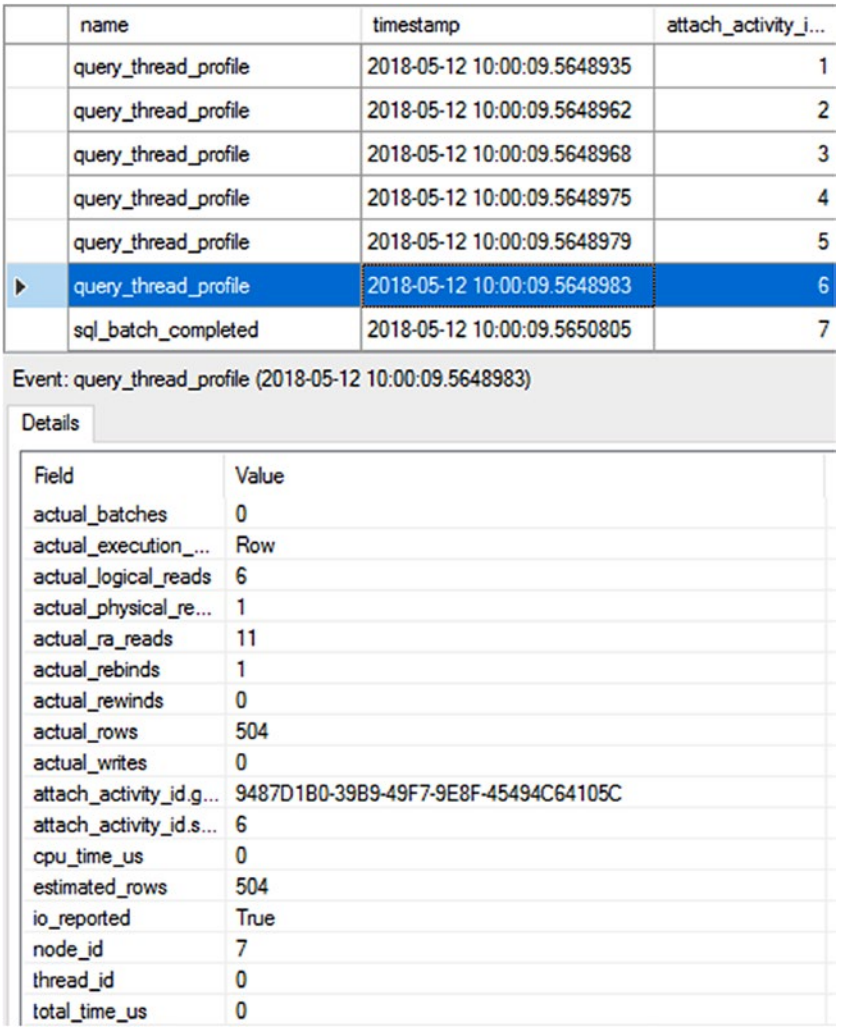
Query Thread Profiles

Mentioned earlier, the new Extended Events event `query_thread_profile` adds new functionality to the system. This event is a debug event. As mentioned in Chapter 6, the debug events should be used sparingly. However, Microsoft does advocate for the use of this event. Running it will allow you to watch live execution plans on long-running queries. However, it does more than that. It also captures row and thread counts for all operators within an execution plan at the end of the execution of that plan. It's very low cost and an easy way to capture those metrics, especially on queries that run fast where you could never really see their active row counts in a live execution plan. This is data that you get with an execution plan, but this is much more low cost than capturing a plan. This is the script for creating a session that captures the query thread profiles as well as the core query metrics:

```
CREATE EVENT SESSION QueryThreadProfile
ON SERVER
    ADD EVENT sqlserver.query_thread_profile
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.sql_batch_completed
    (WHERE (sqlserver.database_name = N'AdventureWorks2017'))
WITH (TRACK_CAUSALITY = ON)
GO
```

With this session running, if you run a small query, such as the one we used at the start of the "Execution Plan Tooling" section, the output looks like Figure 7-24.

www.EBooksWorld.ir



name	timestamp	attach_activity_i...
query_thread_profile	2018-05-12 10:00:09.5648935	1
query_thread_profile	2018-05-12 10:00:09.5648962	2
query_thread_profile	2018-05-12 10:00:09.5648968	3
query_thread_profile	2018-05-12 10:00:09.5648975	4
query_thread_profile	2018-05-12 10:00:09.5648979	5
query_thread_profile	2018-05-12 10:00:09.5648983	6
sql_batch_completed	2018-05-12 10:00:09.5650805	7

