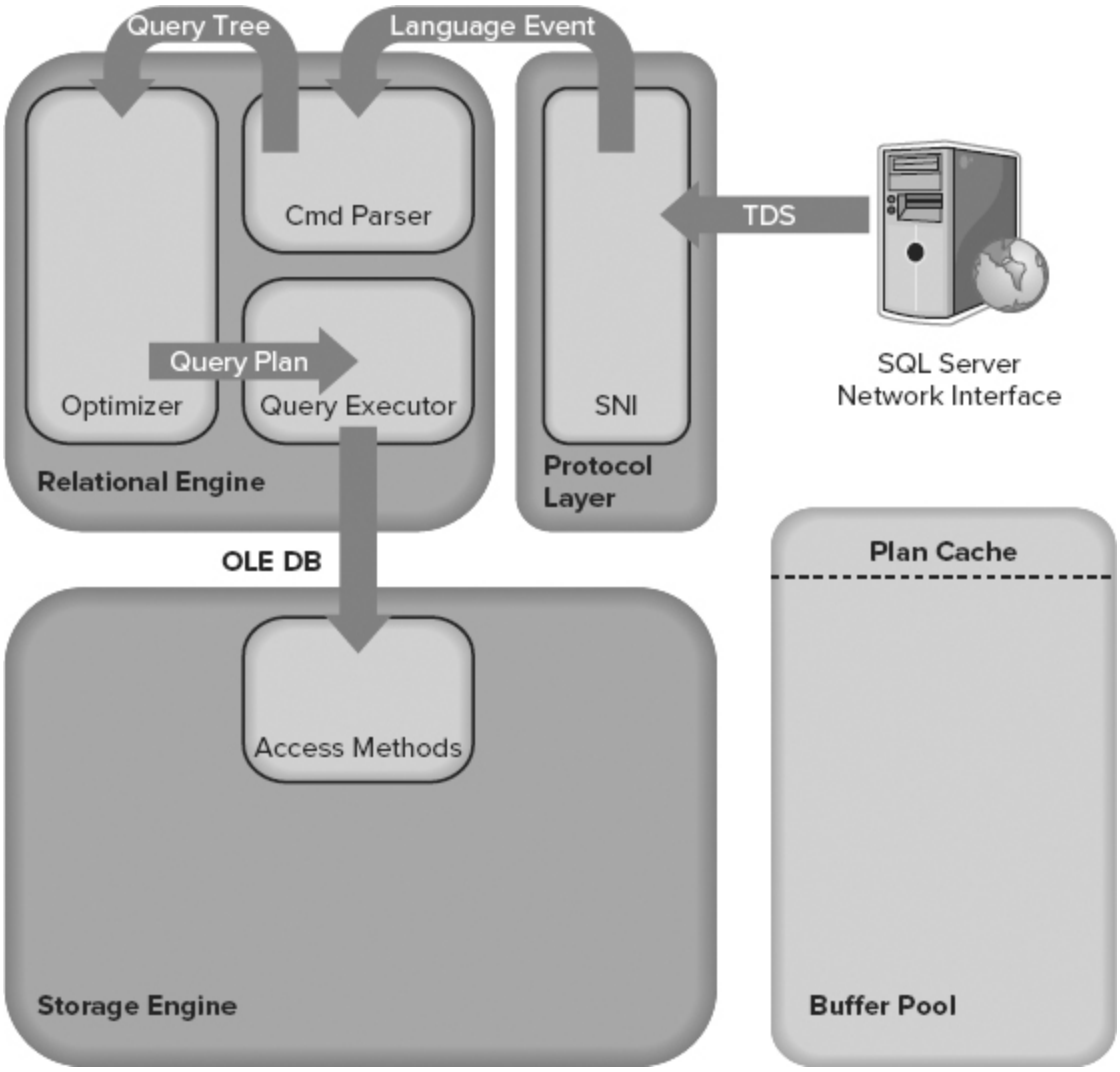
The Query Executor's job is self-explanatory; it executes the query. To be more specific, it executes the query plan by working through each step it contains and interacting with the Storage Engine to retrieve or modify data.

> **NOTE**
>
> *The interface to the Storage Engine is actually OLE DB, which is a legacy from a design decision made in SQL Server's history. The development team's original idea was to interface through OLE DB to allow different Storage Engines to be plugged in. However, the strategy changed soon after that.*
>
> *The idea of a pluggable Storage Engine was dropped and the developers started writing extensions to OLE DB to improve performance. These customizations are now core to the product; and while there's now no reason to have OLE DB, the existing investment and performance precludes any justification to change it.*

The SELECT query needs to retrieve data, so the request is passed to the Storage Engine through an OLE DB interface to the Access Methods.

Figure 1-4 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).

