**Troubleshooting CPU Issue**

1. Check for below counters in PerfMon.

**Processor/ %Privileged Time** *– percentage of time the processor spends on execution of Microsoft Windows kernel commands such as core operating system activity and device drivers.*

**Processor/ %User Time** *– percentage of time the processor spends on executing user processes such as SQL Server. This includes I/O requests from SQL Server.*

**Process (sqlservr.exe)/ %Processor Time** *– the sum of processor time on each processor for all threads of the process.*

• **SQLServer:SQL Statistics/Auto-Param Attempts/sec**

• **SQLServer:SQL Statistics/Failed Auto-params/sec**

• **SQLServer:SQL Statistics/Batch Requests/sec**

• **SQLServer:SQL Statistics/SQL Compilations/sec**

• **SQLServer:SQL Statistics/SQL Re-Compilations/sec**

• **SQLServer:Plan Cache/Cache hit Ratio**

1. Investigate high CPU usage by SQL Server by examining the CPU-related wait statistics, the scheduler details and the aggregated query performance statistics, as follows:

• Verifying the extent of CPU pressure via signal waits, using **sys.dm\_os\_wait\_stats.**

**SELECT \* FROM SYS.OS\_DM\_WAIT\_STATS ORDER BY SIGNAL\_WAI\_TIME DESC**

**Look for these 3 waits in the result set of above query:**

* **SOS\_SCHEDULER\_YEILD**
* **CXPACKET**
* **Thread Pool**

• Diagnosing a CPU-bound system according the types of wait observed, using sys.dm\_os\_wait\_statsand sys.dm\_os\_schedulers.

* Identifying high-CPU cached plans, and associated queries, using **sys.dm\_exec\_**

**query\_statsand sys.dm\_exec\_sql\_text.**

* Identifying currently waiting tasks, especially ones waiting on CPU-related wait typesusing **sys.dm\_os\_waiting\_tasks**.
* Observing the resource usage of currently executing queries with **sys.dm\_exec\_requests**.

1. Clear waits using following command before you start gathering waits.

**DBCC SQLPERF('sys.dm\_os\_wait\_stats', CLEAR);**

1. ***Command:*** *Verifying CPU pressure via signal wait time.*

--- Verifying CPU pressure via signal wait time.

SELECT SUM(signal\_wait\_time\_ms) AS TotalSignalWaitTime,

(SUM(CAST(signal\_wait\_time\_ms AS NUMERIC(20, 2)))

/SUM(CAST(wait\_time\_ms AS NUMERIC(20, 2)))\* 100 )

AS PercentageSignalWaitsOfTo

Signal wait time

Along with a wait\_typecolumn, indicating the type of wait, the sys.dm\_os\_wait\_statsDMV returns several useful wait times, including:

• **wait\_time\_ms**– total amount of time that tasks have waited on this given wait type;this value includes the time in the signal\_wait\_time\_mscolumn. The value incrementsfrom the moment a task stops execution, to wait for a resource, to the point itresumes execution.

**signal\_wait\_time\_ms**– the total amount of time tasks took to start executing after being signalled (i.e. after the resource it was waiting for became available); this is time spent on the runnable queue, and is pure CPU wait.

If the signal wait time is a significant portion of the total wait time, it means that tasksa re waiting a relatively long time to resume execution after the resources that they were waiting for became available. This can indicate either that there are lots of CPU-intensive queries, which may need optimizing, or that the server needs more CPU.

***Note:* compare the wait stats at a particular time with the stats at an earlier time and see how they changed.**

1. Identify the top wait events, ordered according to the total amount of time processes have waited (wait\_time\_ms) on this event. It is important to ignore the benign waits, typically ones caused by system processes that are expected to be waiting most of the time. We're also subtracting out the signal\_wait\_time as that portion of the wait time is not waiting for the particular resource, but waiting for time on the scheduler.

***Command: Finding the top 10 wait events (cumulative)***

SELECT TOP ( 10)

wait\_type,

waiting\_tasks\_count,

(wait\_time\_ms-signal\_wait\_time\_ms)ASresource\_wait\_time,

max\_wait\_time\_ms,

CASE waiting\_tasks\_count

WHEN 0 THEN 0

ELSE wait\_time\_ms/waiting\_tasks\_count

END AS avg\_wait\_time

FROM sys.dm\_os\_wait\_stats

WHERE wait\_type NOT LIKE '%SLEEP%'-- remove eg. SLEEP\_TASK and

-- LAZYWRITER\_SLEEP waits

AND wait\_type NOT LIKE 'XE%'

AND wait\_type NOT IN-- remove system waits

('KSOURCE\_WAKEUP','BROKER\_TASK\_STOP','FT\_IFTS\_SCHEDULER\_IDLE\_WAIT',

'SQLTRACE\_BUFFER\_FLUSH','CLR\_AUTO\_EVENT','BROKER\_EVENTHANDLER',

'BAD\_PAGE\_PROCESS','BROKER\_TRANSMITTER','CHECKPOINT\_QUEUE',

'DBMIRROR\_EVENTS\_QUEUE','SQLTRACE\_BUFFER\_FLUSH','CLR\_MANUAL\_EVENT',

'ONDEMAND\_TASK\_QUEUE','REQUEST\_FOR\_DEADLOCK\_SEARCH','LOGMGR\_QUEUE',

'BROKER\_RECEIVE\_WAITFOR','PREEMPTIVE\_OS\_GETPROCADDRESS',

'PREEMPTIVE\_OS\_AUTHENTICATIONOPS','BROKER\_TO\_FLUSH','XTP\_PREEMPTIVE\_TASK','WAIT\_XTP\_OFFLINE\_CKPT\_NEW\_LOG','DIRTY\_PAGE\_POLL','HADR\_FILESTREAM\_IOMGR\_IOCOMPLETION','WAIT\_XTP\_HOST\_WAIT')

ORDER BY wait\_time\_ms DESC

Interesting wait types to look out for, in regard to CPU pressure, are **SOS\_**

* **SCHEDULER\_YIELD ,CXPACKET and Thread Pool**

**SQLOS**

**Scheduling**

**Schedulers**

**Scheduling Algorithm**

**Cooperative Scheduling Algorithm**

**Pre-emptive Scheduling Algorithm**

**Task**

**Scheduler**

**WORKER**

**MAXDOP=1**

**Cost Threshold for Parallelism**

**Thread**

**Thread Life Cycle**

**Waiter\_List->Runnable->Running->Suspended**

**---session id:62**

**Begin tran**

**Update sales.salesorederdetail set unitprice=2000.00 where salesorederid=345678**

**Commit ---X lock**

**owe**

**---session id:64**

**Begin tran**

**Update sales.salesorederdetail set unitprice=2020.00 where salesorederid=345678**

**Commit**

**Signal\_wait\_time\_ms**

**Suspended**

**MAXDOP**

**Cost Threshold for Parallelism 25**

**Select \* from sys.dm\_os\_wait\_stats oder by signal\_wai\_time**

Select \* from sys.dm\_os\_wait\_stats order by signal\_wait\_time\_ms desc

--CXPACKET----denotes parallelism

--if CXPACKET wait value is high in that case we have to consider changing the following

--1. Cost Threshold for Parallelism(if it is set to 5 then increase it to 30-40)

----2. Check if MAXDOP value is set to 0(which means all processors;

--- if so then assign the half number of CPUs for MAXDOP vale

---or if there are 16 CPUs then not assign more than 8 CPUs for MAXDOP value

---or if there are 32 CPUs then not assign more than 16 CPUs for MAXDOP value

---if CXPACKET vale is too high it can increase the SOS\_SCHEDULER\_YIELD wait value too

---SOS\_SCHEDULER\_YIELD----how much time each thread has to wait to switch from Runnable to Running state

---if SOS\_SCHEDULER\_YIELD value is high then it certainly denotes CPU pressure

1. **Investigating scheduler queues:** The **sys.dm\_os\_schedulers** DMV can identify whether or not a SQL instance is CPU-bound. This DMV returns one row for each of the SQL Server schedulers and it lists the total number of tasks that are assigned to each scheduler, as well as the number that are runnable. A runnable task is one that is in the runnable queues, waiting for CPU time. Other tasks on the scheduler that are in the **current\_tasks\_count** but not the **runnable\_tasks\_count** are ones that are either sleeping or waiting for a resource (lock, latch, I/O, memory, and so on).

SELECT scheduler\_id,

current\_tasks\_count,

runnable\_tasks\_count

FROMsys.dm\_os\_schedulers

WHERE scheduler\_id< 255

1. **Identifying CPU-intensive queries:** In order to determine the worst-running queries in the plan cache of SQL Server, the DMVs **sys.dm\_exec\_query\_stats** and **sys.dm\_exec\_sql\_text** can be used.