Event: query_thread_profile (2018-05-12 10:00:09.5648983)

Details

Field	Value
actual_batches	0
actual_execution_...	Row
actual_logical_reads	6
actual_physical_re...	1
actual_ra_reads	11
actual_rebinds	1
actual_rewinds	0
actual_rows	504
actual_writes	0
attach_activity_id.g...	9487D1B0-39B9-49F7-9E8F-45494C64105C
attach_activity_id.s...	6
cpu_time_us	0
estimated_rows	504
io_reported	True
node_id	7
thread_id	0
total_time_us	0

Figure 7-24. *Extended Events session showing query_thread_profile information*

You can see the details of the event including estimated rows, actual rows, and a lot of the other information we frequently go to execution plans for as part of evaluating statistics and index use among other things. You can now capture this information on the fly for your queries without having to go through the much costlier process of capturing execution plans. Just remember, this is not a zero-cost operation. It's just a lower-cost operation. It's also not going to replace all the uses of an execution plan because the plans show so much more than threads, duration, and row counts.

www.EBooksWorld.ir

Query Resource Cost

Even though the execution plan for a query provides a detailed processing strategy and the estimated relative costs of the individual steps involved, if it's an estimated plan, it doesn't provide the actual cost of the query in terms of CPU usage, reads/writes to disk, or query duration. While optimizing a query, you may add an index to reduce the relative cost of a step. This may adversely affect a dependent step in the execution plan, or sometimes it may even modify the execution plan itself. Thus, if you look only at the estimated execution plan, you can't be sure that your query optimization benefits the query as a whole, as opposed to that one step in the execution plan. You can analyze the overall cost of a query in different ways.

You should monitor the overall cost of a query while optimizing it. As explained previously, you can use Extended Events to monitor the `duration`, `cpu`, `reads`, and `writes` information for the query. Extended Events is an extremely efficient mechanism for gathering metrics. You should plan on taking advantage of this fact and use this mechanism to gather your query performance metrics. Just understand that collecting this information leads to large amounts of data that you will have to find a place to maintain within your system.

There are other ways to collect performance data that are more immediate and easily accessible than Extended Events. In addition to the ones I detail next, don't forget that we have the DMOs, such as `sys.dm_exec_query_stats` and `sys.dm_exec_procedure_stats`, and the Query Store system views and reports, `sys.query_store_runtime_stats` and `sys.query_store_wait_stats`.

Client Statistics

Client statistics capture execution information from the perspective of your machine as a client of the server. This means that any times recorded include the time it takes to transfer data across the network, not merely the time involved on the SQL Server machine. To use them, simply select Query ➤ Include Client Statistics. Now, each time you run a query, a limited set of data is collected including the execution time, the number of rows affected, the round-trips to the server, and more. Further, each execution of the query is displayed separately on the Client Statistics tab, and a column aggregating the multiple executions shows the averages for the data collected. The

www.EBooksWorld.ir

statistics will also show whether a time or count has changed from one run to the next, showing up as arrows, as shown in Figure 7-13. For example, consider this query:

```
SELECT TOP 100
    p.Name,
    p.ProductNumber
FROM Production.Product p;
```

The client statistics information for the query should look something like those shown in Figure 7-25.

	Trial 2	Trial 1	Average
Client Execution Time	10:55:41	10:55:38	
Query Profile Statistics			
Number of INSERT, DELETE and UPDATE statements	0 →	0 →	0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0 →	0 →	0.0000
Number of SELECT statements	2 ↑	1 →	1.5000
Rows returned by SELECT statements	101 ↑	100 →	100.5000
Number of transactions	0 →	0 →	0.0000
Network Statistics			
Number of server roundtrips	2 ↑	1 →	1.5000
TDS packets sent from client	2 ↑	1 →	1.5000
TDS packets received from server	3 ↑	2 →	2.5000
Bytes sent from client	240 ↑	182 →	211.0000
Bytes received from server	4929 ↑	4892 →	4910.5000
Time Statistics			
Client processing time	1 ↓	2 →	1.5000
Total execution time	1 ↓	3 →	2.0000
Wait time on server replies	0 ↓	1 →	0.5000

