

Finding Queries That Kill Your System: A Quick Strategy – The Simplified Version

TechNote Number: SQL-DISK-003

Revision 1.0a

Frank McBath

frankmcb@computationpress.com

Publication Date: 08 February 2008

Abstract

The most difficult issue in finding performance issues usually revolves around just defining what the problem is in the first place. This article will focus on how to use Profiler to capture the necessary data and comb through it to define where the issues are and then using the Database Tuning Advisor (DTA) to solve them.

Introduction

There are two ways to find long running/resource intensive queries on a system. The first is to use dynamic management views (DMVs) provided in SQL Server 2005, the second is to use Profiler.

In SQL Server 2005, there are a variety of new DMVs that provide insight into how/why your system is running slow. The main drawback to them are the fact that they provide a “macro” view of the system and can lack the ability to pinpoint exactly where issues are at. For example, a DMV cannot be initialized at the beginning of a job to “reset” the counters for a baseline or that the data collected in the DMV may reflect changes since yesterday or a week ago depending on the available memory to store the information.

While DMVs are great tools, they are still not a substitute for SQL Server Profiler. Profiler allows you to capture data over a user defined period of time to nail down a specific job or user. This flexibility lets the DBA get a more granular picture of the situation or, if the trace is run for a day, a macro view of your entire system.

Collecting the Data

In the Oracle applications space (JD Edwards, PeopleSoft, Siebel), all the queries are dynamically generated by the application and show up as JDBC/ODBC calls only. The only event data needed to capture these queries are RPC:Completed and SQL:BatchCompleted. To find the issues and define the problem, we need to run Profiler and then aggregate the data up after it’s finished.

To launch Profiler:

1. Start to All Programs
2. Navigate to SQL Server 2005
3. Select Performance Tools
4. Launch SQL Server Profiler

Next, define a trace:

1. Select File
2. New Trace
3. Login to the appropriate server and database, you will need administrator privileges to run a trace
4. Click on the “Events Selection” tab
5. Only select the following events:
 - a. RPC:Completed

b. SQL:BatchCompleted

6. De-select all others. They are not needed.
7. Take all the default columns. Don't touch those.
8. Hit the "Column Filters" button.
9. Set Reads to Greater than or equal to 5000 (These are 40MB queries = 5000 * 8K pages.)
10. Select the "OK" button
11. Select the "Run" button

Now, let the trace run for an hour during a normal user work day. For example, 2PM to 3PM. Or, if there is specific job that needs to be debugged, trace only for the duration of that run.

Once the hour is over, stop the trace:

1. Select File
2. Stop Trace

And save the trace to a file:

1. Select File
2. Save As... Trace File...

This allows us more flexibility in the future. Next, save the trace data to a database table. This will allow us to aggregate up the data and find the problems:

1. Select File
2. Save As... Trace Table...

Save the data into a scratch database and table. I usually call the table "trace". If the table already exists, SQL Server will prompt to overwrite it.

Analyze the Data

Now that the data is in a table, we can start scrubbing it. The main issues with Profiler traces are that people get lost fast in all the data. Often they are thousands of rows in single file. Your objective is to weed out the extraneous information, prioritize the top 10 queries to fix, and then look for repeating patterns which will show you which queries are being run over and again. Usually you will find the 80/20 rule applies. Twenty percent of your queries are doing eighty percent of your IO.

1. Get a reference time frame for the period monitored:

```
select min(StartTime) 'Start', max(StartTime) 'End',
```

```
datediff(mi,min(StartTime), max(StartTime)) 'Minutes Monitored'
from trace
where StartTime is not NULL
go
```

- Here's where we define the problems to solve. Figure out what the Top 10 worst queries are by Reads, CPU, Writes, and Duration. These are the low hanging fruit that are usually table scans, etc... I tend to focus on Reads as this is what 90% of OLTP systems are.

```
select 'Top 10 LRQ by Duration'
go
```

```
select top 10 RowNumber, Duration, Reads, Writes, CPU
from trace
where Duration is not NULL
order by Duration desc
go
```

```
select 'Top 10 LRQ by Reads'
go
```

```
select top 10 RowNumber, Duration, Reads, Writes, CPU
from trace
where Reads is not NULL
order by Reads desc
go
```

```
select 'Top 10 LRQ by Writes'
go
```

```
select top 10 RowNumber, Duration, Reads, Writes, CPU
from trace
where Writes is not NULL
order by Writes desc
go
```

```
select 'Top 10 LRQ by CPU'
go
```

```
select top 10 RowNumber, Duration, Reads, Writes, CPU
from trace
where CPU is not NULL
order by CPU desc
go
```

- Now look at the data by how often the same query is repeated by looking at number of Reads. What you are looking for is repeating patterns. If you see something has been run over and over and has the same number of reads (+/- 10%), chances are it's the same query. So if you can find that one query, and fix it, you have solved many problems at one time. One method for looking at trends is to dump the results into Excel and then graph it.

```

SELECT top 100 COUNT(*) 'Number Times Run', Reads, (Reads * 8192.) 'Bytes
Read per Query'
from trace
where Reads is not NULL
GROUP BY Reads
ORDER BY Reads desc
go