The sys.dm\_exec\_query\_statsDMV provides aggregated statistics and returnsone row for each query statement in the cached plan. Many of the columns are incrementedcounters, and provide information about how many times a query has beenexecuted and the resources that were used. For example, the **\*\_worker\_time** columns represent the time spent on the CPU, and the **\*\_elapsed\_time** columns show the total execution time.

The query below returns the top ten most costly queries in cache by totalworker time. We join to the **sys.dm\_exec\_sql\_text** and **sys.dm\_exec\_query\_plan** functions to retrieve the text and execution plans for these queries, withinthe batch.

SELECT TOP ( 10)

SUBSTRING(ST.text,(QS.statement\_start\_offset/ 2 )+ 1,

((CASE statement\_end\_offset

WHEN-1 THEN DATALENGTH(st.text)

ELSE QS.statement\_end\_offset

END-QS.statement\_start\_offset)/ 2 )+ 1)

AS statement\_text,execution\_count,

total\_worker\_time/ 1000 AS total\_worker\_time\_ms,

(total\_worker\_time/ 1000 )/execution\_count

AS avg\_worker\_time\_ms,

total\_logical\_reads,

total\_logical\_reads/execution\_count AS avg\_logical\_reads,

total\_elapsed\_time/ 1000 AS total\_elapsed\_time\_ms,

(total\_elapsed\_time/ 1000 )/execution\_count

AS avg\_elapsed\_time\_ms,

qp.query\_plan

FROM sys.dm\_exec\_query\_stats qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) st

CROSS APPLY sys.dm\_exec\_query\_plan(qs.plan\_handle)qp

ORDER BY total\_worker\_time DESC

--Find top 5 queries by average CPU time

SELECT TOP 5 total\_worker\_time/execution\_count AS [Avg CPU Time],

Plan\_handle,query\_plan

FROM sys.dm\_exec\_query\_stats AS qs

CROSS APPLY sys.dm\_exec\_query\_plan(qs.plan\_handle)

ORDER BY total\_worker\_time/execution\_count DESC;

1. Queries to find duplicate plans and clear them
2. First find duplicate plans and look for same query hash. If there are same query hash it means they represent  different sql handles and plan handles but for same query .

WITH RedundantQueries AS

(SELECT top 1000 query\_hash, statement\_start\_offset, statement\_end\_offset,

/\* PICK YOUR SORT ORDER HERE BELOW: \*/

COUNT(query\_hash) AS sort\_order, --queries with the most plans in cache

/\* Your options are:

COUNT(query\_hash) AS sort\_order, --queries with the most plans in cache

SUM(total\_logical\_reads) AS sort\_order, --queries reading data

SUM(total\_worker\_time) AS sort\_order, --queries burning up CPU

SUM(total\_elapsed\_time) AS sort\_order, --queries taking forever to run

\*/

COUNT(query\_hash) AS PlansCached,

COUNT(DISTINCT(query\_hash)) AS DistinctPlansCached,

MIN(creation\_time) AS FirstPlanCreationTime,

MAX(creation\_time) AS LastPlanCreationTime,

MAX(s.last\_execution\_time) AS LastExecutionTime,

SUM(total\_worker\_time) AS Total\_CPU\_ms,

SUM(total\_elapsed\_time) AS Total\_Duration\_ms,

SUM(total\_logical\_reads) AS Total\_Reads,

SUM(total\_logical\_writes) AS Total\_Writes,

SUM(execution\_count) AS Total\_Executions,

--SUM(total\_spills) AS Total\_Spills,

N'EXEC sp\_BlitzCache @OnlyQueryHashes=''0x' + CONVERT(NVARCHAR(50), query\_hash, 2) + '''' AS MoreInfo

FROM sys.dm\_exec\_query\_stats s

GROUP BY query\_hash, statement\_start\_offset, statement\_end\_offset

ORDER BY 4 DESC)

SELECT r.query\_hash, r.PlansCached, r.DistinctPlansCached, q.SampleQueryText, q.SampleQueryPlan, r.MoreInfo,

r.Total\_CPU\_ms, r.Total\_Duration\_ms, r.Total\_Reads, r.Total\_Writes, r.Total\_Executions, --r.Total\_Spills,

r.FirstPlanCreationTime, r.LastPlanCreationTime, r.LastExecutionTime, r.statement\_start\_offset, r.statement\_end\_offset, r.sort\_order

FROM RedundantQueries r

CROSS APPLY (SELECT TOP 1000 st.text AS SampleQueryText, qp.query\_plan AS SampleQueryPlan, qs.total\_elapsed\_time

FROM sys.dm\_exec\_query\_stats qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) AS st

CROSS APPLY sys.dm\_exec\_query\_plan(qs.plan\_handle) AS qp

WHERE r.query\_hash = qs.query\_hash

AND r.statement\_start\_offset = qs.statement\_start\_offset

AND r.statement\_end\_offset = qs.statement\_end\_offset

ORDER BY qs.total\_elapsed\_time DESC) q

ORDER BY r.sort\_order DESC, r.query\_hash, q.total\_elapsed\_time DESC;

**(II) Find the plan handle for the given query hash.**

select \* FROM sys.dm\_exec\_query\_stats where query\_hash=0xAF9D12289B855259 order by total\_worker\_time desc

**(III)Start clearing plans from plan cache by supplying their plan handle on by one.**  
dbcc FREEPROCCACHE (0x060008003C8193184000C8C1D302000001000000000000000000000000000000000000000000000000000000)

The execution plan can be viewed by clicking the XML link to open the plan in itsgraphical form. The execution plan returned is for the entire batch, not just thehigh-CPU statement.

When querying the plan cache in order to investigate sub-optimal plans, note that somequeries may not be listed. Execution plans are retained in cache until SQL Server decidesthat the plan has aged to a point where it should be removed to allow for new executionplans to be cached, or the cache is fully or partially cleared by DBCC commands, databasestate changes (restoring a database, detaching or taking a database offline, and so on) orcertain server-wide configuration changes, or SQL Server is restarted. SQL Server willalso remove execution plans from the cache if it finds that extra memory is requiredelsewhere in the system.

***Clearing the plan cache***

*You can flush all plans from the cache using,* **DBCC FREEPROCCACHE***, or supply a* **plan\_handle***or* **sql\_handle***to remove a plan for a specific batch. Alternatively****,* DBCC FREEPROCINDB(db\_id)***can be used to remove plans for a specific database.*

It's also important to note that some queries may never appear in the cache at all.Procedures marked WITH RECOMPILE and queries with the hint OPTION (RECOMPILE)are never cached. Also, the query stats for a query are cleared when the query recompilesfor any reason, for example due to changing statistics or schema changes. As a result,queries that are subject to many recompilation events may also show a very low total forelapsed time, because that total is only for the current plan, which may not have been incache very long.

**Common Causes of High CPU Usage**

1. Missing Indexes
2. Outdated statistics
3. Non-SARGable predicates
4. Implicit conversions
5. Bad Parameter sniffing
6. Ad hoc non-parameterized queries
7. Inappropriate parallelism
8. **Missing Indexes**

Lack of appropriate indexing is one of the most common causes of heavyCPU and I/O utilization in SQL Server. When an appropriate index doesn't exist to satisfya query, the table scans that result can cause significant CPU usage, as SQL has to readand process far, far more rows than is actually necessary to satisfy the query.

Using the execution plans obtained from the plan cache(used in previous query, wecan find any operations that can be replaced with more efficient operations, by addingcovering indexes, or in some cases changing the indexes that are in place.

Let's consider asimple query against the Adventureworks2008 database, as shown in below listing.

SET STATISTIC IO ON

**----SETSTATISTICSTIMEON------**

SELECT per.FirstName,

per.LastName,

p.Name,

p.ProductNumber,

OrderDate,

LineTotal,

soh.TotalDue

FROM Sales.SalesOrderHeader AS soh

INNER JOIN Sales.SalesOrderDetail sod

ON soh.SalesOrderID=sod.SalesOrderID

INNER JOIN Production.Product AS p ON sod.ProductID=p.ProductID

INNER JOIN Sales.Customer AS c ON soh.CustomerID=c.CustomerID

INNER JOIN Person.Person AS per

ON c.PersonID=per.BusinessEntityID

WHERE LineTotal> 25000

This query causes a table scan on the SalesOrderDetailtable, as there is no index onthe LineTotalcolumn. The execution characteristics (with all the necessary pages inthe data cache, so no I/O waits) are as follows:

SQL Server parse and compile time:

CPU time = 16 ms, elapsed time = 19 ms.

(3 row(s) affected)

SQL Server Execution Times:

CPU time = 78 ms, elapsed time = 66 ms.

Now let's add a simple index.

CREATE NONCLUSTERED INDEX idx\_SalesOrderDetail\_LineTotal

ON Sales.SalesOrderDetail(LineTotal)

Now, if we run the same query again, the performance characteristics will be verydifferent:

SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server parse and compile time:

CPU time = 16 ms, elapsed time = 20 ms.