Figure 7-25. *Client statistics*

Although capturing client statistics can be a useful way to gather data, it's a limited set of data, and there is no way to show how one execution is different from another. You could even run a completely different query, and its data would be mixed in with the others, making the averages useless. If you need to, you can reset the client statistics. Select the Query menu and then the Reset Client Statistics menu item.

www.EBooksWorld.ir

Execution Time

Both Duration and CPU represent the time factor of a query. To obtain detailed information on the amount of time (in milliseconds) required to parse, compile, and execute a query, use SET STATISTICS TIME as follows:

```
SET STATISTICS TIME ON;
GO
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product p
        ON sod.ProductID = p.ProductID
WHERE sod.LineTotal > 1000;
GO
SET STATISTICS TIME OFF;
GO
```

The output of STATISTICS TIME for the preceding SELECT statement is shown here:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 9 ms.

(32101 row(s) affected)

 SQL Server Execution Times:
   CPU time = 156 ms,  elapsed time = 400 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

The CPU time = 156 ms part of the execution times represents the CPU value provided by Extended Events. Similarly, the corresponding Elapsed time = 400 ms represents the Duration value provided by the other mechanisms.

www.EBooksWorld.ir

A 0 ms parse and 9 ms compile time signifies that the optimizer has to parse the query first for syntax and then compile it to produce the execution plan.

STATISTICS IO

As discussed in the "Identifying Costly Queries" section earlier in the chapter, the number of reads in the `Reads` column is frequently the most significant cost factor among `duration`, `cpu`, `reads`, and `writes`. The total number of reads performed by a query consists of the sum of the number of reads performed on all tables involved in the query. The reads performed on the individual tables may vary significantly, depending on the size of the result set requested from the individual table and the indexes available.

To reduce the total number of reads, it will be useful to find all the tables accessed in the query and their corresponding number of reads. This detailed information helps you concentrate on optimizing data access on the tables with a large number of reads. The number of reads per table also helps you evaluate the impact of the optimization step (implemented for one table) on the other tables referred to in the query.

In a simple query, you determine the individual tables accessed by taking a close look at the query. This becomes increasingly difficult the more complex the query becomes. In the case of stored procedures, database views, or functions, it becomes more difficult to identify all the tables actually accessed by the optimizer. You can use `STATISTICS IO` to get this information, irrespective of query complexity.

To turn `STATISTICS IO` on, navigate to Query ➤ Query Options ➤ Advanced ➤ Set Statistics IO in Management Studio. You may also get this information programmatically as follows:

```
SET STATISTICS IO ON;
GO
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
```


www.EBooksWorld.ir

```
    JOIN Production.Product p
        ON sod.ProductID = p.ProductID
WHERE sod.SalesOrderID = 71856;
GO
SET STATISTICS IO OFF;
GO
```

If you run this query and look at the execution plan, it consists of three clustered index seeks with two loop joins. If you remove the `WHERE` clause and run the query again, you get a set of scans and some hash joins. That's an interesting fact—but you don't know how it affects the query I/O usage! You can use `SET STATISTICS IO` as shown previously to compare the cost of the query (in terms of logical reads) between the two processing strategies used by the optimizer.

You get following `STATISTICS IO` output when the query uses the hash join:

```
(121317 row(s) affected)
Table 'Workfile'. Scan count 0, logical reads 0...
Table 'Worktable'. Scan count 0, logical reads 0...
Table 'SalesOrderDetail'. Scan count 1, logical reads 1248...
Table 'SalesOrderHeader'. Scan count 1, logical reads 689...
Table 'Product'. Scan count 1, logical reads 6...

(1 row(s) affected)
```

Now when you add back in the `WHERE` clause to appropriately filter the data, the resultant `STATISTICS IO` output turns out to be this:

```
(2 row(s) affected)
Table 'Product'. Scan count 0, logical reads 4...
Table 'SalesOrderDetail'. Scan count 1, logical reads 3...
Table 'SalesOrderHeader'. Scan count 0, logical reads 3...

(1 row(s) affected)
```

Logical reads for the `SalesOrderDetail` table have been cut from 1,248 to 3 because of the index seek and the loop join. It also hasn't significantly affected the data retrieval cost of the `Product` table.

www.EBooksWorld.ir

While interpreting the output of `STATISTICS IO`, you mostly refer to the number of logical reads. The number of physical reads and read-ahead reads will be nonzero when the data is not found in the memory, but once the data is populated in memory, the physical reads and read-ahead reads will tend to be zero.

