THE FUNDAMENTAL GUIDE TO SQL QUERY OPTIMIZATION

Five performance tuning tips for SQL Server, with copy-and-paste queries and case studies

By Janis Griffin, Sr. Systems Consultant, Quest Software

Quest®

“Why is the system running so slowly today?”

How many times have you heard that complaint this week? This quarter? This year?

Almost all SQL Server database professionals hear it eventually. It’s a valid complaint, because a slow-running database bogs down productivity for internal and external users alike.

However, it doesn’t give you very much to go on, does it? Your users can’t tell you, “I think you need to deal with that wild card in the `SELECT` statement in line 443,” or “The Order Status query is performing too many unnecessary logical reads and hogging CPU cycles.”

That’s up to you to figure out. And that’s where the SQL query tuning tips in this e-book come into play. Here you’ll find a reliable method for analyzing and addressing the performance bottlenecks in your SQL Server databases, along with case studies and diagnostic queries you can use right away. We’ve put together this resource to help you identify and fix performance issues more efficiently, so you spend less time hearing how the system is running slowly today.

“SQL query optimization is easy.” Said nobody. Ever.

HERE ARE A FEW REASONS WHY.

Where to optimize? — Locating the SQL statements that hamper your database performance can be like searching for a needle in a haystack of code. Besides, SQL Server could be processing dozens or hundreds of statements at any one time. So, where should you focus your effort? If you’ve ever tuned SQL to your satisfaction but no one else noticed an improvement, then you probably worked on the wrong statement or measured the wrong effect.

How to optimize? — SQL tuning requires experience in different areas. Do you know how to read and interpret execution plans? Do you know the best data access path off the disk? Which are the best Join methods to use? And, of course, do you know how to write good SQL? It often becomes necessary to rewrite an offending query.

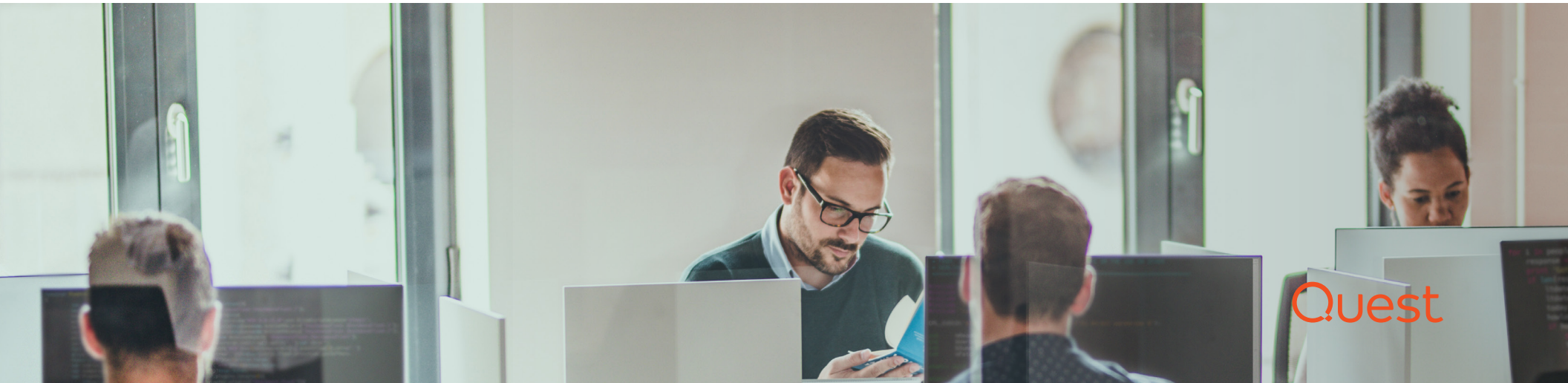
Why optimize? — It’s useful to know the business purpose of the SQL query. Suppose you get complaints at the end of every month about poor performance. You determine that the culprit is a set of reports for accounting that runs and prints for hours, then goes straight into a filing cabinet, unexamined by anybody. That’s not a SQL query optimization problem; it’s a business process problem.

Who should optimize? — The responsibility for optimizing may be murky. The DBAs say, “The developers wrote the code, so they should tune it.” The developers counter, “The DBAs see the code in production. They know the environment and how the servers are set up.” The organization has to wrap some process around that first.

What to optimize? — SQL query optimization is about software; it doesn’t make hardware problems go away. Before you start tuning anything, make sure your hardware resources (processor power, memory size, storage speed, network throughput) are suited to the database and the application running on it.[1]

Finally, like any diagnostic pursuit, SQL query optimization takes time, trial and error, as the following five tips illustrate.

[1] For a holistic view of query performance, watch the webcast “Why Are My SQL Server Queries So Slow?” from Quest.



Tip 1: Monitor wait time

SQL Server incorporates wait types that allow you to monitor not only the total wait time but also each step of the query as it's processed through the database. Wait types offer invaluable clues about the amount of time taken and resources consumed by a query.

The first step is to run queries that capture and store wait time data so you can analyze it.

CAPTURE THE DATA IN WAIT TIME TABLES

SQL Server tracks data on wait time. Starting in SQL Server 2005, it holds the data in dynamic management views (DMVs).

Figure 1 shows the DMVs (table names underlined) and corresponding column names that contain the data most useful for tuning.

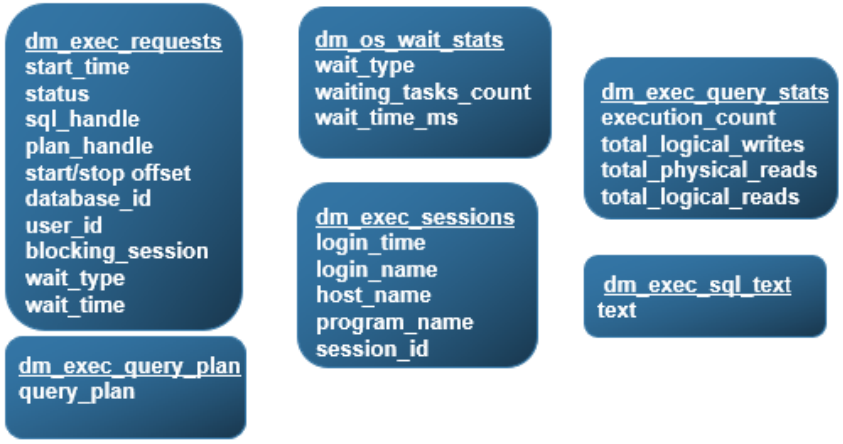


Figure 1: System DMVs and execution-related DMVs/functions

The problem is that DMVs are real-time, non-persistent views that reside only in memory. They offer only a snapshot, so they cannot tell you what happened, say, between 3:00 AM and 5:00 AM last Sunday morning. One option in SQL Server 2012 and later is to use extended events to gather wait types and query results, but even that doesn't make it easy to spot trends over time.

An effective and inexpensive way to capture exactly the data you want from the DMV is to poll it with a query at some interval, include the timestamp and save it to a table.

As you'll see below, from `dm_exec_requests` you can poll the `sql_handle` and `plan_handle` for the execution plan. If it's a stored procedure, you can get the snippet of code that the procedure is waiting on by reviewing the `statement_start_offset` and `statement_end_offset` columns. If a session is encountering a lock wait, `blocking_session_id` will return its name.

You can JOIN `dm_exec_requests` to `dm_exec_sessions` to list information like `login_name`, `host_name` and `program_name` about the session that's running the SQL. Then, JOIN that into `dm_exec_query_stats` for `execution_count`, `total_logical_reads` and `total_logical_writes`. Finally, two table-valued functions, `dm_exec_sql_text` and `dm_exec_query_plan`, hold the text of the SQL statement and the cached query plan, respectively.

VIEWING WAIT TYPES — SESSION LEVEL

Start your optimization at the session level. The query in Figure 2 polls **dm_exec_requests** (green type) to generate a view of each running session and the resource that the SQL statement is waiting on.

```
SELECT r.session_id, r.wait_time, r.status,
r.wait_type, r.blocking_session_id, s.text,
r.statement_start_offset, r.statement_end_
offset, p.query_plan

FROM sys.dm_exec_requests r

OUTER APPLY sys.dm_exec_sql_text (r.sql_handle) s

OUTER APPLY sys.dm_exec_text_query_plan
(r.plan_handle, r.statement_start_offset,
r.statement_end_offset) p

WHERE r.status <> 'background' AND r.status <>
'sleeping' AND r.session_id <> @@SPID
```

Figure 2: Query session-level wait types

The query JOINs it to **dm_exec_sql_text** with the **sql_handle**, then to **dm_exec_text_query_plan** (blue text), using the **plan_handle**. If the session is waiting on a lock wait type, it will also capture the **blocking_session_id**.

The **r.status** column in the **WHERE** statement is useful in reducing extraneous noise by excluding background processes and any sleeping or idle processes. Also, the **WHERE** statement excludes the session belonging to the administrator running this query.

Figure 3 shows the first six columns of output of this query against an instance of the AdventureWorks sample database in SQL Server.

	session_id	wait_time	status	wait_type	bloc...	text
1	52	0	running	NULL	0	SELECT r.session_id, r.wait_time, r.status, r.wait_ty...
2	55	0	runnable	NULL	0	NULL
3	56	0	suspended	ASYNC_NETWORK_IO	0	SELECT* FROM [AdventureWorks2014].[Produ...
4	57	0	runnable	NULL	0	SELECT* FROM [AdventureWorks2014].[Produ...
5	60	0	runnable	NULL	0	SELECT pth.*, ptha.* FROM Production.Transact...