(3 row(s) affected)

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 1 ms.

1. ***Outdated statistics***

The SQL Server Optimizer uses statistics to calculate the estimated cardinality forvarious query operators. That cardinality, essentially number of rows, affects the cost ofthe operators. The cost of the operators, in turn, determines the cost of the plan. If thecardinality estimation is wrong, because of outdated statistics, the cost that the optimizercalculates for the operators will also be wrong, leading the optimizer to select a plan thathas a low estimated cost, but a very high actual cost when it is executed.

The most common side effect of incorrect statistics is that the optimizer estimates on thelow-side for the number of rows, and so chooses operators that are very good for smallnumbers of rows, such as nested loop joins and key lookups. When the query is executedand it turns out that a large number of rows need to be processed, the chosen operatorsscale badly and the plan is highly inefficient.

One way to tell if there is a problem with statistics for a particular query, is to runthe query in Management Studio, return the actual execution plan and examine the estimated and actual row counts for any index seek and scan operations within the execution plan. If the two counts are significantly different, bearing in mind that the estimated count is per execution of the operator and the actual count is a total for all executions of the operator, then one possibility is that the statistics are outdated.

**Example: Returning all statistics information**

----for all statistics of all indexes

SELECT OBJECT\_NAME(object\_id)AS [ObjectName]

,[name] AS [StatisticName]

,STATS\_DATE([object\_id], [stats\_id])AS [StatisticUpdateDate]

FROM sys.stats;

The following example returns all statistics information for the IXC\_ServerChecks\_Results\_LoadID index of the ServerChecks.Results table.

DBCC SHOW\_STATISTICS ([Sales.SalesOrderDetail], PK\_SalesOrderDetail\_SalesOrderID\_SalesOrderDetailID);

select count(\*) from sales.SalesOrderDetail where SalesOrderID between 43659 and 43692

**Fixing outdated statistics**

Fixing outdated statistics is done via the **UPDATE STATISTICS** statement. This can be run for all the statistics on a table (UPDATE STATISTICS <Table name>) or just for onespecific statistics set (UPDATE STATISTICS <Table name><statistic name>).

If the problem is due to stale statistics, i.e. an update of the statistics fixes the problem,then you need to prevent a recurrence of the problem, and there are three ways to do it:

1. If the database setting, **Auto\_Update\_Statistics**, is off, consider turning it on.Alternatively, a database-wide statistics update job can be run on a regular basis.

2. If automatic updates are disabled for a particular index of set of statistics, as a result of rebuilding the index with the NORECOMPUTE option, they should be enabled.

3. A job can be created that updates the specific statistics that suffer from insufficientupdates, and so result in performance problems. This job can be scheduled as often asnecessary even could be hourly.

----Used to show information for a particular statistics

DBCC SHOW\_STATISTICS ([Sales.SalesOrderDetail], PK\_SalesOrderDetail\_SalesOrderID\_SalesOrderDetailID);

---- Used to show information for all statistics in a database

SELECT sp.stats\_id,

name,

filter\_definition,

last\_updated,

rows,

rows\_sampled,

steps,

unfiltered\_rows,

modification\_counter

FROM sys.stats AS stat

CROSS APPLY sys.dm\_db\_stats\_properties(stat.object\_id, stat.stats\_id) AS sp

--WHERE stat.object\_id = OBJECT\_ID('HumanResources.Employee');

select count(\*) from sales.SalesOrderDetail where SalesOrderID between 43659 and 43692

----used to update all statistics in a database

exec sp\_updatestats

----used to update all statistics in a table

Update STATISTICS Sales.Salesorderdetail

----used to update a particular statistics

Update STATISTICS [Sales].[Salesorderdetail] [missing\_idx\_Linetotal\_ProductID]

1. **Non-SARGable predicates**

SARGable, where SARG stands for Search Argument; it simply means that that a predicate can be usedin an index seek operation. The rules for SARGable predicates, in general, are that thecolumn should be directly compared (equality or inequality) to an expression, and thatany functions specified on the column will make the predicate non-SARGable. In otherwords, WHERE SomeFunction(Column) = @Value is not SARGable, whereas WHEREColumn = SomeOtherFunction(@Value) is SARGable. Note that SARGability doesn’t rule out the use of operators such as LIKE or BETWEEN (both inequality comparisons) orIN (treated as a set of equality comparisons).

Non-SARGable predicates can result in table or index scans and, similar to the case of missing indexes, this will cause significant CPU usage as SQL has to read and process far more rows than necessary. The query below shows an example of a WHERE clause predicate that is non-SARGable due to the usage of some date-manipulation functions on the ModifiedDate column. This example assumes that an index has been added on ModifiedDate, as there is not one in the standard AdventureWorksdatabase.

SELECTsoh.SalesOrderID,

OrderDate,

DueDate,

ShipDate,

Status,

SubTotal,

TaxAmt,

Freight ,

TotalDue

FROMSales.SalesOrderheaderASsoh

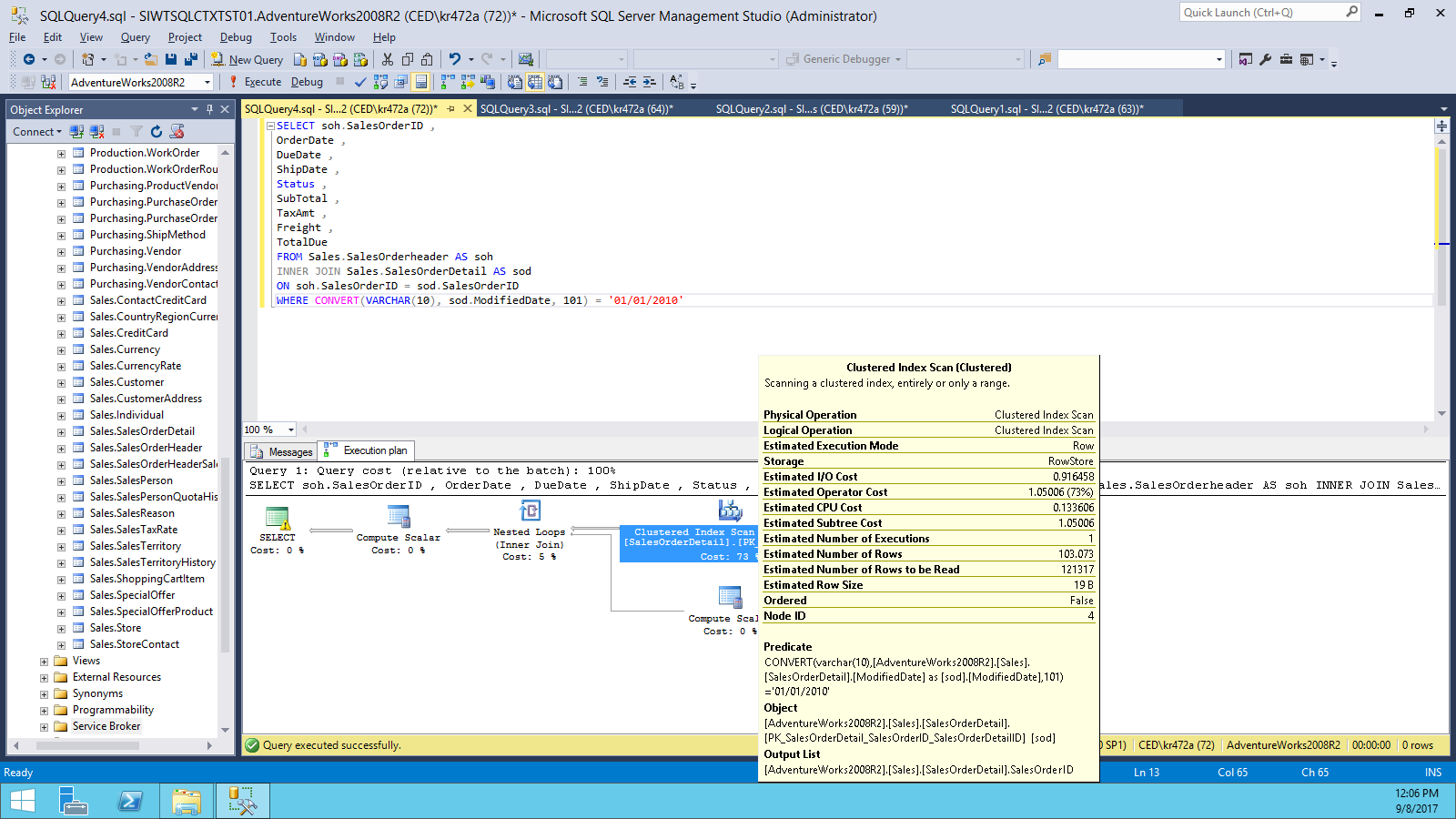
INNERJOINSales.SalesOrderDetailAS sod

ONsoh.SalesOrderID=sod.SalesOrderID

WHERE**CONVERT(VARCHAR(10),sod.ModifiedDate, 101)='01/01/2010'**

**Query: A non-SARGable predicate in the search condition**

From the execution plan, we can see that use of the functions on the ModifiedDatecolumn meant that an index seek operation was not possible; the entire index wasscanned in order to locate the matching values, as shown in figure below.