There is another advantage to knowing all the tables used and their corresponding reads for a query. Both the `duration` and `CPU` values may fluctuate significantly when reexecuting the same query with no change in table schema (including indexes) or data because the essential services and background applications running on the SQL Server machine can affect the processing time of the query under observation. But, don't forget that logical reads are not always the most accurate measure. Duration and CPU are absolutely useful and an important part of any query tuning.

During optimization steps, you need a nonfluctuating cost figure as a reference. The reads (or logical reads) don't vary between multiple executions of a query with a fixed table schema and data. For example, if you execute the previous `SELECT` statement ten times, you will probably get ten different figures for `duration` and `CPU`, but `Reads` will remain the same each time. Therefore, during optimization, you can refer to the number of reads for an individual table to ensure that you really have reduced the data access cost of the table. Just never assume that is your only measure or even the primary one. It's just a constant measure and therefore useful.

Even though the number of logical reads can also be obtained from Extended Events, you get another benefit when using `STATISTICS IO`. The number of logical reads for a query shown by Profiler or the Server Trace option increases as you use different `SET` statements (mentioned previously) along with the query. But the number of logical reads shown by `STATISTICS IO` doesn't include the additional pages that are accessed because `SET` statements are used with a query. Thus, `STATISTICS IO` provides a consistent figure for the number of logical reads.

Actual Execution Plans

As mentioned earlier in the chapter, actual execution plans now capture and display some query performance metrics within the execution plan itself along with the traditional metrics. If we open the `SELECT` operator for the previous query and plan, the one without the `WHERE` clause, Figure 7-26 shows both the `QueryTimeStats` and `WaitStats` values.

www.EBooksWorld.ir

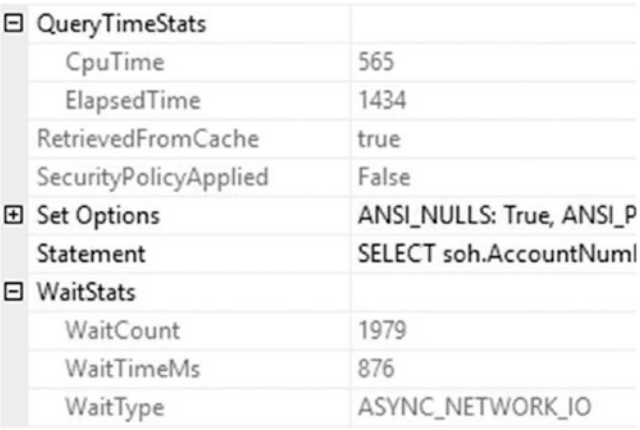


Figure 7-26. *QueryTimeStats and WaitStats within an actual execution plan*

You can now see the `CpuTime` and `ElapsedTime` for the query directly within the execution plan, as long as you're capturing an actual execution plan. These values are measured in milliseconds. You can also see the top wait or waits for a query. In the Figure 7-26 example it's `ASYNC_NETWORK_IO`, probably explained by the fact that we're returning 121,000 rows across the network. The wait statistics show up only if they are longer than 1ms. This does lead to the waits shown within an execution being not as accurate as the other mechanisms for capturing waits. However, this is a handy tool to help evaluate the behavior of the query within the execution plan.

This gives you yet another quick and easy way to see query performance. If you look at one of the other operators, you can also see the I/O for that operator, measured in pages.

Summary

In this chapter, you saw that you can use Extended Events to identify the queries causing a high amount of stress on the system resources in a SQL workload. Collecting the session data can, and should be, automated using system stored procedures. For immediate access to statistics about running queries, use the DMV `sys.dm_exec_query_stats`. You can further analyze these queries with Management Studio to find the costly steps in the processing strategy of the query. For better performance, it is important to consider both the index and join mechanisms used in an execution plan while analyzing a query. The number of data retrievals (or reads) for the individual tables provided by

www.EBooksWorld.ir

`SET STATISTICS IO` helps concentrate on the data access mechanism of the tables with the most reads. You also should focus on the CPU cost and overall time of the most costly queries.