Figure 3: Result of session-level query

A status of running or runnable shows that the session is on CPU or in the CPU queue. The suspended status, in row 3 for example, indicates that the session is waiting on a wait type, most likely to feed data to the client.

That query shows you what is running right now. So, to examine trends over time, you add a timestamp to the query, run it at a suitable interval and load the results into a table. You can then quickly discover which queries are spending the most time in the database and begin to see how to tune them.

VIEWING WAIT TYPES — INSTANCE LEVEL

But suppose you're not familiar with an instance, yet you need to get control of many instances in short order. In that case, you'll want to see which resources the instance is using and where the bottlenecks lie.

To look at wait types at the instance level, you can view `dm_os_wait_stats`. Note, however, that it shows data accumulated since the most recent startup or when the cache was last cleared. You can see where the bottlenecks lie, but not at any single point in time.

So, if you have a performance issue right now and you have no other diagnostic tools, you can clear the view by running:

```
DBCC SQLPERF (N'sys.dm_os_wait_stats', CLEAR);
```

Next, run the query from the article "SQL Server Wait Statistics" for the most recent statistics, shown in Figure 4.

WaitType	Wait_S	Resource_S	Signal_S	WaitCount	Percentage	AvgWait_S	AvgRes_S	AvgSig_S
LCK_M_S	10954.90	10954.89	0.02	15	80.40	730.3269	730.3259	0.0010
ASYNC_NETWORK_IO	891.43	819.74	71.69	4507	6.54	0.1978	0.1819	0.0159
SOS_SCHEDULER_YIELD	666.84	0.85	665.99	945421	4.89	0.0007	0.0000	0.0007
THREADPOOL	577.06	577.04	0.01	295536	4.24	0.0020	0.0020	0.0000

Figure 4: Wait statistics – Example 1

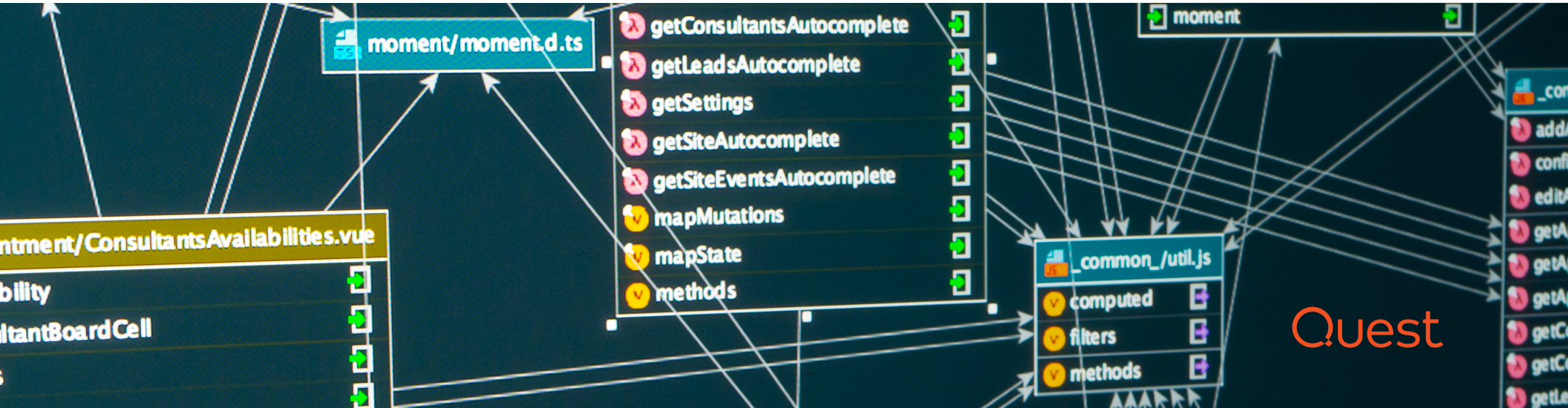
The query excludes idle wait types, which rarely hamper performance, and its output shows you, at the instance level, where you're waiting the most. For example, the instance queried in Figure 4 is spending over 80 percent of its processing time on a lock wait.

Figure 5 shows an example of another instance and what it's waiting on the most.

	WaitType	Wait_S	Resource_S	Signal_S	WaitCount	Percentage	AvgWait_S	AvgRes_S	AvgSig_S
1	SOS_SCHEDULER_YIELD	11432.86	5.00	11427.87	584690	52.65	0.0196	0.0000	0.0195
2	ASYNC_NETWORK_IO	4085.79	3373.00	712.78	10911	18.81	0.3745	0.3091	0.0653
3	THREADPOOL	2648.13	2648.10	0.03	571491	12.19	0.0046	0.0046	0.0000
4	RESOURCE_SEMAPHORE	564.01	555.15	8.86	25	2.60	22.5606	22.2060	0.3546
5	PREEMPTIVE_XE_CALLBACKEXECUTE	365.95	365.95	0.00	519097625	1.69	0.0000	0.0000	0.0000
6	PAGEIOLATCH_SH	344.94	337.99	6.95	30529	1.59	0.0113	0.0111	0.0002
7	PREEMPTIVE_OS_QUERYREGISTRY	339.71	339.71	0.00	25609	1.56	0.0133	0.0133	0.0000
8	PREEMPTIVE_OS_AUTHENTICATIONOPS	337.35	337.35	0.00	44621	1.55	0.0076	0.0076	0.0000

Figure 5: Wait statistics – Example 2

The instance spent more than half its time on `SOS_SCHEDULER_YIELD`, meaning it gave up processes because it was yielding to the CPU scheduler.



COLLECTING WAIT TIME DATA AT INTERVALS

Now that you've seen those views, you can run the following base queries to collect wait time data. The process is to first create a table in which to store the data, then automate polling to insert data into the table at some interval. The final step outputs the table of wait time data.

Figure 6 contains code to create the Wait Time Analysis table by `SELECT`ing into `rta_data`. (The green text shows the delta from the query in Figure 2.)

```
SELECT r.session_id, r.wait_time, r.status,
r.wait_type, r.blocking_session_id, s.text,
r.statement_start_offset, r.statement_end_offset,
p.query_plan, CURRENT_TIMESTAMP time_polled

INTO rta_data

FROM sys.dm_exec_requests r

OUTER APPLY sys.dm_exec_sql_text (r.sql_handle) s

OUTER APPLY sys.dm_exec_text_query_plan
(r.plan_handle, r.statement_start_offset,
r.statement_end_offset) p

WHERE r.status <> 'background' AND r.status <>
'sleeping' AND r.session_id <> @@SPID
```

Figure 6: Query to create wait-time table

The query in Figure 7 automates the `INSERT`, polling `dm_exec_requests` every second; that interval is granular enough to be useful. Since this is an in-memory query, its execution has negligible impact on the overall performance of the database.

```
INSERT INTO rta_data

SELECT r.session_id, r.wait_time, r.status, r.wait_
type, r.blocking_session_id,

              s.text, r.statement_start_offset,
r.statement_end_offset, p.query_plan,

            CURRENT_TIMESTAMP time_polled

FROM sys.dm_exec_requests r

OUTER APPLY sys.dm_exec_sql_text (r.sql_handle) s

OUTER APPLY sys.dm_exec_text_query_plan

 (r.plan_handle, r.statement_start_offset,
r.statement_end_offset) p

WHERE r.status <> 'background' AND r.status <>
'sleeping' AND r.session_id <> @@SPID
```

Figure 7: Query to automate INSERT

Finally, Figure 8 shows a query with a common table expression (CTE) to define a temporary named result set that contains the wait type and count.

```
WITH rta (sql_text, wait_type, time_in_second)

AS

(SELECT text as sql_text, wait_type, COUNT(*) as
time_in_second

  FROM rta_data rta

  GROUP BY text,wait_type) ,tot (text, tot_time)

AS

(SELECT text, COUNT(*) as tot_time

   FROM rta_data GROUP BY text)

SELECT sql_text, wait_type, time_in_
second, tot_time

FROM rta

JOIN tot ON sql_text = text

ORDER BY tot_time, time_in_second, wait_type,
sql_text
```

Figure 8: Query for CTE

The results from the query above are shown in Figure 9. The SQL statement that spent the most time in the database spent 2 seconds on network wait, 68 seconds on I/O completion and 203 seconds on CPU.

sql_text	wait_type	time_in_second	tot_time
SELECT ProductID, LineTotal FROM Sales.SalesOrd...	NULL	1	1
SELECT ProductID, OrderQty, UnitPrice, LineTotal FR...	NULL	1	1
/* If you want to make sure there are at least one thous...	NULL	1	1
DECLARE @CurrentEmployee hierarchyid SELECT @...	NULL	1	1
SELECT ProductID, LineTotal FROM Sales.SalesOrder...	NULL	1	1
/* http://msdn.microsoft.com/en-us/library/ms187731....	NULL	2	2
/* http://msdn.microsoft.com/en-us/library/ms187731....	NULL	1	4
/* This is the query that calculates the revenue for ea...	NULL	1	4
/* http://msdn.microsoft.com/en-us/library/ms187731....	CXPACKET	3	4
/* This is the query that calculates the revenue for ea...	CXPACKET	3	4
declare @prod int declare prod_cursor CURSOR fOR ...	NULL	4	4
(@minAmt money , @maxAmt money) SELECT p.Name ...	ASYNC_NETWORK_IO	2	273
(@minAmt money , @maxAmt money) SELECT p.Name ...	IO_COMPLETION	68	273
(@minAmt money , @maxAmt money) SELECT p.Name ...	NULL	203	273

Figure 9: Wait time analysis

Again, when there is no wait type, the SQL statement is usually either on CPU or in the CPU queue awaiting execution.