Now modify the predicate to be a range (inequality)search for date and times within the desired day.

SELECTsoh.SalesOrderID,

OrderDate,

DueDate,

ShipDate,

Status,

SubTotal,

TaxAmt,

Freight ,

TotalDue

FROMSales.SalesOrderheaderASsoh

INNERJOINSales.SalesOrderDetailAS sod ONsoh.SalesOrderID= sod.

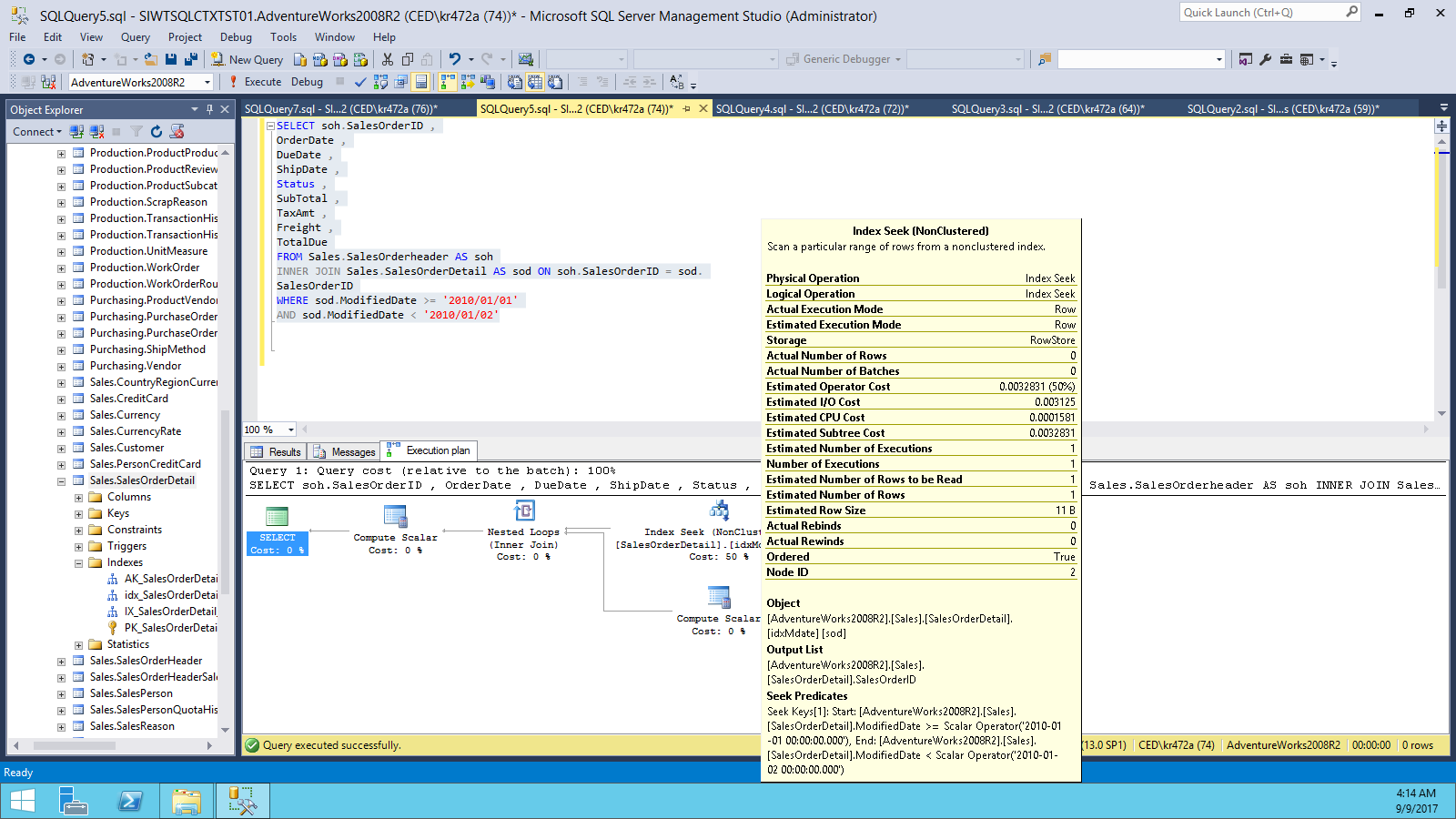
SalesOrderID

WHEREsod.ModifiedDate>='2010/01/01'

ANDsod.ModifiedDate<'2010/01/02'

**Query: A SARGable predicate in the search condition**

We now see an index seek operation (as shown below).



This is a fairly common problem in many databases; I often see functions such asUPPER, LTRIM, ISNULL being used in queries, either in the joins or in the WHEREclause, and in many cases there is simply no need for them. If the columns use a caseinsensitivecollation, then uppercase and lowercase values are considered equal, andthe use of the UPPER or LOWER functions do nothing other than degrade performance.Similarly, with string comparisons SQL ignores trailing spaces, removing the need forthe RTRIM function.

1. **Implicit conversions**

An implicit conversion results from a comparison of unequal data types. SQL cannot compare values of differing types and it must convert one of the columns involved to the same data type as the other, in order to do the comparison.When an implicit conversion occurs on a column that is used in a WHERE or FROM clause, the SQL Server Optimizer dictates a conversion of all the column values before the filter can be applied. This means that, during the query execution, the query processor will convert the lower precedence data type to the higher precedence data type before applying the filter or join condition. This means that, as with the case of functions on the column, ***the predicate is considered non-SARGable and so index seeks cannot be used***,SQL must process more rows than necessary to get the results, **and this leads to higherCPU usage**.

An example of this problem is the comparison of NVARCHAR parameters to columns that are of type VARCHAR. There are some data access libraries (JDBC) that pass string constants as Unicode (NVARCHAR), by default.**The problem is demonstrated in query below, where the AccountNumbercolumn is a VARCHAR and the parameter is a Unicode string (NVARCHAR), so designated by the N before the opening quote.**

SELECTp.FirstName,

p.LastName,

c.AccountNumber

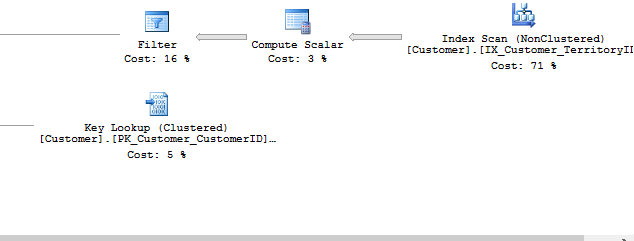
FROMSales.CustomerAS c

INNERJOINPerson.PersonAS p ONc.PersonID=p.BusinessEntityID

WHEREAccountNumber=N'AW00029594'