Once you identify a costly query and finish the initial analysis, the next step should be to optimize the query for performance. Because indexing is one of the most commonly used performance-tuning techniques, in the next chapter. I will discuss in depth the various indexing mechanisms available in SQL Server.

www.EBooksWorld.ir

CHAPTER 8

Index Architecture and Behavior

The right index on the right column, or columns, is the basis on which query tuning begins. A missing index or an index placed on the wrong column, or columns, can be the basis for all performance problems starting with basic data access, continuing through joins, and ending in filtering clauses. For these reasons, it is extremely important for everyone—not just a DBA—to understand the different indexing techniques that can be used to optimize the database design.

In this chapter, I cover the following topics:

- What an index is
- The benefits and overhead of an index
- General recommendations for index design
- Clustered and nonclustered index behavior and comparisons
- Recommendations for clustered and nonclustered indexes

What Is an Index?

One of the best ways to reduce disk I/O is to use an index. An index allows SQL Server to find data in a table without scanning the entire table. An index in a database is analogous to an index in a book. Say, for example, that you wanted to look up the phrase *table scan* in this book. In the paper version, without the index at the back of the book, you would have to peruse the entire book to find the text you needed. With the index, you know exactly where the information you want is stored.

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_8

www.EBooksWorld.ir

While tuning a database for performance, you create indexes on the different columns used in a query to help SQL Server find data quickly. For example, the following query against the `Production.Product` table results in the data shown in Figure 8-1 (the first 10 of 500+ rows):

```
SELECT TOP 10
    p.ProductID,
    p.[Name],
    p.StandardCost,
    p.[Weight],
    ROW_NUMBER() OVER (ORDER BY p.Name DESC) AS RowNumber
FROM Production.Product p
ORDER BY p.Name DESC;
```

	ProductID	Name	StandardCost	Weight	RowNumber
1	852	Women's Tights, S	30.9334	NULL	1
2	853	Women's Tights, M	30.9334	NULL	2
3	854	Women's Tights, L	30.9334	NULL	3
4	867	Women's Mountain Shorts, S	26.1763	NULL	4
5	868	Women's Mountain Shorts, M	26.1763	NULL	5
6	869	Women's Mountain Shorts, L	26.1763	NULL	6
7	870	Water Bottle - 30 oz.	1.8663	NULL	7
8	842	Touring-Panniers, Large	51.5625	NULL	8
9	965	Touring-3000 Yellow, 62	461.4448	30.00	9
10	964	Touring-3000 Yellow, 58	461.4448	29.79	10

Figure 8-1. *Sample Production.Product table*

The preceding query scanned the entire table since there was no `WHERE` clause. If you need to add a filter through the `WHERE` clause to retrieve all the products where `StandardCost` is greater than 150, without an index the table will still have to be scanned, checking the value of `StandardCost` at each row to determine which rows contain a value greater than 150. An index on the `StandardCost` column could speed up this process by providing a mechanism that allows a structured search against the data rather than a row-by-row check. You can take two different, and fundamental, approaches for creating this index.

www.EBooksWorld.ir

- *Like a dictionary*: A dictionary is a distinct listing of words in alphabetical order. An index can be stored in a similar fashion. The data is ordered, although it will still have duplicates. The first ten rows, ordered by `StandardCost DESC` instead of by `Name`, would look like the data shown in Figure 8-2. Notice the `RowNumber` column shows the original placement of the row when ordering by `Name`.

	ProductID	Name	StandardCost	Weight	RowNumber
1	749	Road-150 Red, 62	2171.2942	15.00	125
2	753	Road-150 Red, 56	2171.2942	14.68	126
3	752	Road-150 Red, 52	2171.2942	14.42	127
4	751	Road-150 Red, 48	2171.2942	14.13	128
5	750	Road-150 Red, 44	2171.2942	13.77	129
6	774	Mountain-100 Silver, 48	1912.1544	21.42	170
7	773	Mountain-100 Silver, 44	1912.1544	21.13	171
8	772	Mountain-100 Silver, 42	1912.1544	20.77	172
9	771	Mountain-100 Silver, 38	1912.1544	20.35	173
10	778	Mountain-100 Black, 48	1898.0944	21.42	174

Figure 8-2. *Product table sorted on StandardCost*

So, now if you wanted to find all the data in the rows where `StandardCost` is greater than 150, the index would allow you to find them immediately by moving down to the first value greater than 150. An index that applies order to the data stored based on the index key order is known as a *clustered index*. Because of how SQL Server stores data, this is one of the most important indexes in your database design. I explain this in detail later in the chapter.