WHY ANALYZE WAIT TIME?

Analyzing wait time has several benefits in SQL query optimization.

Before tuning any SQL, it's important to gather baseline metrics to see whether your changes improve performance. How long did a given operation take before the change? How much improvement does the user expect? How far, practically, can you tune? Baseline metrics allow you to set an acceptable goal and stop when you reach it.

It's also useful to know how to interpret the different wait types at work in your database, including locking/blocking (`LCK`), I/O problems (`PAGEIOLATCH`), latch contention (`LATCH`) and network slowdown (`NETWORK`). A query spending most of its time on `ASYNC_NETWORK_IO` does not necessarily mean a network problem; the system could simply be feeding too much data to the client. And where a single query generates multiple wait types, you'll find it best to tune for the wait type most often encountered and see how the others behave in response.

Finally, aside from the value of wait time analysis in SQL query optimization, if a production query slows down unexpectedly, you can analyze it to see what changed.

Quest

Tip 2: Review the execution plan

There are many ways to get the execution plan in SQL Server. SQL Server Management Studio (SSMS) allows for estimated, actual and real-time statistics.

Without executing the query, `SET SHOWPLAN_XML { ON | OFF }` tells the Query Optimizer in SQL Server to return an estimated plan of how the query will be executed.

To generate an actual plan after execution has ended, use `SET STATISTICS XML { ON | OFF }`. In addition to statistics on execution and resources used, the actual plan includes number of rows processed. It's best to include `SET STATISTICS IO ON` and `SET STATISTICS TIME ON` when testing your query.

Of course, an estimated execution plan from Query Optimizer is subject to inaccuracy because it's based on the estimated statistics without execution. But an actual execution plan can also mislead you if you run it in a dev/test environment where the configuration and data sets are different from those in production.

Other ways to get the execution plan include Profiler Tracing and Extended Events, options that tend to generate too much data that is not useful in SQL query optimization. Also, tracing requires that you trace all the time or know when the problem is going to occur.

To generate the plan that is currently active, query the table-valued function `dm_exec_query_plan`. Use the `plan_handle` and the start/stop offset columns from the `dm_exec_request` view for the query. Figure 10 shows an example based on an order query against the AdventureWorks database.

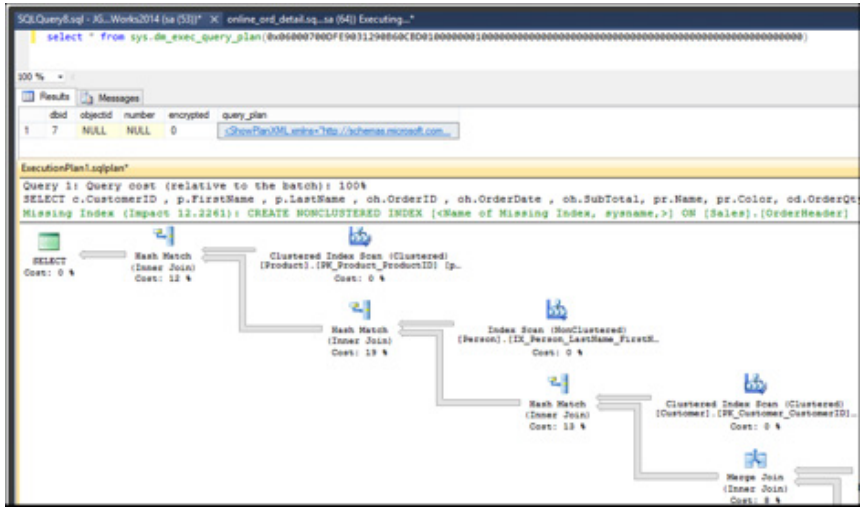


Figure 10: Sample execution plan (partial)

SSMS lists the SQL statement with a graphical execution plan to view. Read from right to left and bottom to top to see the sequence of steps.

WHAT TO LOOK FOR IN THE EXECUTION PLAN

Make sure the Join methods shown in the plan are appropriate.

Nested Loops Join — Compares each row from one table ("outer table") to each row in another table ("inner table") and returns rows that satisfy the Join predicate. The cost is proportional to the product of the number of rows in the two tables. This Join is well suited to smaller data sets.

Merge Join — Compares two sorted inputs, one row at a time. The cost is proportional to the sum of the total number of rows. This requires an equi-join condition and is efficient for larger data sets, especially in analytical queries.

Hash Match Join — Reads rows from one input, then hashes the rows, based on the equi-join condition, into an in-memory hash table. The Join does the same for the second input and then returns matching rows. It is most useful for very large data sets (especially data warehouses).

Ensure that expensive operations like full table scans and clustered index scans are justified; that is, do you need the query to read all that data? If an index is missing, the query may be reading too much data for the end result set and take up excessive CPU time that other processes could use.

Quest

Each step in the plan contains details, as shown in Figure 11. Examine the plan for exceptionally high numbers, such as Estimated I/O Cost, Estimated CPU Cost and Estimated Number of Rows.

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	22.0505
Estimated Operator Cost	23.4698 (33%)
Estimated Subtree Cost	23.4698
Estimated CPU Cost	1.41923
Estimated Number of Executions	1
Estimated Number of Rows	1135350
Estimated Row Size	32 B
Ordered	False
Node ID	10

Predicate

[AdventureWorks2014].[Sales].[OrderHeader].
[OnlineOrderFlag] as [oh].[OnlineOrderFlag]=(1)

Figure 11: Execution plan detail

Review the predicate information to understand how parameters are being interpreted. If the data type of a column is varchar and you're passing in an integer, that requires an implicit conversion, which burns CPU cycles unnecessarily.

Check the step in your query at which any filtering predicate is applied. It's preferable to filter in the early steps.

When you include `SET STATISTICS IO ON` in your query, SSMS displays messages with the number of logical reads, as depicted in Figure 12.

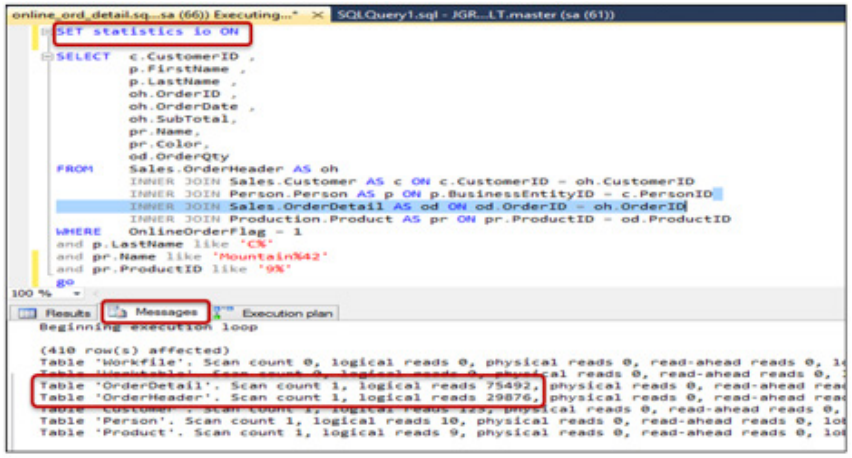


Figure 12: Logical I/O

Here, the busiest tables are `OrderDetail` with more than 75,000 logical reads and `OrderHeader` with almost 30,000 logical reads.

Logical I/O is a valuable baseline metric because it is a good target for optimization. Logical reads tend to be CPU-intensive. Every query performs at least one logical read, so if you can reduce that number, you can reclaim CPU cycles and keep from thrashing your memory — both good ways to tune.

IDENTIFYING CANDIDATES FOR SQL QUERY OPTIMIZATION

Figure 13 shows the execution plan for the same query against the AdventureWorks database.

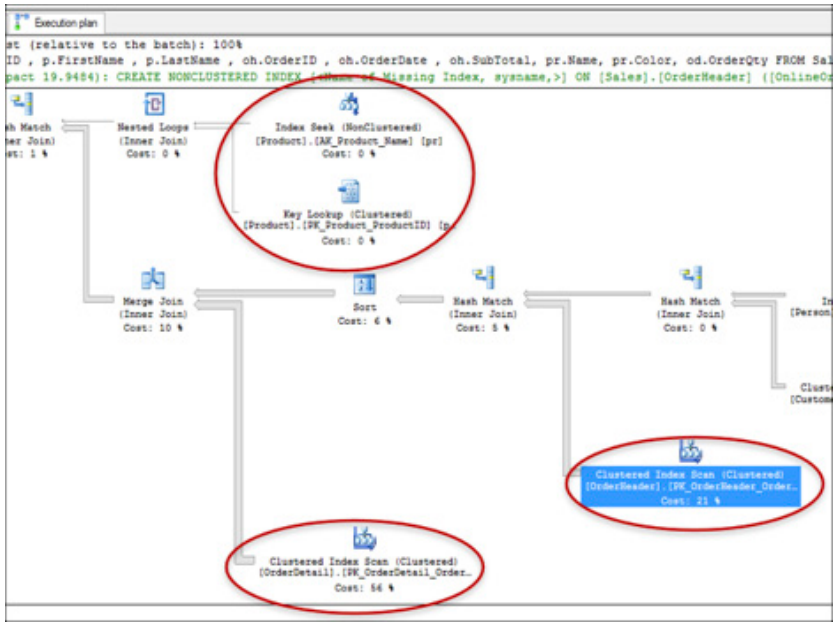


Figure 13: Execution plan (partial) – 3 areas to optimize

Between the steps in the plan, the width of the light-gray arrows indicates the amount of data that each step returns to the next step. The two most-expensive steps are the clustered index scans on `OrderHeader` and `OrderDetail`, in the lower half of the figure.