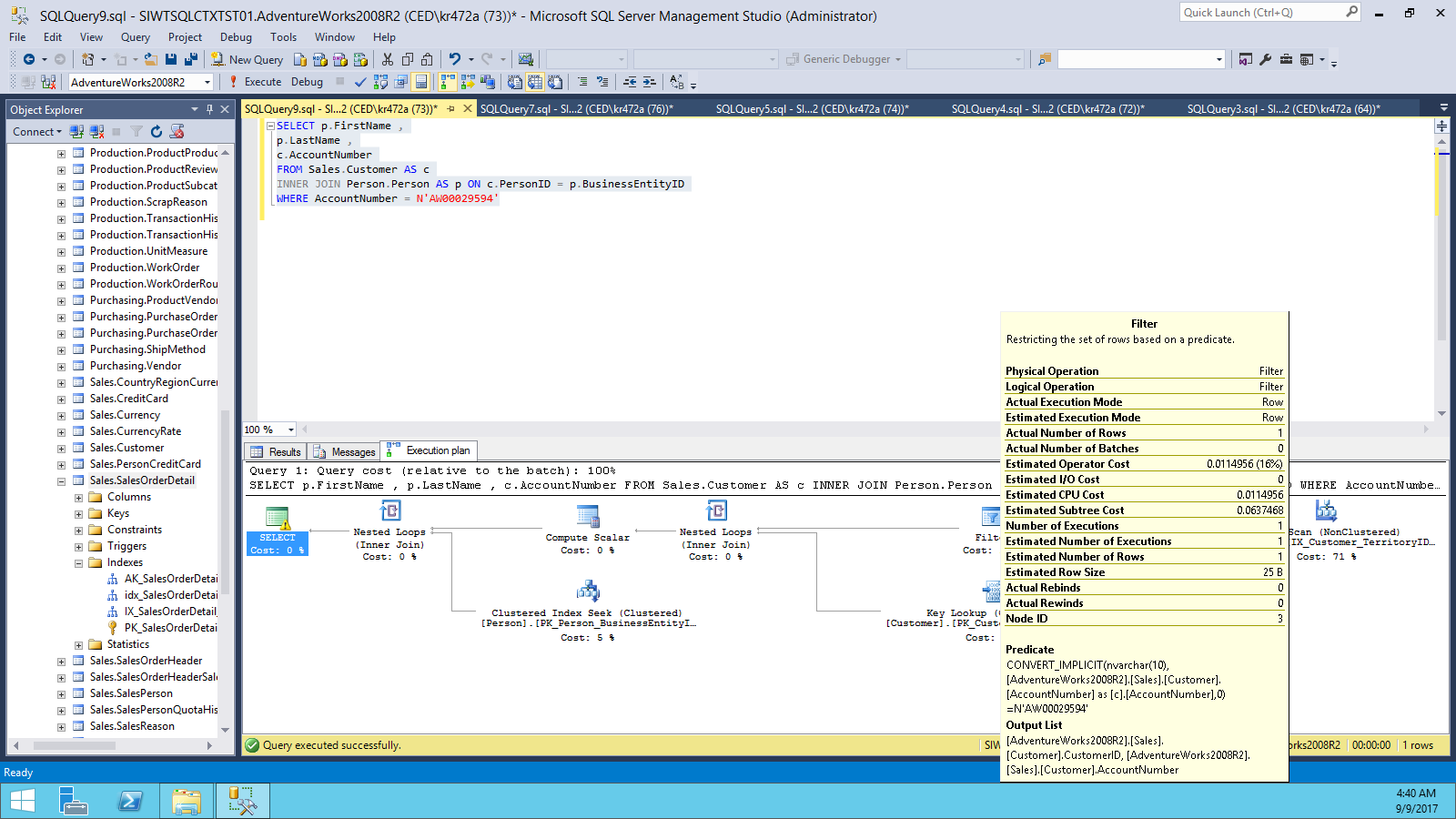
Query:An implicit data type conversion in the search condition.

The relevant section of the execution plan, shown below confirms that we get an**index scan** operation.



**Figure:** The non-SARGable predicate results in an index scan.

The **Filter** properties window, in figure below shows the implicit conversion.



**Figure: The filter predicate dictates the need to convert all the rows in the AccountNumber column to NVARCHAR**

**Solution**: The fix for implicit conversions is to ensure that columns used in joins are always of the same type and that, in the WHERE clause, any variables, parameters or constants are of the same type as the columns to which they are being compared. If they are not, make careful use of conversion functions (CAST, CONVERT) on the variables, parameters or constants sothat they match the data type of the column.

If using data access libraries like JDBC, check the properties to ensure that they are notpassing all string values as NVARCHAR regardless of the underlying column data type.

1. **Parameter Sniffing**

Parameter sniffing is a process used by SQL Server when creating an execution plan for astored procedure, function, or parameterized query. The first time the plan is compiled,SQL Server will examine, or "sniff", the input parameter values supplied, and use them,in conjunction with the column statistics, to estimate the number of rows that willbe touched by the query. It then uses that estimate in its costing of various possibleexecution plans. A problem only arises if the values that were passed as input parameterson initial plan creation, result in a row count that is atypical of that which will result fromfuture executions of the procedure. Parameter sniffing only occurs at the time a plan iscompiled or recompiled, and all subsequent executions of the stored procedure, function,or parameterized query will use the same plan.

**What is Parameter Sniffing?**

When an SP is not in the procedure cache, when it is executed the query optimizer needs to compile the SP to create an execution plan. In order to do this the query optimizer needs to look at the parameters that are passed and the body of the SP to determine the best method to go about processing the SP. When the query optimizer looks at the SP’s parameters, to help determine how to optimize the execution of the SP, it is known as parameter sniffing.

Once the query optimizer has sniffed the parameters and determined the best approach for processing the SP, it caches the execution plan in the procedure cache. All subsequent executions of the SP re-use the execution plan from the procedure cache regardless if different parameters are passed.

*The potential problem with this approach is the parameters that were used when the plan was cached might not produce an optimal plan for all execution of the SP, especially those that have significantly different set of records returned depending on the parameters passed.*

For instance, if you passed parameters that required a large number of records to be read, the plan might decide a table or index scan would be the most efficient method to process the SP. Then if the same SP was called with a different set of parameters that would only return a specific record, it would use the cached execution plan and perform a table or index scan operation to resolve it’s query, even if an index seek operation would be more efficient in returning the results for the second execution of the SP.

***If you have an SP that sometimes processes quickly, and other times processes slowly with different sets of parameters, then possibly parameter sniffing is causing your procedures to have varying execution times.***

**Problem Caused by Parameter Sniffing**

Before we look into how to write your SPs to control the issues related to parameter sniffing, let’s look at the issues caused by it first. To do that let’s review the following code:

USE AdventureWorks

GO

CREATE PROC GetCustOrders (@FirstCustint, @LastCustint)

AS

SELECT \* FROM Sales.SalesOrderHeader

WHERE CustomerID between @FirstCust and @LastCust;

This code accepts two parameters. These two parameters are used in a WHERE clause to determine which *SalesOrderHeader* records to return. Now, let’s look at the execution plan for this SP when we call it with two different sets of parameters. First, let’s call it with the following code:

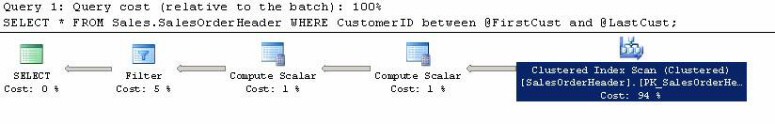
USE AdventureWorks

GO

DBCC FREEPROCCACHE

EXEC GetCustOrders 1,1000

When I execute the above code, I get the following execution plan:



Here, you can see that this execution of the GetCustomerOrder SP performed a Clustered Index Scan operation when I called this procedure with a range of CustomerID’s from 1 to 1000. Note, I freed the procedure cache using the “DBCC FREEPROCCACHE” statement to make sure that the query optimizer compiled the GetCustOrders SP based on the parameters I passed.

Now, I will execute the same SP with a smaller range of CustomerID’s and see what kind of execution plan I get. Here is the code I will be executing:

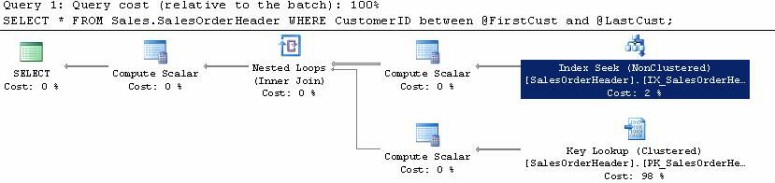
USE AdventureWorks

GO

DBCC FREEPROCCACHE

EXEC GetCustOrders 600,610

When I execute this code, I get the following execution plan:



This time I get a different execution plan. Now, you can see that when using a smaller range of CustomerID’s (600-610), I get an Index Seek operation. Depending on which execution of this SP was the first to be execute it, would compile and cache the execution plan for all subsequent executions of this SP. To verify this let’s run the following code and review the execution plans:

USE AdventureWorks

GO

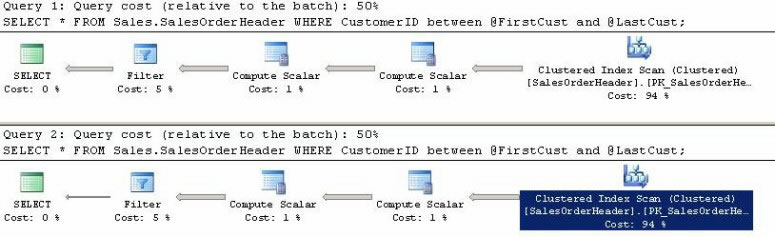
DBCC FREEPROCCACHE

EXEC GetCustOrders 1,1000

GO

EXEC GetCustOrders 600,610

Here is the execution plan for the above T-SQL batch:



By looking at this code, you can see that the second execution of the GetCustOrder SP now performs a Clustered Index Scan operation to find all the customer orders from a small range of CustomerID’s. This occurs because the query optimizer only sniffed the parameters for the first execution of the GetCustOrder, which had a large range, and then cached that execution plan. The cached execution plan was then used for the second execution of the GetCustOrder SP.