- *Like a book's index architecture*: An ordered list can be created without altering the layout of the table, similar to the way the index of a book is created. Just like the keyword index of a book lists the keywords in a separate section with a page number to refer to the main content of the book, the list of `StandardCost` values is created as a separate structure and refers to the corresponding row in the `Product` table through a pointer. For the example, I'll use `RowNumber` as the pointer. Table 8-1 shows the structure of the manufacturer index.

www.EBooksWorld.ir

Table 8-1. *Structure of the Manufacturer Index*

StandardCost	RowNumber
2171.2942	125
2171.2942	126
2171.2942	127
2171.2942	128
2171.2942	129
1912.1544	170

SQL Server can scan the manufacturer index to find rows where `StandardCost` is greater than 150. Since the `StandardCost` values are arranged in a sorted order, SQL Server can stop scanning as soon as it encounters the row with a value of 150 or less. This type of index is called a *nonclustered index*, and I explain it in detail later in the chapter.

In either case, SQL Server will be able to find all the products where `StandardCost` is greater than 150 more quickly than without an index under most circumstances.

You can create indexes on either a single column (as described previously) or a combination of columns in a table. SQL Server also automatically creates indexes for certain types of constraints (for example, `PRIMARY KEY` and `UNIQUE` constraints).

The Benefit of Indexes

SQL Server has to be able to find data, even when no index is present on a table. When no clustered index is present to establish a storage order for the data, the storage engine will simply read through the entire table to find what it needs. A table without a clustered index is called a *heap table.* A heap is just an unordered stack of data with a row identifier as a pointer to the storage location. This data is not ordered or searchable except by walking through the data, row by row, in a process called a *scan.* When a clustered index is placed on a table, the key values of the index establish an order for the data. Further, with a clustered index, the data is stored with the index so that the data itself is now ordered. When a clustered index is present, the pointer on the nonclustered index consists of the values that define the clustered index key. This is a big part of what makes clustered indexes so important.

www.EBooksWorld.ir

Data within SQL Server is stored on a page, which is 8KB in size. A page is the minimum amount of information that moves off the disk and into memory, so how much you can store on a page becomes important. Since a page has a limited amount of space, it can store a larger number of rows if the rows contain a fewer number of columns or the columns are of smaller size. The nonclustered index usually doesn't (and shouldn't) contain all the columns of the table; it usually contains only a limited number of the columns. Therefore, a page will be able to store more rows of a nonclustered index than rows of the table itself, which contains all the columns. Consequently, SQL Server will be able to read more values for a column from a page representing a nonclustered index on the column than from a page representing the table that contains the column.

Another benefit of a nonclustered index is that because it is in a separate structure from the data table, it can be put in a different filegroup, with a different I/O path, as explained in Chapter 3. This means SQL Server can access the index and table concurrently, making searches even faster.

Indexes store their information in a balanced tree, referred to as a *B-tree*, structure, so the number of reads required to find a particular row is minimized. The following example shows the benefit of a B-tree structure.

Consider a single-column table with 27 rows in a random order and only 3 rows per leaf page. Suppose the layout of the rows in the pages is as shown in Figure 8-3.

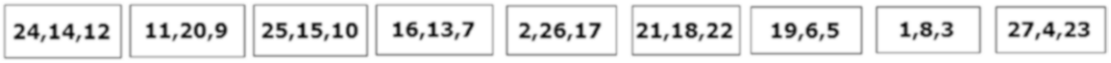


Figure 8-3. *Initial layout of 27 rows*

To search the row (or rows) for the column value of 5, SQL Server has to scan all the rows and the pages since even the last row in the last page may have the value 5. Because the number of reads depends on the number of pages accessed, nine read operations (retrieving pages from the disk and transferring them to memory) have to be performed without an index on the column. This content can be ordered by creating an index on the column, with the resultant layout of the rows and pages shown in Figure 8-4.