In the upper half of the figure is a key lookup. That indicates that the query is not using the primary key, but rather going to a non-clustered index, retrieving the data, then returning to the clustered table. In other words, it incurs another I/O to go back into the table itself.

Those three expensive steps are prominent candidates for optimization. Notice also that SQL Server has identified a missing index: `CREATE NONCLUSTERED INDEX` in green type. We'll review adding that index later, in case study 2.

Tip 3: Gather object information

The next task is to gather information about the objects associated with the expensive steps.

Review all information about the tables in the query, keeping in mind that they could be a view or a table-valued function. (You can determine that by hovering over the `FROM` clause in SSMS.) Know where the table resides physically. Examine the indexes, keys and constraints and how the tables are related.

INFORMATION ON TABLES, COLUMNS AND ROWS

Look at the size of the table and columns used, especially in the `WHERE` clause. To find the cardinality and distributions of a column, use SQL Server Integration Services (SSIS) to create a data profile task with all the tables in the query. As shown in Figure 14, you can then view that information using the `data_viewer` located on the Windows Start menu.

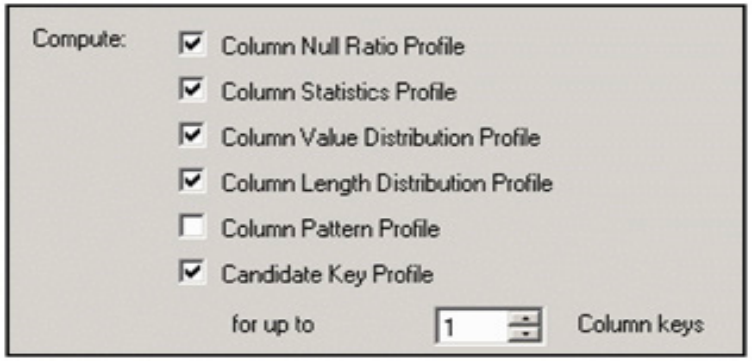
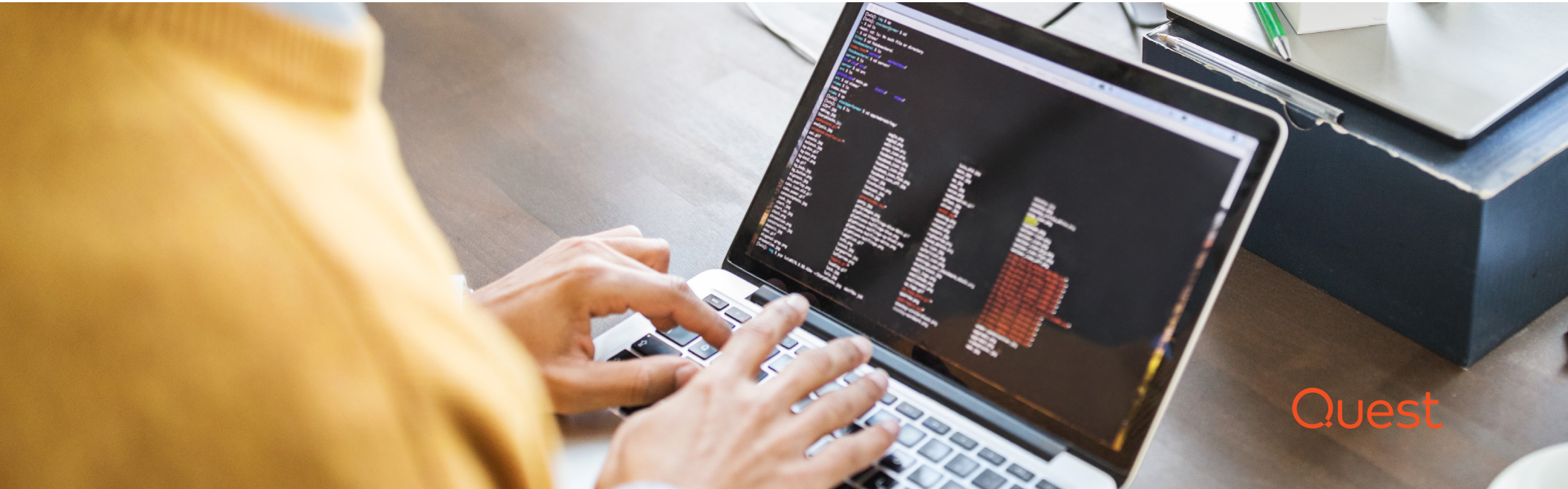


Figure 14: Data profile task in SSIS

Know whether additional statistics are being gathered and, if so, make sure they're up to date (for example, using `SET AUTO_UPDATE_STATISTICS ON;`). A query is only as good as its statistics.

Find out the row count for each table involved. An easy way to do that is to run a query from objects and partitions, as shown in Figure 15, to list the statistics that Query Optimizer knows.



```
SELECT so.name, sp.rows, so.type

FROM sys.objects so INNER JOIN sys.partitions sp

        ON so.object_id = sp.object_id

WHERE so.type IN ('TF','U','V')

AND sp.index_id = 1

ORDER BY so.name
```

Figure 15: Query for row count

The query lists the row counts for each table in a database. Notice that it filters on three types: table functions (`T`), user tables (`U`) and views (`V`). Figure 16 shows the output.

Results | Messages

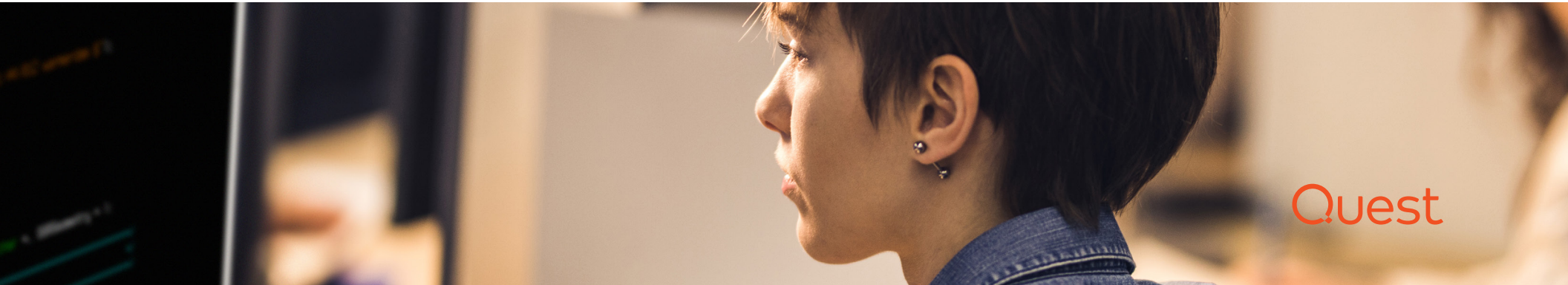
	name	rows	type
26	OrderDetail	4973997	U
27	OrderHeader	1290065	U
28	Password	19972	U
29	Person	19972	U
30	PersonCreditCard	19118	U
31	PersonPhone	19972	U
32	PhoneNumberType	3	U
33	Product	504	U

Figure 16: Output of query for row count

Row counts become important in finding the driving table (see below). The largest tables are `OrderDetail` with about 5 million records, and `OrderHeader` with 1.3 million. Coincidentally, they represent two of the three most expensive steps identified in the execution plan (see Figure 13).

Review the columns in the `SELECT` section of the query. If you find wild cards, make sure all those columns are necessary. It's generally a bad idea to `SELECT` * because it chews up memory and CPU cycles.

Other techniques that degrade performance include functions to convert mismatched data types (like integer to varchar) and non-searchable arguments in the `WHERE` clause. Without a valid `WHERE` clause, there's no filter, so no way to tune.



INFORMATION ON INDEXES

If it's a multi-column index, understand the order and selectivity of the columns.

If you have multiple tables in a query, it's useful to know the relationships among them. In SSMS, you can generate a diagram of the tables in the query, as depicted in Figure 17.