*The first compile of an SP creates the execution plan based on the parameters passed, and then this plan is stored in the procedure cache for the current and all future executions of the same SP. The last code segment above demonstrated how this could cause our second execution of GetCustOrder to use a less efficient execution plan--in this case, a Clustered Index Scan operation instead of an Index Seek operation. Let’s look at ways to overcome this problem.*

**Eliminating the Parameter Sniffing Problem by Disabling Parameter Sniffing**

In the prior example, I demonstrated how the first execution of an SP sniffed the parameters and then built an execution plan that was optimized for that set of parameters. This can lead to poor execution plans for any call that the SP uses that have a different set of parameters that might lead to a different execution plan. In my demonstrations above I showed that when a large range of CustomerID’s (1-1000) was sent to the GetCustOrder SP it used an index scan operation, but if a small range (600-610) was sent the optimal plan was an index seek operation. However, because SQL Server tries to minimize compiles of the SP, my small range of customerID’s used an index scan operation. If you have a wide variation in parameters that might be passed to an SP, you can eliminate the parameter sniffing problems by disabling parameter sniffing. You do this by coding your SP a specific way. Below I have rewritten my SP to eliminate parameters sniffing:

CREATE PROC GetCustOrders (@FirstCustint, @LastCustint)

AS

DECLARE @FC int

DECLARE @LC int

SET @FC = @FirstCust

SET @LC = @LastCust

SELECT \* FROM Sales.SalesOrderHeader

WHERE CustomerID BETWEEN @FC AND @LC

Here, can see I have declared two local variables (@FC and @LC) and then populated them with the values of the parameters passed to my SP. By doing this, the actual values of the parameters are no longer contained in the BETWEEN clause in the SELECT statement, instead only those local variables are present. Because of this small change, the query optimizer looks at all the statistics related to objects in my query and determines on average what might be the best execution plan to use based on the statistics.

Still, this method of eliminating the parameter sniffing problem doesn’t mean you will get an optimal plan for each execution of the SP. You still only get one execution plan stored in the procedure cache, which will be used for all executions of the SP. Although that one execution plan on average will perform optimal if you call the SP many times with many different parameter values. If you really want to create different execution plans based on the parameters passed then you will need to use a different approach.

**Resolving the Parameter Sniffing Problem Using a Decision Tree SP**

If you have an SP that is called with a number of different parameter values, where depending on the parameters passed they get different execution plans, you can solve the parameter sniffing problem by creating a decision tree SP. The decision tree approach has a single SP that is called, which decides which SP to call based on the parameters passed. This allows more than one SP to support your varying parameter values and allows for a more optimize plan to be used based on the parameters passed. Let me show you an example of a decision tree approach to resolving the parameter sniffing problem.

Let’s use the same situation as the above examples, where I want to return SalesOrderHeader records based on a range of CustomerIDs. Instead of the GetCustOrders SP selecting the order header records, it will instead determine the difference in range and then call a different SP based on whether the range of CustomerID’s is small or large. Let’s look at the code for the following three SPs:

CREATE PROC GetCustOrders (@FirstCustint, @LastCustint)

AS

IF @LastCust - @FirstCust< 100

EXEC GetCustOrdersNarrow @FirstCust, @LastCust

ELSE

EXEC GetCustOrdersWide @FirstCust, @LastCust

GO

-- Proc for Large Range of Customers

CREATE PROC GetCustOrdersWide (@FirstCustint, @LastCustint)

AS

SELECT \* FROM Sales.SalesOrderHeader

WHERE CustomerID BETWEEN @FirstCust AND @LastCust

GO

-- Proc for Small Range of Customers

CREATE PROC GetCustOrdersNarrow (@FirstCustint, @LastCustint)

AS

SELECT \* FROM Sales.SalesOrderHeader

WHERE CustomerID BETWEEN @FirstCust AND @LastCust

GO

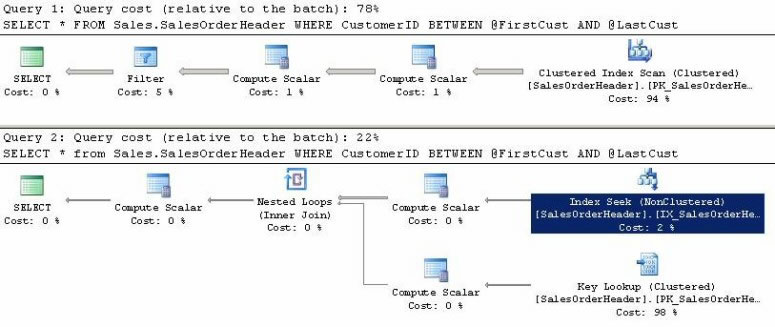
Here, you can see I have created two different SPs. The first SP (GetCustOrders) is my decision tree SP. This SP determines if the range of CustomerIDs is less than 100. If it is, it calls GetCustOrdersNarrowotherwise it calls GetCustOrdersWide. These two SPs are exactly the same. I’ll execute the following code to determine what execution plans are generated based on the parameters passed using this code:

DBCC FREEPROCCACHE

EXEC GetCustOrders 1,1000

EXEC GetCustOrders 600,610

Here are the two different execution plans produced by the two different EXEC statements:



As you can see, the first execution that had a CustomerID range greater than 100 used a Clustered Index Scan operation to resolve its execution. Whereas the second execution that had a CustomerID range of less than 100 used an Index Seek operation followed by a Key Lookup to resolve this smaller range query. So by building the decision tree approach, each execution with different parameters was resolved with a more efficient execution plan.

**Controlling Parameter Sniffing**

1. Disable Parameter Sniffing at instance level by enabling Trace Flag 4136

b)Disable Parameter Sniffing at query level by using TRACEQUERYON Trace Flag 4136.

CREATE PROCEDURE user\_GetCustomerShipDates

(

@ShipDateStartDATETIME ,

@ShipDateEndDATETIME

)

AS

SELECT CustomerID,

SalesOrderNumber

FROM Sales.SalesOrderHeader

WHERE ShipDateBETWEEN @ShipDateStartAND @ShipDateEnd

**OPTION (QUERYTRACEON 4136)**

GO

1. Specify **OPTION OPTIMIZE FOR** in stored procedure.

This allows the optimizer to optimize the plan for a parameter value that is known tobe more typically used. This will remove the possibility for parameter sniffing, but at thesame time may still lead to a plan that is not as efficient, if atypical values are used.

CREATE PROCEDURE user\_GetCustomerShipDates

(

@ShipDateStartDATETIME ,

@ShipDateEndDATETIME

)

AS

SELECT CustomerID,

SalesOrderNumber

FROM Sales.SalesOrderHeader

WHERE ShipDateBETWEEN @ShipDateStartAND @ShipDateEnd

**OPTION ( OPTIMIZEFOR ( @ShipDateStart= '2001/07/08',**

**@ShipDateEnd= '2004/01/01' ) )**

In SQL Server 2008, this was "extended" to provide the **OPTIMIZE FOR UNKNOWN**hint, which instructs SQL Server to not use parameter sniffing at all. This allows for a queryby-query control of parameter sniffing.

1. We can use the WITH RECOMPILE option, when creating a stored procedure, as another possible solution to parameter sniffing issues. This will mean a plan is never cached for the procedures, since it forces a recompile, and generation of a new plan on every execution. This means that row estimations will always be based on the current parameter value, but at the cost of increasing the execution time of the procedure.

CREATE PROCEDURE user\_GetCustomerShipDates

(

@ShipDateStartDATETIME ,

@ShipDateEndDATETIME

)

AS

SELECT CustomerID,

SalesOrderNumber

FROM Sales.SalesOrderHeader

WHERE ShipDateBETWEEN @ShipDateStartAND @ShipDateEnd

**OPTION ( RECOMPILE)**

GO

**Ad hoc non-parameterized queries**

Ad hoc queries are statements sent to the optimizer that are not predefined by using **stored procedures**, **sp\_executesql** or other ways to force reuse of execution plans. SQL Server always checks on the plan cache to see if a suitable plan can be reused for a given query, before going through the full process of generating a new execution plan and storing it in cache.

Ad hoc queries cause execution plans to be generated for each and every statement.This causes excessive use of resources, especially CPU. Consider the three queries:

SELECT soh.SalesOrderNumber,

sod.ProductID

FROM Sales.SalesOrderHeaderAS soh

INNER JOIN Sales.SalesOrderDetailAS sod

ON soh.SalesOrderID= sod.SalesOrderID

WHERE soh.SalesOrderNumber= 'SO43662'

SELECT soh.SalesOrderNumber,

sod.ProductID

FROM Sales.SalesOrderHeaderAS soh

INNER JOIN Sales.SalesOrderDetailAS sod

ON soh.SalesOrderID= sod.SalesOrderID

WHERE soh.SalesOrderNumber= 'SO58928'

SELECT soh.SalesOrderNumber,

sod.ProductID

FROM Sales.SalesOrderHeaderAS soh

INNER JOIN Sales.SalesOrderDetailAS sod

ON soh.SalesOrderID= sod.SalesOrderID

WHERE soh.SalesOrderNumber= 'SO70907'

**Queries: Three simple but non-parameterized queries**

These three statements should produce the same execution plan, but they don't. The different values hard-coded into value assignment in the WHERE clause mean that they are considered by the optimizer to be three different queries, and hence get separate execution plans.