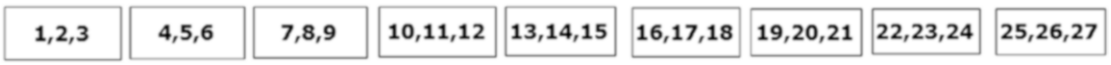


Figure 8-4. *Ordered layout of 27 rows*

www.EBooksWorld.ir

Indexing the column arranges the content in a sorted fashion. This allows SQL Server to determine the possible value for a row position in the column with respect to the value of another row position in the column. For example, in Figure 8-4, when SQL Server finds the first row with the column value 6, it can be sure that there are no more rows with the column value 5. Thus, only two read operations are required to fetch the rows with the value 5 when the content is indexed. However, what happens if you want to search for the column value 25? This will require nine read operations! This problem is solved by implementing indexes using the B-tree structure (as shown in Figure 8-5).

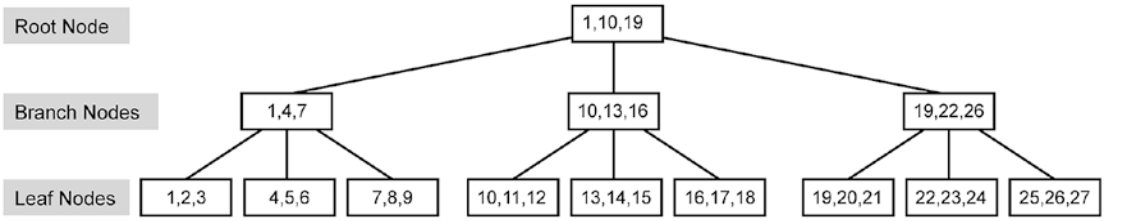


Figure 8-5. *B-tree layout of 27 rows*

A B-tree consists of a starting node (or page) called a *root node* with *branch nodes* (or pages) growing out of it (or linked to it). All keys are stored in the leaves. Contained in each interior node (above the leaf nodes) are pointers to its branch nodes and values representing the smallest value found in the branch node. Keys are kept in sorted order within each node. B-trees use a balanced tree structure for efficient record retrieval—a B-tree is balanced when the leaf nodes are all at the same level from the root node. For example, creating an index on the preceding content will generate the balanced B-tree structure shown in Figure 8-5. At the bottom level, all the leaf nodes are connected to each other through a doubly linked list, meaning each page points to the page that follows it, and the page that follows it points back to the preceding page. This prevents having to go back up the chain when pages are traversed beyond the definitions of the intermediate pages.

The B-tree algorithm minimizes the number of pages to be accessed to locate a desired key, thereby speeding up the data access process. For example, in Figure 8-5, the search for the key value 5 starts at the top root node. Since the key value is between 1 and 10, the search process follows the left branch to the next node. As the key value 5 falls between the values 4 and 7, the search process follows the middle branch to the next node with the starting key value of 4. The search process retrieves the key value 5 from this leaf page. If the key value 5 doesn't exist in this page, the search process will stop since it's the leaf page. Similarly, the key value 25 can also be searched using the same number of reads.

www.EBooksWorld.ir

Index Overhead

The performance benefit of indexes does come at a cost. Tables with indexes require more storage and memory space for the index pages in addition to the data pages of the table. Data manipulation queries (`INSERT`, `UPDATE`, and `DELETE` statements, or the CUD part of Create, Read, Update, Delete [CRUD]) can take longer, and more processing time is required to maintain the indexes of constantly changing tables. This is because, unlike a `SELECT` statement, data manipulation queries modify the data content of a table. If an `INSERT` statement adds a row to the table, then it also has to add a row in the index structure. If the index is a clustered index, the overhead is greater still because the row has to be added to the data pages themselves in the right order, which may require other data rows to be repositioned below the entry position of the new row. The `UPDATE` and `DELETE` data manipulation queries change the index pages in a similar manner.

When designing indexes, you'll be operating from two different points of view: the existing system, already in production, where you need to measure the overall impact of an index, and the tactical approach where all you worry about is the immediate benefits of an index, usually when initially designing a system. When you have to deal with the existing system, you should ensure that the performance benefits of an index outweigh the extra cost in processing resources. You can do this by using Extended Events (explained in Chapter 3) to do an overall workload optimization (explained in Chapter 27). When you're focused exclusively on the immediate benefits of an index, SQL Server supplies a series of dynamic management views that provide detailed information about the performance of indexes, `sys.dm_db_index_operational_stats` or `sys.dm_db_index_usage_stats`. The view `sys.dm_db_index_operational_stats` shows the low-level activity, such as locks and I/O, on an index that is in use. The view `sys.dm_db_index_usage_stats` returns statistical counts of the various index operations that have occurred to an index over time. Both of these will be used more extensively in Chapter 21 when I discuss blocking.