**FIGURE 1-4**




# 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.

> **NOTE**
>
> *The* `PAGEIOLATCH` *wait type represents the time it takes to read a data page from disk into memory. Wait types are covered later in this chapter*.

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 determined that the `SELECT` query needed a single page, 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 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.

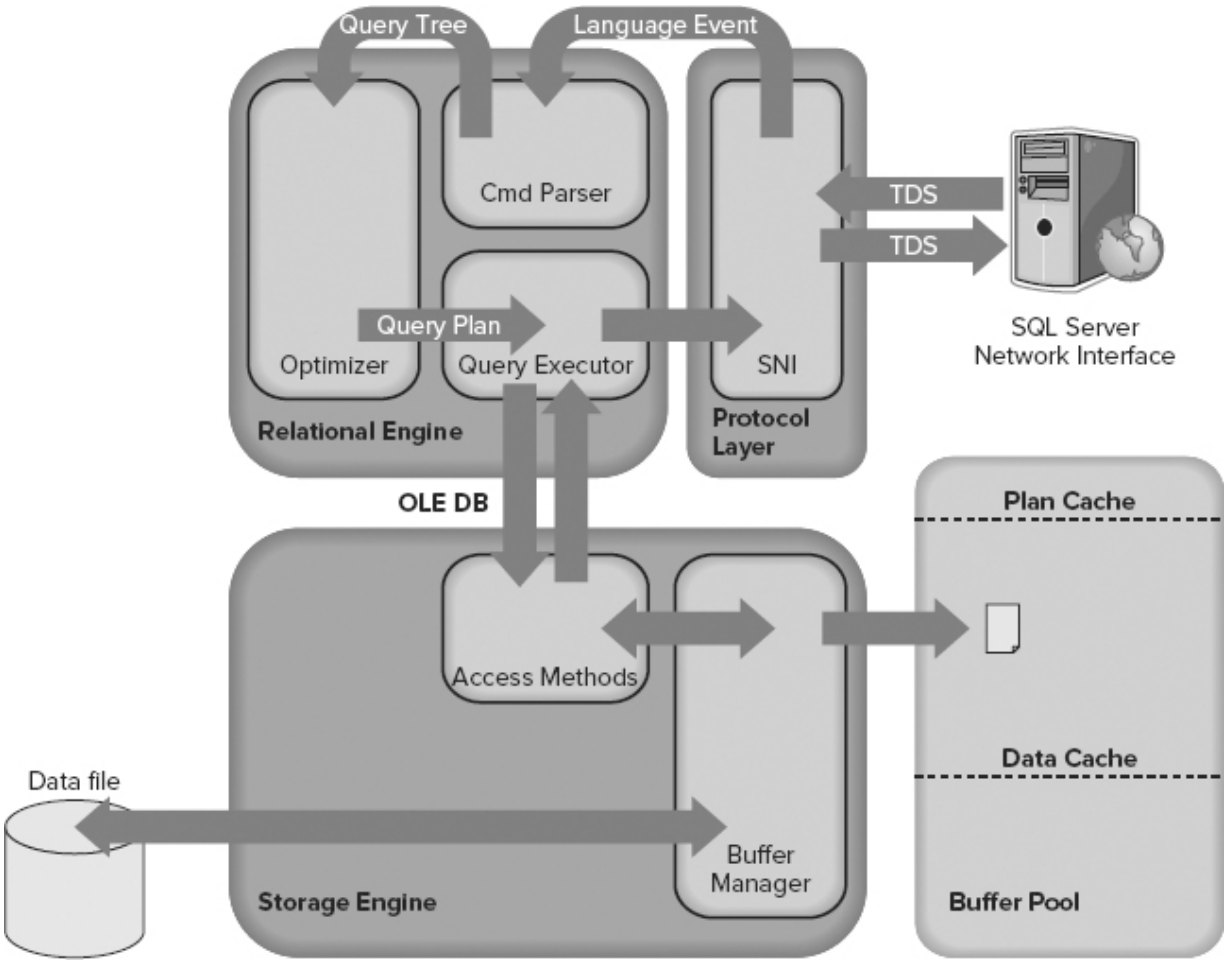
Under memory pressure, data pages are flushed from cache far more frequently. Microsoft has a long standing recommendation for a minimum of 300 seconds for PLE but a good value is generally considered to be 1000s of seconds these days. Exactly what your acceptable threshold should be is variable depending 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.

## A Basic SELECT Statement Life Cycle Summary

Figure 1-5 shows the whole life cycle of a `SELECT` query, described here:

**FIGURE 1-5**




**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.

# A Simple Update Query

Now that you understand the life cycle for a query that just reads some data, the next step is to determine what happens when you need to write data. To answer that, this section 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* (as defined in the ACID properties at the beginning of the chapter) 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, but locking is covered in depth in Chapter 6.*

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.