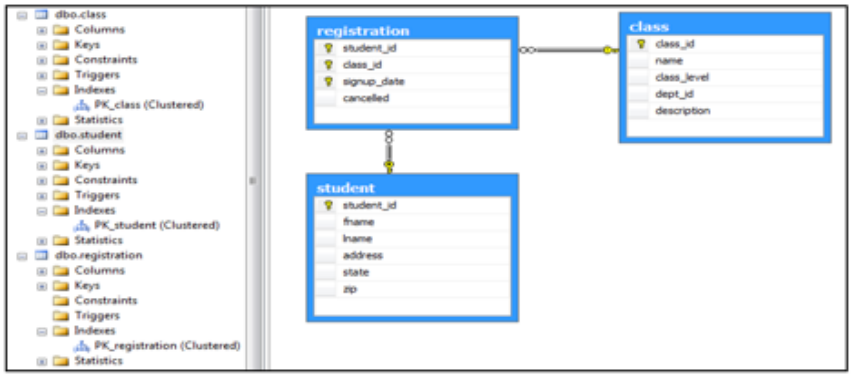


Figure 17: SSMS diagram of tables

The diagram shows how the tables are related. Here you can see all the columns for Join criteria and `WHERE` clauses. If you leave off a column that should have been used in the Join, Query Optimizer may choose a different or worse plan.

Similarly, keys and constraints can help Query Optimizer in choosing the execution plan. But many developers configure tables without any foreign or primary keys because they fear that keys slow processing down. When keys and constraints are used in good measure, they do not affect processing noticeably.

Watch for functions on indexed columns; they can cause Query Optimizer to turn off that index, in which case you may need to rewrite the query.

If the index is a composite or multi-column index, make sure you are referencing the left leading column in the index; otherwise, Query Optimizer won't use the index at all.

Finally, know whether and when you are rebuilding indexes. Rebuilding can have a sort of yo-yo effect on query performance: the index becomes fragmented, so queries run poorly, then the index gets rebuilt, so queries run well, then get fragmented again, and so on. The undesirable result is good performance on some days and bad performance on others.

Tip 4: Find the driving table

Your goal now is to drive the query with the table that returns the least data. That reduces the number of logical reads. In short, you study Joins and predicates, and filter earlier in the query rather than later.

For example, if you have two tables with 1 million rows each and `JOIN` them, you'll have a lot of data. But if you're interested in only two or three records, your query has generated a lot of needless logical reads. Filtering early whittles down the possible data sets and reduces that work, which is why it's useful to compare the size of the final result set with the sizes of the data sets being returned in each step of the execution plan.

As described above, the selectivity of the columns in your tables plays a role in your `WHERE` clauses, as does the actual size of data sets for each step in the execution plan. A close look at the Joins will give you an idea of how much data needs to be read.

Another useful technique for finding the driving table is SQL diagramming (see on page 18), a graphical method for mapping the amount of data in the tables and finding which filter will return the least amount.

The following two case studies illustrate the optimization tips given so far.

Case study 1: University billing system

The business issue behind this case study is that a university was trying to determine who registered for a course called “SQL TUNING” on a specific day.

They used the query shown in Figure 18.

```
SELECT s.fname, s.lname, r.signup_date

FROM   student s

    INNER JOIN registration r ON s.student_id
= r.student_id

    INNER JOIN class c ON r.class_id = c.class_id

WHERE  c.name  = 'SQL TUNING'

AND    r.signup_date BETWEEN :beg_date
AND :beg_date +1

AND    r.cancelled = ‘N‘
```

Figure 18: Case study 1 – original query

The query **SELECT**s student name and signup date from the student table, then **JOIN**s to **registration** on **student.id** and to class on **class.id**. It looks for current records with the name of ‘SQL Tuning’, a specific signup date and the cancelled flag of ‘N’ for non-cancelled. Wait time analysis shows that this query was taking the most time in the database.

MONITOR WAIT TIME

Wait time analysis yielded these statistics:

Execution stats – 3,009,351 logical reads

Average SQL response time – 5.2 seconds

Executions – 523

REVIEW THE EXECUTION PLAN AND LOGICAL I/O

Next, run the execution plan (Figure 19).

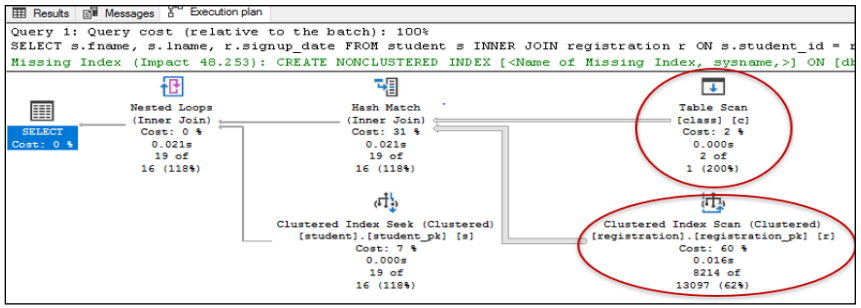


Figure 19: Original execution plan

In the upper right, the plan reveals a table scan of **class**, which represents a heap. With a heap, the table sits on disk, not organized in any way, probably generating a lot of logical I/O. Although the table itself is not very large, a table scan is inefficient.

The wide arrow leading from the registration table in the lower right reflects the clustered index scan and the high (60 percent) cost.

Also, in green type SSMS suggests adding an index on **registration** for the columns **cancelled** and **signup_date**. It is best to validate that suggestion before accepting it.

Quest

As in Figure 12, Figure 20 shows a query with `SET STATISTICS IO ON`.

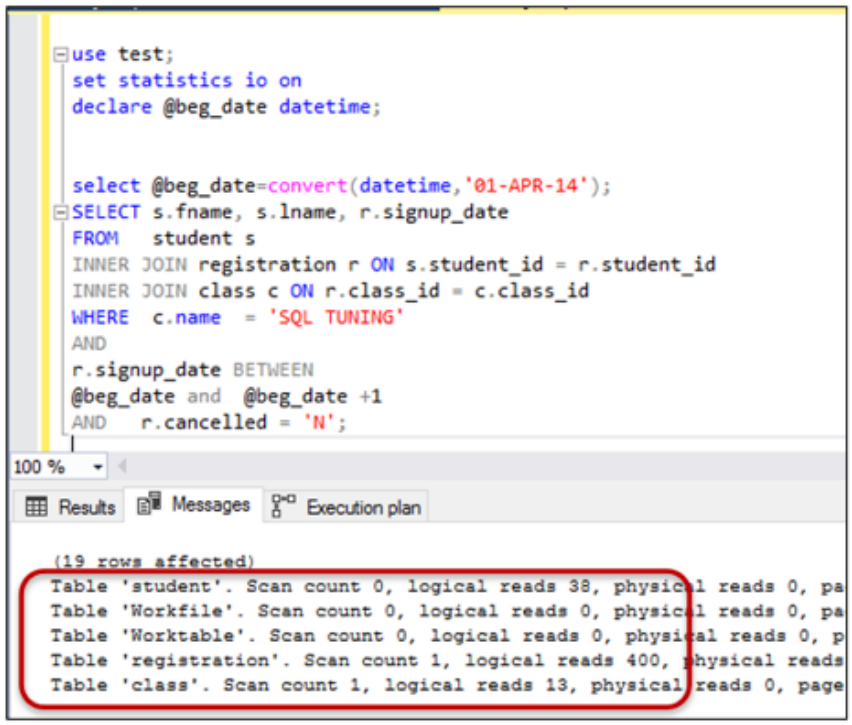


Figure 20: Query for logical I/O

Logical reads are not excessively high. However, the optimizer included the work tables `Workfile` and `Worktable`, which take up temporary space and add extra steps for the query to work through.

GATHER OBJECT INFORMATION

In gathering object information about the tables and indexes, the tables `student` (10,000 rows) and `registration` (80,000 rows) stand out as the largest (Figure 21).

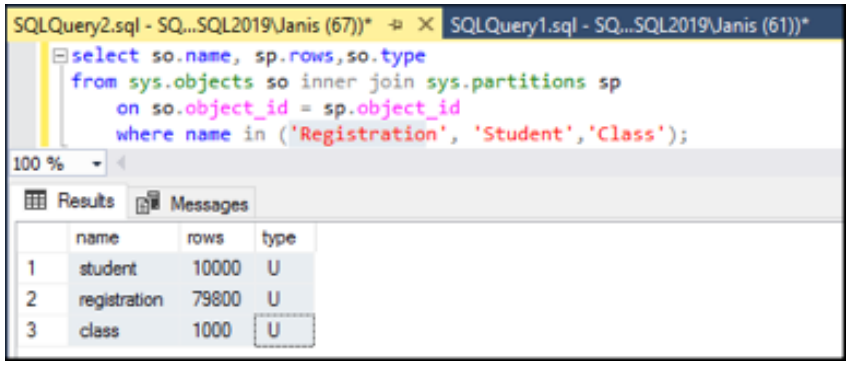


Figure 21: Examining database objects

A query on the indexes using `exec sp_helpindex` reveals that there are indexes for `registration` and `student`, but not for `class`, which uses the heap observed in the execution plan (Figure 22).