Note In some parts of the book, I use the `STATISTICS IO` and `STATISTICS TIME` measurements against the queries that I'm running. You can add `SET` commands to the code, or you can change the connection settings for the query window. I suggest just changing the connection settings. However, there should also be a warning here. Using `STATISTICS IO` and `STATISTICS TIME` together can sometimes cause problems. The time it takes to retrieve the I/O information

www.EBooksWorld.ir

is added to the `STATISTICS TIME` information, thus skewing the results. If you don't need table-level I/O, it's better to capture the execution metrics using Extended Events. You also can get `CpuTime` and `ElapsedTime` from actual execution plans if you're capturing one for a query.

To understand the overhead cost of an index on data manipulation queries, consider the following example. First, create a test table with 10,000 rows.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1
(
    C1 INT,
    C2 INT,
    C3 VARCHAR(50)
);

WITH Nums
AS (SELECT TOP (10000)
        ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns ac1
        CROSS JOIN master.sys.all_columns ac2
   )
INSERT INTO dbo.Test1
(
    C1,
    C2,
    C3
)
SELECT n,
       n,
       'C3'
FROM Nums;
```


www.EBooksWorld.ir

Run an UPDATE statement, like so:

```
UPDATE dbo.Test1
SET C1 = 1,
    C2 = 1
WHERE C2 = 1;
```

Then the number of logical reads reported by SET STATISTICS IO is as follows:

```
Table 'Test1'. Scan count 1, logical reads 29
```

Add an index on column c1, like so:

```
CREATE CLUSTERED INDEX iTest
ON dbo.Test1(C1);
```

Then the resultant number of logical reads for the same UPDATE statement increases from 29 to 38 but also has added a worktable with an additional 5 reads, for a total of 43.

```
Table 'Test1'. Scan count 1, logical reads 38
Table 'Worktable'. Scan count 1, logical reads 5
```

The number of reads goes up because it was necessary to rearrange the data in order to store it in the correct order within the clustered index, increasing the number of reads beyond what was necessary for a heap table to just add the data to the end of the existing storage.

Note A *worktable* is a temporary table used internally by SQL Server to process the intermediate results of a query. Worktables are created in the tempdb database and are dropped automatically after query execution.

Even though it is true that the amount of overhead required to maintain indexes increases for data manipulation queries, be aware that SQL Server must first find a row before it can update or delete it; therefore, indexes can be helpful for UPDATE and DELETE statements with necessary WHERE clauses. The increased efficiency in using the index to locate a row usually offsets the extra overhead needed to update the indexes, unless the table has a lot of indexes or lots of updates. Further, the vast majority of systems are read heavy, meaning they have a lot more data being retrieved than is being inserted or modified.

www.EBooksWorld.ir

To understand how an index can benefit even data modification queries, let's build on the example. Create another index on table `Test1`. This time, create the index on column `C2` referred to in the `WHERE` clause of the `UPDATE` statement.

```
CREATE NONCLUSTERED INDEX iTest2
ON dbo.Test1(C2);
```

After adding this new index, run the `UPDATE` command again.

```
UPDATE  dbo.Test1
SET     C1 = 1,
        C2 = 1
WHERE   C2 = 1;
```

The total number of logical reads for this `UPDATE` statement decreases from 43 to 20 (= 15 + 5).

```
Table 'Test1'. Scan count 1, logical reads 15
Table 'Worktable'. Scan count 1, logical reads 5
```

The examples in this section have demonstrated that although having an index adds some overhead cost to action queries, the overall result can be a decrease in cost because of the beneficial effect of indexes on searching, even during updates.

Index Design Recommendations

The main recommendations for index design are as follows:

- Examine the `WHERE` clause and `JOIN` criteria columns.
- Use narrow indexes.
- Examine column uniqueness and selectivity.
- Examine the column data type.
- Consider column order.
- Consider the type of index (clustered versus nonclustered).

Let's consider each of these recommendations in turn.

www.EBooksWorld.ir