For very simple queries, SQL Server can use a technique called **simple parameterization** to replace the fixed values with parameters, and so allow for plan reuse. However, even the queries in Listing above are too complex to qualify for simple parameterization.

The problem with non-parameterized queries is two-fold:

1. **The plan cache fills up with lots of single-use plans from ad hoc queries**. This means that the memory is used less efficiently. It also means that plans that might have been reusable can get discarded from cache due to memory pressure, require them to be compiled again when the queries are rerun.
2. **The compilation of these single-use plans wastes CPU**. Compilation is expensive, using relatively large amounts of CPU, and the repeated compilation of plans for adhoc queries, which are unlikely to be reused, is just a waste of resources. Cases where a lack of parameterization is causing excessive plan compilation, or where simple (or forced) parameterization is attempted but fails, can be identified using thefollowing counters from the SQL Statistics objects in Performance Monitor:

• **SQLServer: SQL Statistics: SQL Compilations/Sec**

• **SQLServer: SQL Statistics: Auto-Param Attempts/Sec**

• **SQLServer: SQL Statistics: Failed Auto-Param/Sec**

If the non-parameterized ad hoc queries are causing a problem, there are a couple of options for fixing it. The first and best option is to fix the problem at source, in the calling application. If that is not an option, there are settings in SQL Server that can be changed to alleviate the problem.

If it is possible to change the application that is sending these non-parameterized queries to SQL Server, then that option should be the one chosen. This can involve moving dataaccess from ad hoc queries embedded in the front-end code into stored procedures, or it may just involve changing the ad hoc queries that are embedded in the front-end code to their parameterized versions.

**Here is an example of *unparameterized* query being sent to SQL Server.**

cmd.CommandType = CommandType.Text;

cmd.CommandText = @"SELECT soh.SalesOrderNumber,

sod.ProductID

FROM Sales.SalesOrderHeader AS soh

INNER JOIN Sales.SalesOrderDetail AS sod

ON soh.SalesOrderID = sod.SalesOrderID

WHERE soh.SalesOrderNumber = '" + txtSalesOrderNo.Text + "'";

dtrSalesOrders = cmd.ExecuteReader();

**Query:** A non-parameterized query being sent to SQL Server.

Here is the above query but now in a parameterized form that will allow plan reuse.

dtrSalesOrders.Close();

cmd.CommandType = CommandType.Text;

cmd.CommandText = @"SELECT soh.SalesOrderNumber,

sod.ProductID

FROM Sales.SalesOrderHeader AS soh

INNER JOIN Sales.SalesOrderDetail AS sod  
ON soh.SalesOrderID = sod.SalesOrderID

WHERE soh.SalesOrderNumber = @SalesOrderNo";

cmd.Parameters.Add("@SalesOrderNo", SqlDbType.NVarChar, 50);

cmd.Parameters["@SalesOrderNo"].Value = txtSalesOrderNo.Text;

dtrSalesOrders = cmd.ExecuteReader();

**Query:** A parameterized query being sent to SQL Server.

If changing the application is not possible, as is often the case with vendor applications, or where the source code for the application is unavailable, then there are two options in SQL Server that can help alleviate the problem: **forced parameterization** and **optimize for ad hoc workloads**.

**Forced parameterization in SQL Server**

Database-level PARAMETERIZATION option can be set to FORCED, using the ALTER DATABASE statement, as below. This will force all ad hoc queries against that database to be parameterized before the compile process starts.

ALTER DATABASE AdventureWorks SET PARAMETERIZATION FORCED

**Query: Setting the PARAMETERIZATION option to FORCED.**

If you run this command and then rerun the three queries that we used earlier, the query that the SQL Query Optimizer gets to optimize will be the parameterized version, andyou'll find that there's only one plan created in cache, not three.

There are potential downsides to using forced parameterization, in that this setting forces SQL to use one plan for all matching queries, no matter what the literal values, so there is a possibility for the same parameter sniffing problems to which stored procedures are susceptible. If such a problem does occur, it can be investigated and resolved in much the same way as with stored procedures, discussed earlier.

**Optimize for ad hoc workloads**

We can use optimize for ad hoc workloads, which is a server-level setting, i.e. it will affect every single database on the server (as opposed to

Forced parameterization, which is database-specific).

With this setting enabled, SQL Server does not cache the plan of an ad hoc query the firsttime it is encountered. Instead, a plan-stub is cached that just indicates that the query has been seen before. Only if the query is seen a second time is the plan cached. This won't reduce the number of compiles for ad hoc queries, but it will make it less likely that the plan cache will grow as much, since the initial stub takes up very little memory. As such,it reduces the chances that other plans which could be reusable will be discarded due to memory pressure.

To enable the optimize for ad hoc workloads setting, use sp\_configure, as shown below.

EXEC sp\_configure'show advanced options',1

RECONFIGURE

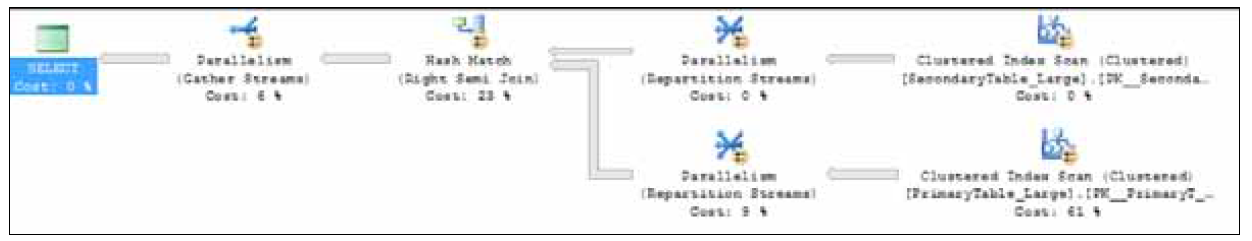
EXEC sp\_configure 'optimize for ad hoc workloads',1

RECONFIGURE

**Query: Enabling the optimize for ad hoc workloads setting.**

1. **Inappropriate parallelism**

SQL Server is designed to be able to make use of multiple processors when processinguser requests. Query parallelism is the mechanism used by the SQL query executionengine to split the work of a query into multiple threads, each of which will execute ona separate scheduler. Queries are parallelized at the operator level; in other words, if thequery runs in parallel, some of the query operators may run in their parallel form, whileothers may not.



**Figure:** Execution plan showing that operators are running in parallel.

When a query is submitted to SQL Server for processing, the query optimizer compiles an execution plan that has been optimized to allow the query to execute in the fastest manner possible. If the estimated cost of executing the plan serially exceeds the 'cost threshold for parallelism' sp\_configure option, the number of logical CPUs available to SQL Server is greater than one, and the 'max degree of parallelism'sp\_configure option is set to the default of zero or greater than one, the plan produced will include parallelism. The Degree of Parallelism (DOP) is not included as a part of the plan; this is, instead, determined at the time of execution based on the number of logical processors, the 'max degree of parallelism' sp\_configure or, if the MAXDOP query hint is being used, the value specified by the hint, and the number of available worker threads.

Parallel query processing can reduce the time required to process a query by horizontally partitioning the input data, distributing the partitions across multiple logical CPUs,and simultaneously executing the same operation across multiple processor cores. This can be very beneficial to data warehouse and reporting operations, which have a few large queries that deal with volumes of data and only a few requests occur concurrently.

By splitting the request across multiple OS threads on multiple processor cores, the optimizer increases the utilization of the hardware resources by spreading the load across all of the processors on the server, resulting in a reduction of total execution time.

The specific impact of a parallel workload depends on a number of factors, not the least of which is the ability of the remaining hardware components in the system to cope with the heavy demands for memory allocation and disk I/O that a parallel workload can generate. When parallelism is used appropriately, on high-cost queries, it can have avery beneficial effect on overall server performance. However, it can be very detrimental to OLTP environments where the workload consists of lots of smaller queries executing concurrently, since the parallel operation can utilize up to all of the processor cores on the server, causing other requests to wait to execute. If the primary use of the server is for an OLTP database that has a lot of smaller concurrent requests, parallelism of a single common query can sink throughput.

SQL Server has two configuration options that control the parallelexecution of queries by the engine. They are the cost threshold for parallelismand the max degree of parallelism options of sp\_configure. The max degree of parallelism option exists to prevent a single query from utilizing all of the processorcores on a SQL Server. The cost threshold for parallelism option exists to control the threshold for a query that causes the optimizer to use parallelism to execute a query.