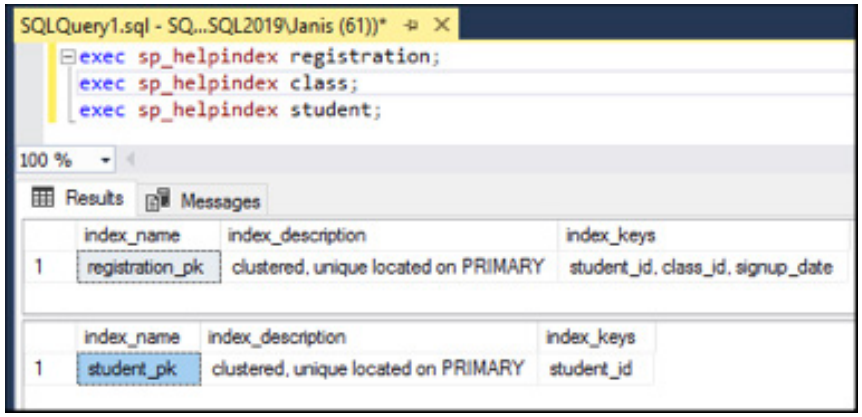


Figure 22: Exec query for indexes

As noted above, `class` is a small table; still, creating an index for it would be a low-impact, worthwhile fix to avoid having it use a heap.

Quest

FIND THE DRIVING TABLE

At this point, SQL diagramming is a good way to find the driving table.

First, determine which tables contain the detailed information and which tables are the master or lookup tables. In this simple case study, `registration` is the detail table. It has two lookup tables, `student` and `class`. To diagram these tables, draw an upside-down tree connecting the detail table (at the top) with arrows (or links) to the lookup tables, as depicted in Figure 23.

Figure 23: University billing system – SQL diagram

Now, calculate the relative number of records required for the Join criteria and put the numbers at each end of the arrow. For every 1 student there are about 5 records in the `registration` table, and for every 1 class there are about 30 records in `registration`. That means it should never be necessary to `JOIN` more than 150 (5 x 30) records to get a result for 1 student or 1 class.

In fact, if you just make sure that your Join columns are properly indexed, you can skip figuring out the math for it.

Next, look at the filtering predicates to find which table to drive the query with. This query had two filters: one on registration cancelled = '`N`' and the other on `signup_date` between two dates. To see how selective the filter is, run this query on registration:

```
select count(1) from registration where cancelled = 'N'

AND r.signup_date BETWEEN :beg_date AND :beg_date +1
```

It returns 4,344 records out of the 79,800 total records in registration. That is, roughly 5 percent of the records will be read with that filter.

The other filter is on class:

```
select count(1) from class where name = 'SQL TUNING'
```

It returns two records, or 0.2 percent, which represents a much more selective filter.

FIX 1 — CLASS TABLE

Thus, `class` is the driving table. As shown in Figure 19, `class` was using a table scan (heap) because it was missing an index. To fix `class`, add `CREATE UNIQUE CLUSTERED INDEX class_pk ON class(class_id);` for an index with a primary key.

Because the name column is in the `WHERE` clause, add a non-clustered index on name: `CREATE NONCLUSTERED INDEX class_nm_idx ON class(name);`

Figure 24 shows the execution plan, now that that index is being used.

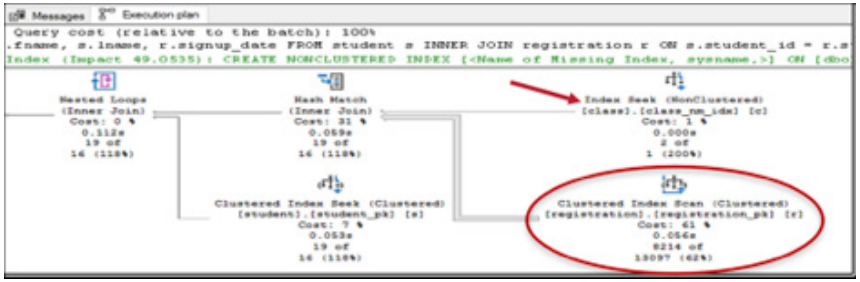


Figure 24: Execution plan (partial) after Fix 1

The logical reads for `registration` (400) and `student` (38) were unchanged (see Figure 20), and a clustered index scan is still being performed on `registration`. Why? Because, as described in "Information on indexes," the index for `registration` is leading by the `student_id` column instead of `class_id`. So, SQL Server can't use the index at all, and it has to use a clustered index scan.

FIX 2 — REGISTRATION TABLE

Add a non-clustered index:

```
CREATE NONCLUSTERED INDEX reg_alt ON
registration(class_id);
```

The new execution plan is shown in Figure 25.

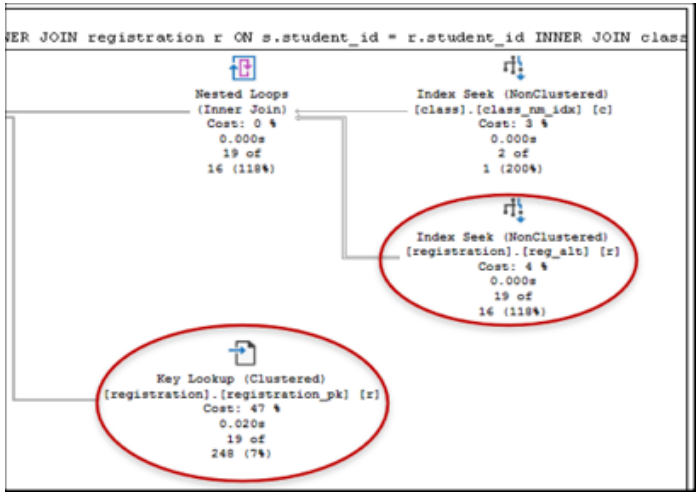


Figure 25: Execution plan (partial) after Fix 2

Now both `class` and `registration` are being accessed by an index seek and the costs of both steps have decreased significantly. However, notice the new Key Lookup step that the optimizer introduced. It added this step because it found the data in a non-clustered index but must go back to the table to get the rest of the data it needs for the query.

The Messages tab shows that the number of logical reads for `class` is still 2 and for `registration` it has fallen from 400 to 63, which demonstrates even more progress.

Quest

FIX 3 — COVERING INDEX

If you add a covering index, the optimizer can retrieve all the information it needs from the index without going back to the table, thus reducing I/O. (See "A note on indexes" below for more details on covering and filtering indexes.)

Modify the index by adding all the columns from `registration` that the optimizer will need to resolve the query. Run the following:

```
CREATE NONCLUSTERED INDEX reg_alt ON
registration(class_id, signup_date, cancelled);
```

The execution plan now looks like Figure 26.

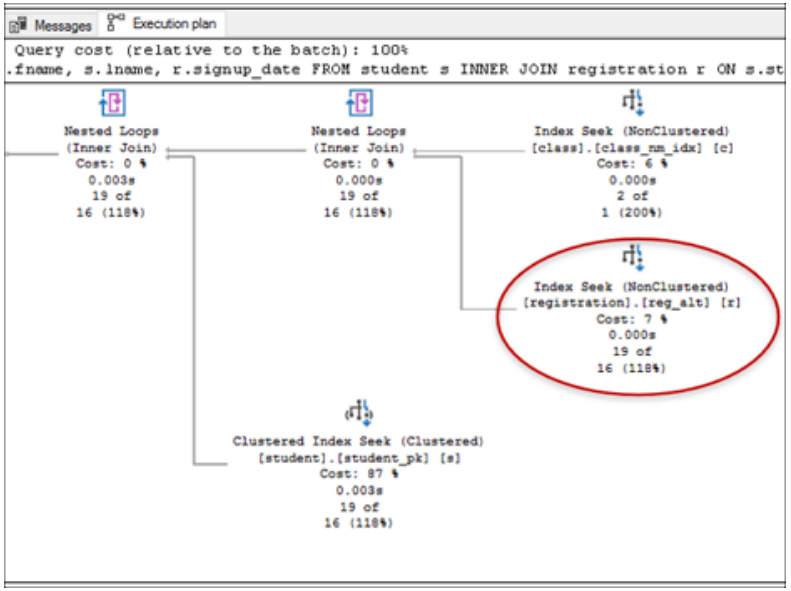


Figure 26: Execution plan (partial) after Fix 3

By `changing reg_alt` to `add signup_date` and `cancelled`, in descending order of specificity, the number of logical reads for `registration` falls from 400 to 6.

Average SQL Response time has gone from 5.2 down to 1.3 seconds, with 2.5x more executions (59 to 160) in the same timeframe. Also, the original query spent 77 percent of its time waiting on `ASYNC_NETWORK_IO;` it now spends most of its time on CPU.

Quest

Case study 2: Customer query

This case study takes you through the flaws in a query on the AdventureWorks database. It leads up to the fifth and final SQL query optimization tip: Identify performance inhibitors.

The query shown in Figure 27 **SELECT**s the same columns used for tip 2 and tip 3 above, but with different filters in the **WHERE** clause. It performs **INNER JOIN** on four tables.

The goal is to figure out the best execution plan without relying on the Query Optimizer in SQL Server. In other words, what is the most selective way to get the smallest amount of data first? And then, what is the most efficient way to build upon that data to satisfy the columns in the **SELECT** clause?

```
SELECT  c.CustomerID ,
        p.FirstName ,
        p.LastName ,
        oh.OrderID ,
        oh.OrderDate ,
        oh.SubTotal,
        pr.Name,
        pr.Color,
        od.OrderQty
FROM Sales.OrderHeader AS oh
        INNER JOIN Sales.Customer AS c ON c.CustomerID = oh.CustomerID
        INNER JOIN Person.Person AS p ON p.BusinessEntityID = c.PersonID
        INNER JOIN Sales.OrderDetail AS od ON od.OrderID = oh.OrderID
        INNER JOIN Production.Product AS pr ON pr.ProductID
= od.ProductID
WHERE oh.OnlineOrderFlag = 1
and p.LastName like 'C%'
and pr.Name like 'Mountain%42'
and pr.ProductID like '9%'
```

Figure 27: Case study 2 – Original query

MONITOR WAIT TIME

Wait time analysis yields these statistics:

Average SQL response time – 1.15 seconds

Executions – 439

Logical reads – 92,196,992

Logical reads are very high. While average SQL response time is low, executions are too.