```

Sample output:

Number Times Run	Reads	Bytes Read per Query
1	4309177	35300777984
1	4251641	34829443072
1	3559320	29157949440
1	3558897	29154484224
1	3557698	29144662016
1	3529061	28910067712
2	3435668	28144992256
1	3398073	27837014016
1	3225342	26422001664
1	3213913	26328375296
1	3213364	26323877888
2	3205498	26259439616
1	3187905	26115317760
1	3187782	26114310144
1	3186815	26106388480
1	3180706	26056343552
1	3171847	25983770624
1	3161991	25903030272
1	3151664	25818431488
1	3146256	25774129152
1	3145443	25767469056
1	3141366	25734070272
1	3140949	25730654208
1	3138458	25710247936
1	3134655	25679093760
1	3132166	25658703872
1	3132038	25657655296
1	3131357	25652076544

What you see in the above is one query run over and over with Reads in the 3.1M to 3.5M per iteration.

Fixing the Query

To actually find the syntax for the query, open the trace file up in Profiler and then search on the number of Reads:

1. File... Open... Trace File...
2. Edit... Find...
 - a. In the "Find what:" section, cut and paste the number of Reads for the query we are trying to find. For example, in the above: 3131357.
 - b. In the "Search in column:" section, select Reads.
 - c. Hit the "Find Next" button.

- d. It will now show you the query in the TextData section. The query will look something like this:

```

declare @P1 int
set @P1=89
declare @P2 int
set @P2=180150132
declare @P3 int
set @P3=8
declare @P4 int
set @P4=1
declare @P5 int
set @P5=1
exec sp_cursorprepexec @P1 output, @P2 output, N'@P1 char(30),@P2
char(2),@P3 char(30),@P4 char(2)', N'SELECT ILITM, ILLITM, ILAITM,
ILMCU, ILLOCN, ILLOTN, ILLOTS, ILLOTP, ILLOTG, ILMMCU, ILKCO, ILDOC,
ILDCT, ILDGL, ILGLPT, ILDCO, ILKCOO, ILTRDJ, ILTRUM, ILTRQT,
ILUNCS, ILPAID, ILUKID FROM PRODDTA.F4111 WHERE ( ( ILLOTN = @P1 AND
ILDCT = @P2 OR ILLOTN = @P3 AND ILDCO = @P4 ) ) ORDER BY ILUKID ASC',
@P3 output, @P4 output, @P5 output, '5670957',
'IK', '5670957', 'IF'
select @P1, @P2, @P3, @P4, @P5

```

Problem Query: With ODBC Wrapper and Parameterized

3. Cut and paste this query into Management Studio.
4. To get a “back of the envelope” fix, strip out the ODBC “wrapper” from the query and hardcode the variables, then save it so an “.SQL” file as seen in the following JD Edwards example:

```

SELECT ILITM, ILLITM, ILAITM, ILMCU, ILLOCN, ILLOTN, ILLOTS, ILLOTP,
ILLOTG, ILMMCU, ILKCO, ILDOC, ILDCO, ILDGL, ILGLPT, ILDCO, ILKCOO,
ILTRDJ, ILTRUM, ILTRQT, ILUNCS, ILPAID, ILUKID
FROM PRODDTA.F4111
WHERE ( ( ILLOTN = '5670957' AND ILDCO = 'IK'
OR ILLOTN = '5670957' AND ILDCO = 'IF' ) )
ORDER BY ILUKID ASC

```

Problem Query: Stripped and Hard Coded

5. From Profiler, start the Database Tuning Advisor:
 - a. Tools... Database Tuning Advisor
6. Log onto a test database where we can debug.
7. In the “Workload” section of the screen, provide the directory and file name from step number four above.
 - a. Select the database for workload analysis on the drop down.
 - b. Select the check box for the database in the section. In this example, it would be the PRODDTA database.

8. From the menu, select
 - a. Actions... Start Analysis
9. SQL Server will make several passes at the data and show a status. In the final action, it will say "Generating Recommendations".
 - a. SQL Server will then provide you with an estimate as to the overall improvement and a list of indexes and statistics to apply to fix the query.
10. I generally save the recommendations to a script and then apply them. This allows you to have "build scripts" when rolling out new changes.
 - a. Actions... Save Recommendations
 - b. Actions... Apply Recommendations
11. Finally, take the indexes that have been recommended and add them back into the application's repository. For example, use PeopleTools and add the index into it's meta-data.

Note: As long as the query is not run as "sp_cursorfetch" or "sp_cursorexecute" then the above methodology works fine providing there's not an implicit cursor conversion issue. If the query is a fetch or execute, then additional analysis needs to be performed to find the originating SQL.

Summary

Finding issues is always a taxing problem. Usually the sheer quantity of data and the complexity of the tools can overwhelm people in deciding just where to start. This article has provided a simple and straightforward framework to prioritize and attack problems using SQL Server's robust set of tools: Profiler, Management Studio, and Database Tuning Advisor.

While in this method explained above, we were focusing on 40MB queries, the reality is that you can tweak the process based on duration or lowering the reads. A good case for using duration would be if you were debugging a batch job where anything over 500ms would be considered an issue.