Too often, when CPU issues related to "inappropriate parallelism" arise, the suggested solution seems to focus solely on changing the value for max degree of parallelism.For example, a quick online search of the problem, especially the CXPACKET wait type(a classic indicator of parallelism-related issues) will result in numerous posts that recommend reducing the max degree of parallelism to one half for one fourth the number of logical processors or processor cores on the server, or even to completely disable parallelism by setting it to 1. While this may solve the problem, it may not be the ideal solution. The best solution is to consider, in tandem, the appropriate value for each of these options.

**Cost threshold for parallelism**

The cost threshold for parallelism option determines a threshold

"cost" which, when exceeded, will cause a parallel execution plan to be generated, in orderto execute the user request. Since parallel execution is only possible on multi-processorsystems, the cost threshold for parallelism option is only used by the enginewhen multiple processors exist, the server is not affinitized to a single processor and maxdegree of parallelism is set to a value other than 1.

The "cost" is the estimated amount of time in seconds that it would take to executethe query serially with a given execution plan. The default value is five, meaning thata parallel plan will only be generated and used by queries that are estimated to takelonger than five seconds to execute serially on the given system. On larger databases,this threshold may be low enough to cause multiple concurrent executions of commonqueries, and so to cause contention in the system.

To determine what might be an appropriate setting for the cost threshold for parallelism option, it is possible to query the existing plans in the plan cache to determine the costs associated with the plans that have been executing with parallelism.

WITH XMLNAMESPACES

(DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/showplan')

SELECT query\_plan AS CompleteQueryPlan,

n.value('(@StatementText)[1]', 'VARCHAR(4000)') AS StatementText,

n.value('(@StatementOptmLevel)[1]', 'VARCHAR(25)')

AS StatementOptimizationLevel,

n.value('(@StatementSubTreeCost)[1]', 'VARCHAR(128)')

AS StatementSubTreeCost,

n.query('.') AS ParallelSubTreeXML,

ecp.usecounts,

ecp.size\_in\_bytes

FROM sys.dm\_exec\_cached\_plans AS ecp

CROSS APPLY sys.dm\_exec\_query\_plan(plan\_handle) AS eqp

CROSS APPLY query\_plan.nodes

('/ShowPlanXML/BatchSequence/Batch/Statements/StmtSimple')

AS qn( n)

WHERE n.query('.').exist('//RelOp[@PhysicalOp="Parallelism"]') = 1

Query: Determining the estimated cost of parallel execution plans.

Analysis of the most commonly executed statements that result in parallel queries can guide the appropriate setting of the cost threshold for parallelism option to minimize the impact of multiple concurrently executing parallel requests which drive CPU and I/O contention in the system.

**Max degree of parallelism**

Whenever the estimated cost of executing a query serially exceeds our carefully-evaluatedvalue for cost threshold for parallelism, the database engine can spread theexecution load for that query across multiple available processors, according to the degreeof parallelism dictated by the max degree of parallelism option. The number of processors used will be determined by the lowest of the following three values:

• Number of available processors

• Max degree of parallelism option

• MAXDOP query hint provided for the query being executed (which overrides the value specified by max degree of parallelism).

The appropriate value for the max degree ofparallelism option depends largely on the type of workload being executed, and theability of the other hardware subsystems to cope with the additional workload associated with parallel execution in the system. If your system is experiencing parallelism-relatedissues (see the *Diagnosing inappropriate parallelism* section), then it may be necessary tolimit the degree of parallelism, in conjunction with tuning the cost threshold for parallelismto resolve the problem.

One of the more common online recommendations is to disable parallelism entirely bysetting the max degree of parallelism to 1. There are cases where this configurationmight make sense, for example, true OLTP workloads where all of the transactions aresmall and there are a lot of transactions executing concurrently. These types of databaserarely exist today and disabling parallelism entirely is more likely to reduce performancein the long term.

In a SMP system, setting MAXDOP to half the number of available physical processor cores, or to the number of physical cores on asingle processor die, or even setting it to 1 to disable parallelism entirely. But, recommendation should be based on analysis of the query workload, and a review of the wait types of associated workers and subtasks which are executing using parallelism. In particular, we should analyse occurrences of the CXPACKET session wait type. In most systems,CXPACKET is the symptom and not the problem; there is often a different underlying waittype that can be seen in sys.dm\_os\_waiting\_tasks for the session. By focusing on this wait type, a better decision regarding the appropriate max degree of parallelism option can be made. For example, if the underlying wait type, is PAGEIOLATCH\_SH then the parallel operation is waiting on a read from the disk I/O subsystem, and reducing the max degree of parallelism won't resolve the root problem; it will just reduce the number of workers being used in the system, and reduce the accumulated wait time for the CXPACKET wait type. However, this may reduce the additional load the parallelism operations place on the disk I/O subsystem, and buy you time to scale up the I/O performance of the server.

There are, however, some specific considerations, relating to the memory architectureof the processors. On NUMA-based (Non-Uniform Memory Access) systems, the max degree of parallelism option should be set to the minimum number of processors available on a single NUMA node. This is done to prevent cross-node parallel processing of a request from occurring, which incurs significant expense since sharing memory across nodes is an expensive operation.

**Hyper-threading and parallelism**

Hyper-threading is an Intel technology designed to improve parallel execution by presenting to the operating system two logical cores for each physical core. This means that instead of one scheduler per processor core you get two, and so two threads can be executed "simultaneously."

With recent processor architectures, especially Intel Nehalem and later, my adviceis to enable hyper-threading unless you find a good reason to turn it off. It's certainly a mistake to disable hyper-threading without first thoroughly testing your application workload with hyper-threading enabled, then disabled, in order to truly know whether or not there is a benefit to having hyper-threading turned on or off.

**Diagnosing inappropriate parallelism**

The best way to determine if parallel processing is causing a resource bottleneck in aspecific system is to look at the wait statistics and latch statistics for an instance of SQLServer. When a bottleneck exists during the parallel execution of a query, the CXPACKET wait type shows up as one of the top waits for SQL Server. This wait type is set whenever a parallel process has to wait in the exchange iterator for another worker to continue processing. As previously discussed, when this happens, there is generally an underlying,non-CXPACKET wait type, which is associated with the stalled worker. However, since multiple workers are forced to wait when this occurs, the volume of CXPACKET waits will generally exceed the underlying root wait type being exhibited.

Whenever possible, it is best to isolate and troubleshoot the underlying wait type, since this will lead to overall system throughput improvements. The CXPACKET waits are simply a symptom of a problem in most cases, not the actual problem. There are scenarios where it may not be possible to eliminate the underlying wait type; for example, whenthe disk I/O subsystem can't keep up with the demand required by the parallel executionof a query, the root wait type may be an IO\_COMPLETION, ASYNC\_IO\_COMPLETION, or PAGEIOLATCH\_\* wait type, and scaling out the I/O subsystem is not possible. When this occurs, reducing the level of parallelism to a degree that still allows parallel processing to occur without bottlenecking in the disk I/O subsystem can improve overall system performance.

It is possible that CXPACKET waits in conjunction with other wait types,for example LATCH\_\* and SOS\_SCHEDULER\_YIELD, do show that parallelism is the actual problem, and further investigation of the latch stats on the system will validate if this is actually the case. The sys.dm\_os\_latch\_stats DMV contains information about the specific latch waits that have occurred in the instance, and if one of the top latch waits is ACCESS\_METHODS\_DATASET\_PARENT, in conjunction with CXPACKET,

LATCH\_\*, and SOS\_SCHEDULER\_YIELD wait types as the top waits, the level of parallelism on the system is the cause of bottlenecking during query execution, and reducing the 'max degree of parallelism' sp\_configureoption may be required to resolve the problems.

EXEC sp\_configure 'show advanced options', 1;

GO

RECONFIGURE WITH OVERRIDE;

GO

EXEC sp\_configure 'max degree of parallelism', 16;

GO

RECONFIGURE WITH OVERRIDE;

GO

Starting with SQL Server 2016 (13.x), use the following guidelines when you configure the **max degree of parallelism** server configuration value:

|  |  |  |
| --- | --- | --- |
| Server with single NUMA node | Less than or equal to 8 logical processors | Keep MAXDOP at or below # of logical processors |
| Server with single NUMA node | Greater than 8 logical processors | Keep MAXDOP at 8 |
| Server with multiple NUMA nodes | Less than or equal to 16 logical processors per NUMA node | Keep MAXDOP at or below # of logical processors per NUMA node |
| Server with multiple NUMA nodes | Greater than 16 logical processors per NUMA node | Keep MAXDOP at half the number of logical processors per NUMA node with a MAX value of 16 |

**Windows Server and BIOS power saving options**

The first thing to do is to check the current Windows power management scheme. If it is set to **Balanced**, change it to **High Performance**, which prevents the system fromswitching to a low performance state and ensures the performance characteristics ofthe system are. If **High Performance** is being used by Windows, the BIOS setting should be checked. If it's set to **Hardware**, change it to **OS Control**.