REVIEW THE EXECUTION PLAN AND LOGICAL I/O

Per Figure 28, the execution plan shows more than 126,000 logical reads for `SalesOrderHeader` and almost 80,000 for `SalesOrderDetail`.

```
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0
Table 'Worktable'. Scan count 0, logical reads 0, physical reads
Table 'SalesOrderHeader'. Scan count 0, logical reads 126821, phy
Table 'SalesOrderDetail'. Scan count 1, logical reads 79270, phys
Table 'Product'. Scan count 1, logical reads 9, physical reads 0,
Table 'Customer'. Scan count 1, logical reads 157, physical reads
Table 'Person'. Scan count 1, logical reads 10, physical reads 0,
```

Figure 28: Logical I/O

On the right side of Figure 29, the execution plan shows a key lookup on `Product`. Why would a table that should have a primary key use a key lookup? That seems inefficient.

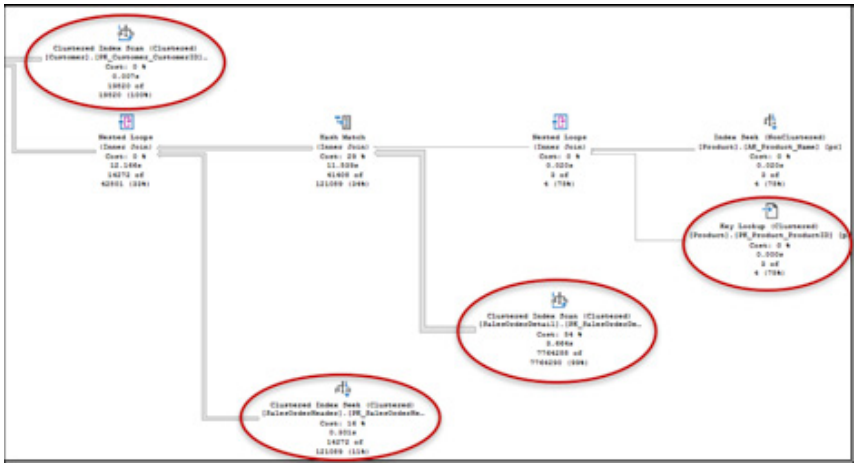


Figure 29: Execution plan

The clustered index scan on `SalesOrderDetail` (second red oval from right) shows the rather high cost of 54 percent. The next-highest cost of 16 percent is imposed by the clustered index seek on `SalesOrderHeader` (bottom oval).

Although the clustered index scan on `Customer` is low in cost, it's worth a look through the object information for a better access path.

GATHER OBJECT INFORMATION

Figure 30 shows that `SalesOrderDetail` has almost 8 million rows and `SalesOrderHeader` has about 2 million.

name	rows	type
Customer	19820	U
Person	19972	U
Product	504	U
SalesOrderDetail	7764288	U
SalesOrderHeader	2013760	U

Figure 30: Row count

The `exec sp_helpindex` queries (Figure 31) show that the column `product.name` has an index, as does the other selective column, `person.lastname`.

Results | Messages

	index_name	index_description	index_keys
1	AK_Product_Name	nonclustered, unique located on PRIMARY	Name
2	AK_Product_ProductNumber	nonclustered, unique located on PRIMARY	ProductNumber
3	AK_Product_rowguid	nonclustered, unique located on PRIMARY	rowguid
4	PK_Product_ProductID	clustered, unique, primary key located on PRIMARY	ProductID

	index_name	index_description	index_keys
1	AK_Person_rowguid	nonclustered, unique located on PRIMARY	rowguid
2	IX_Person_LastName_FirstName_MiddleName	nonclustered located on PRIMARY	LastName, FirstName, MiddleName
3	PK_Person_BusinessEntityID	clustered, unique, primary key located on PRIMARY	BusinessEntityID

	index_name	index_description	index_keys
1	AK_Customer_AccountNumber	nonclustered, unique located on PRIMARY	AccountNumber
2	AK_Customer_rowguid	nonclustered, unique located on PRIMARY	rowguid
3	IX_Customer_TerritoryID	nonclustered located on PRIMARY	TerritoryID
4	PK_Customer_CustomerID	clustered, unique, primary key located on PRIMARY	CustomerID

	index_name	index_description	index_keys
1	AK_SalesOrderDetail_rowguid	nonclustered, unique located on PRIMARY	rowguid
2	PK_SalesOrderDetail_SalesOrderID_SalesOrderDeta...	clustered, unique, primary key located on PRIMARY	SalesOrderID, SalesOrderDetailID

	index_name	index_description	index_keys
1	AK_SalesOrderHeader_rowguid	nonclustered, unique located on PRIMARY	rowguid
2	AK_SalesOrderHeader_SalesOrderNumber	nonclustered, unique located on PRIMARY	SalesOrderNumber
3	IX_SalesOrderHeader_CustomerID	nonclustered located on PRIMARY	CustomerID
4	IX_SalesOrderHeader_SalesPersonID	nonclustered located on PRIMARY	SalesPersonID
5	PK_SalesOrderHeader_SalesOrderID	clustered, unique, primary key located on ...	SalesOrderID

Figure 31: Indexes and keys

That means it is likely that the Optimizer will drive the query beginning with the name of the product or the last name of the customer.

FIND THE DRIVING TABLE

Use SQL diagramming to quickly find the driving table. As shown in Figure 32, `SalesOrderDetail` is the detailed table. It has lookups into `SalesOrderHeader` and `Product`. `SalesOrderHeader` has a lookup into `Customer`, which has a lookup into `Person`.

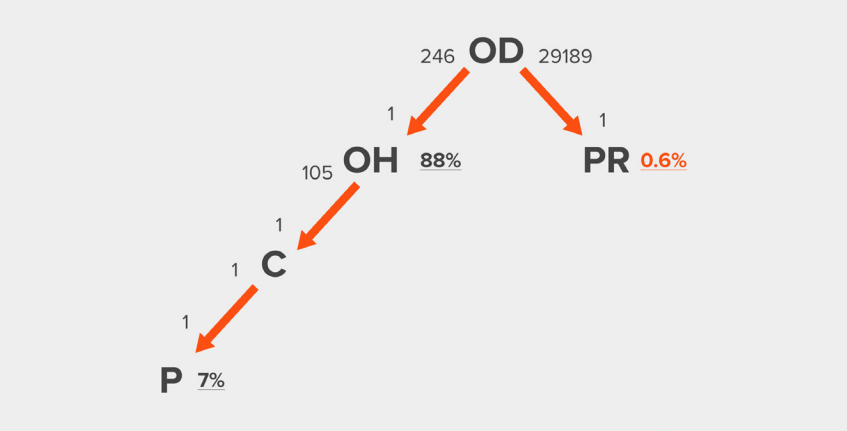


Figure 32: SQL diagramming

Now see how selective the filters are. The query

```
select count(1) from sales.SalesOrderHeader oh

where oh.OnlineOrderFlag = 1
```

returns 1,770,176 records out of 2,013,760 in SalesOrderHeader, or 88 percent, so that filter is not very selective. The query

```
select count(1) from person.Person p

where p.LastName like 'C%'
```

returns 1,386 records from 19,972 in Person, or 7 percent, which is much more selective. Finally, the query

```
select count(1) from production.Product  pr

where pr.Name like 'Mountain%42' and
pr.ProductID like '9%'
```

returns 3 records out of 504 in `Product`, or 0.6 percent. Thus, `Product` is the driving table for the query in this case study because it has the most selective filter.

Is there an index on `product.name`? The `exec sp_helpindex` query on all the indexes (see Figure 31) shows a clustered primary key on `ProductID`; however, it isn't being used. Instead the optimizer is using the alternative non-clustered index on name, `AK_product_name`.

Examination of the step on `Product` reveals that there is an implicit conversion on `ProductID` as shown in Figure 33.

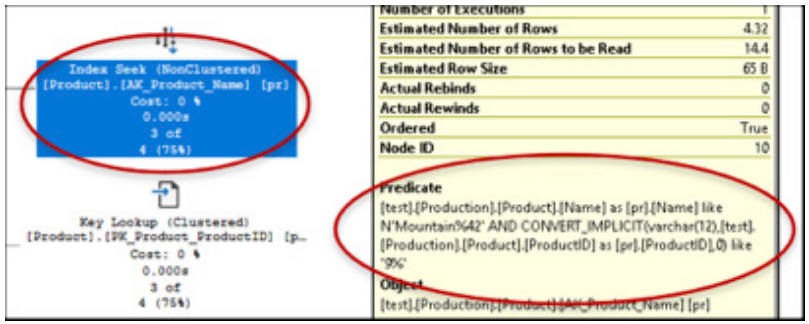


Figure 33: Execution plan – Implicit conversion

Quest

In the portion that reads:

```
WHERE oh.OnlineOrderFlag = 1

and p.LastName like 'C%'

and pr.Name like 'Mountain%42'

and pr.ProductID like '9%'
```

the parameter that is used to compare against the `ProductID` column is a varchar. However, `ProductID` is in fact an integer, so the optimizer has to change the integer to a varchar before it can perform any comparison. Furthermore, it can't use the clustered index because of the conversion.

Fixes for case study 2 are part of SQL query optimization tip 5.

Tip 5: Identify performance inhibitors

In fast-paced database development, it's understandable to take shortcuts or lose sight of the best way to engineer something. SQL query optimization is the opportunity to go back and evaluate the efficiency of a piece of code.

THINGS TO LOOK FOR

1. Look for cursor or row-by-row processing. Like any relational database, SQL Server works best on set processing and not on looping through each row.
2. Is the query processing in parallel? Does it really need to? If not, then it's consuming valuable resources that other queries may need.
3. Look for hard-coded hints, which are easily forgotten. If the underlying data changes, the hint becomes invalid.
4. Are you running nested views with a linked server? It's better to avoid linked servers and replicate the data if possible.
5. Watch for abuse of wild cards — as in `SELECT *`, described earlier.
6. Use `WHERE` clauses for filtering as early as possible in the query.
7. Beware of third-party SQL generators such as EMF, LINQ and NHibernate, which often produce sub-optimal code.
8. Avoid implicit data conversions, which can drive CPU cycles far above normal. If you find them, simply use the correct data type.
9. Don't turn off indexes. As described above, placing a function on an indexed column keeps SQL Server from using the index. For example, in `SELECT . . . WHERE` upper (`company_name`) = '`QUEST`', it would be better to move the upper() off of the indexed column and onto the literal being passed.

FIXES FOR CASE STUDY 2

To return to case study 2, an implicit conversion on `ProductID` was causing the optimizer to not use the clustered index. It happens that the `Product` table has only 504 records. Furthermore, the highest possible value in the `Product` column is 999, so the `9%` criterion would be better written as `>=900`. That way, SQL Server would not incur the implicit conversion and it would perform an index seek using the primary key instead of using the alternate non-clustered index, `AK_product_name`.

A NOTE ON INDEXES

Adding indexes is not always the right thing to do. If you need the performance on `INSERT`, `UPDATE` or `DELETE` statements, then indexes can actually hamper that performance because of the extra work the Optimizer must do to maintain the indexes. However, if you need better performance on `SELECT` statements, then you can choose from the following types of indexes:

Covering indexes reduce I/O because the Optimizer doesn't need to return to the table to get the data. The `INCLUDE` clause holds all the columns required in the query so it can get all the information out of the index. For example:

```
CREATE NONCLUSTERED INDEX
CIX_OrderHeader_OnlineOrderFlag

ON Sales.OrderHeader(OnlineOrderFlag)

INCLUDE (OrderID,OrderDate,CustomerID,SubTotal)
```

Quest

Filtered indexes use a `WHERE` clause to retrieve only a subset of data in a table. The indexes are usually very small because the index contains only the data that is in the filter. For example, the filtered index below would contain only those rows where `OnlineOrderFlag = 1`. If the table has 6 million rows, and only 100 rows have `OnlineOrderFlag = 1`, then the index would have only 100 entries.

```
CREATE NONCLUSTERED INDEX
FIX_OrderHeader_OnlineOrderFlag

ON Sales.OrderHeader(OnlineOrderFlag)

WHERE OnlineOrderFlag = 1
```

You can also **combine both index types, covering and filtered**. For example:

```
CREATE NONCLUSTERED INDEX
FcIX_OrderHeader_OnlineOrderFlag

ON Sales.OrderHeader(OnlineOrderFlag)

INCLUDE (OrderID,OrderDate,CustomerID,SubTotal)

WHERE OnlineOrderFlag = 1
```

COVERING INDEXES FOR CASE STUDY 2

Figure 29 above showed that `SalesOrderDetail` is the highest-cost step in the query. Add a covering index as follows:

```
CREATE NONCLUSTERED INDEX CIX_OrderDetail_ProductID

ON Sales.SalesOrderDetail(ProductID)

INCLUDE (OrderID,OrderQty)
```

Figure 34 shows the new execution plan.

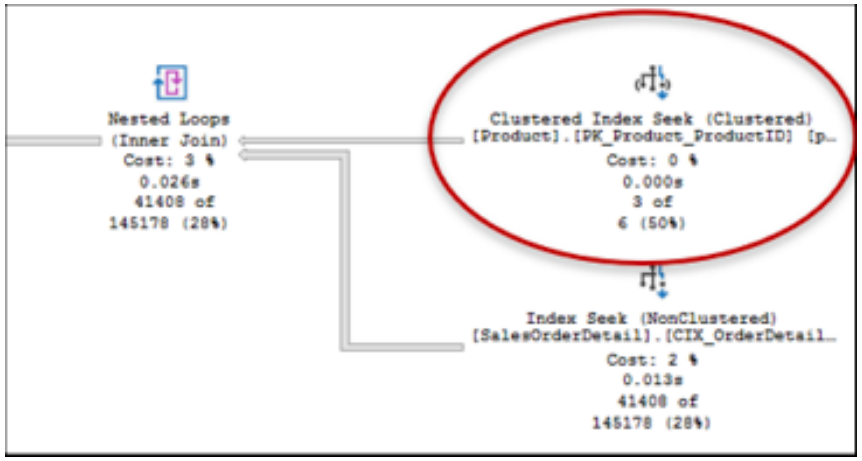


Figure 34: Execution plan (partial) with covering index

Note that the order of the steps and the Join methods have again changed. `Product` (red oval) is again the driving table; it is the most selective so it should be first. `Product` does a nested loop Join into `SalesOrderDetail` using an index seek.

Furthermore, logical I/O has fallen, as depicted in Figure 35.

```
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0,
Table 'SalesOrderHeader'. Scan count 0, logical reads 126821, physi
Table 'SalesOrderDetail'. Scan count 3, logical reads 116, physical
Table 'Product'. Scan count 1, logical reads 6, physical reads 0, p
Table 'Customer'. Scan count 1, logical reads 157, physical reads 0,
Table 'Person'. Scan count 1, logical reads 10, physical reads 0, p
```

Figure 35: Logical I/O with covering index on SalesOrderDetail

The number of logical reads on `SalesOrderDetail` falls from 79,270 to 116.

As a final tuning step, try to lower the number of logical reads (126,821) shown in Figure 28. In the execution plan, SQL Server suggested adding a covering index for `SalesOrderHeader` also. Add the following to the query:

```
CREATE NONCLUSTERED INDEX
CIX_OrderHeader_OnlineOrderFlag

ON Sales.OrderHeader(OnlineOrderFlag)

INCLUDE (OrderID,OrderDate,CustomerID,SubTotal)
```

Logical I/O has fallen for `SalesOrderHeader` also, as shown in Figure 36.

```
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page ser
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page se
Table 'SalesOrderHeader'. Scan count 1, logical reads 6798, physical reads
Table 'SalesOrderDetail'. Scan count 3, logical reads 116, physical reads 0
Table 'Product'. Scan count 1, logical reads 6, physical reads 0, page serv
Table 'Customer'. Scan count 1, logical reads 157, physical reads 0, page s
Table 'Person'. Scan count 1, logical reads 10, physical reads 0, page serv
```

Figure 36: Logical I/O with covering index on SalesOrderHeader

The number of logical reads on `SalesOrderHeader` has fallen from 126,821 to 6,798.

IMPACTS ON OVERALL PERFORMANCE

Wait time analysis now shows improvement in three main areas:

Average SQL response time – from 1.15 seconds to 0.38 seconds (66 percent improvement)

Executions – from 439 to 629 (43 percent improvement)

Logical reads – from 92,196,992 to 4,457,723 (95 percent improvement)

Conclusion

The five SQL query optimization tips in this e-book comprise a method for tuning your SQL Server queries for higher speed and better performance. By monitoring wait time, reviewing the execution plan, gathering object information, finding the driving table and identifying performance inhibitors, database professionals like you can improve performance in your database environment.

It's important to be able to prove quantitatively that tuning made a difference. Regular measurements help you keep your fingers on the pulse of performance. They also keep you mindful of the next tuning opportunity. SQL query optimization is, after all, iterative and there is always room for improvement. The more you optimize, the more you can be sure you are tuning things that make a difference.

ABOUT THE AUTHOR

Janis Griffin is a senior systems consultant at Quest, where she specializes in performance tuning and analyzes database performance. A database administrator with over 30 years of experience, Janis started out on Oracle version 3. Most of her expertise is in Oracle, SQL Server and MySQL.

ABOUT QUEST

Quest provides software solutions for the rapidly changing world of enterprise IT. We help simplify the challenges caused by data explosion, cloud expansion, hybrid data centers, security threats and regulatory requirements. We're a global provider to 130,000 companies across 100 countries, including 95% of the Fortune 500 and 90% of the Global 1000. Since 1987, we've built a portfolio of solutions which now includes database management, data protection, identity and access management, Microsoft platform management and unified endpoint management. With Quest, organizations spend less time on IT administration and more time on business innovation. For more information, visit www.quest.com.

If you have any questions regarding your potential use of this material, contact:

Quest Software Inc.
Attn: LEGAL Dept
4 Polaris Way
Aliso Viejo, CA 92656

Refer to our website (www.quest.com) for regional and international office information.

eBook-FundamentalGuideSQL-US-JY-59506

Quest