# SQL_R2_Logo.jpg

Understanding and Controlling Parallel Query Processing in SQL Server

**Writers:** Don Pinto, Eric Hanson

**Technical Reviewers:** Kevin Cox, Thomas Kejser, Jay (In-Jerng) Choe

**Published:** October 2010

**Applies to:** SQL Server 2008 R2

**Summary:** Data warehousing and general reporting applications tend to be CPU intensive because they need to read and process a large number of rows. To facilitate quick data processing for queries that touch a large amount of data, Microsoft SQL Server exploits the power of multiple logical processors to provide parallel query processing operations such as parallel scans. Through extensive testing, we have learned that, for most large queries that are executed in a parallel fashion, SQL Server can deliver linear or nearly linear response time speedup as the number of logical processors increases. However, some queries in high parallelism scenarios perform suboptimally. There are also some parallelism issues that can occur in a multi-user parallel query workload. This white paper describes parallel performance problems you might encounter when you run such queries and workloads, and it explains why these issues occur. In addition, it presents how data warehouse developers and can detect these issues, and how they can work around them or mitigate them.

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Contents

[Introduction 4](#_Toc274998119)

[Parallel Query Processing in SQL Server 4](#_Toc274998120)

[Methods of Controlling Parallelism in SQL Server 4](#_Toc274998121)

[SQL Server Query Optimizer 5](#_Toc274998122)

[Plan Simplification 5](#_Toc274998123)

[Plan Enumeration 5](#_Toc274998124)

[Plan Evaluation 6](#_Toc274998125)

[SQL Server Query Execution Engine 6](#_Toc274998126)

[Dynamic Selection of Degree of Parallelism 6](#_Toc274998127)

[Dynamic Selection of Number of Worker Threads 8](#_Toc274998128)

[Query Execution Memory Requirements 10](#_Toc274998129)

[Benefits of Parallel Query Processing in SQL Server 12](#_Toc274998130)

[Problems 13](#_Toc274998131)

[(1) Insufficient Memory During Execution of Parallel Query Plans 13](#_Toc274998132)

[(2) Synchronization Overhead of Parallel Operators 13](#_Toc274998133)

[(3) Statistical Estimation Errors 14](#_Toc274998134)

[(4) Data Skew 14](#_Toc274998135)

[Problem Identification and Mitigation 15](#_Toc274998136)

[Degree of Parallelism and Memory Grants 15](#_Toc274998137)

[Worker Thread Balance 16](#_Toc274998138)

[Distribution of Data 16](#_Toc274998139)

[Individual Threads Statistics and CXPACKET Waits 17](#_Toc274998140)

[Indexes and Statistics 19](#_Toc274998141)

[Recommendations 19](#_Toc274998142)

[Parallel Data Warehouse 20](#_Toc274998143)

[Conclusions 20](#_Toc274998144)

[Bibliography 20](#_Toc274998145)

# Introduction

Parallel query processing is a crucial Microsoft SQL Server feature that helps speed up processing of data warehouse queries. Most queries in OLTP systems run serially, but some larger OLTP queries also benefit from parallelism. SQL Server exploits the power of modern multicore hardware platforms to shorten query response time for queries that access a large amount of data. A *logical processor* is either a CPU core, or in the case of hyper-threaded cores, one of multiple hardware threads that can run simultaneously on a core. In general, the parallel query processing capabilities of SQL Server provide linear or close to linear speedup as the number of logical processors working on a query increases. However, there are some cases where performance doesn’t benefit as much as anticipated from parallelism, and database or application tuning can help improve performance. This paper gives an overview of how parallelism works in SQL Server, and it shows how you can identify the most common parallel query processing performance problems and work around them. If you’re already familiar with parallel processing in SQL Server, you might want to skip the details explained in the next few sections and go directly to the recommendation section.

In this paper, we focus on CPU parallelism. We assume that your I/O system is configured properly for high bandwidth using the Fast Track data warehousing guidelines [5], so that the I/O system is not the primary performance limiter. To help you understand how CPU parallelism works, we discuss the roles of the SQL Server query optimizer and query execution engine in parallel query processing. To assist with troubleshooting, we describe the most common problems observed in multi-user parallel query processing environments. These include insufficient memory to execute complex query plans, lack of worker threads, synchronization overhead using parallel operators, inaccurate cardinality estimates, and data skew. We present some techniques you can use to detect these problems, and provide guidance about how to mitigate them. Finally, we also briefly discuss how the new SQL Server Parallel Data Warehouse product can reduce these issues significantly for large data warehouses.

# Parallel Query Processing in SQL Server

Parallel query processing aims at reducing response time by utilizing the processing power of multiple CPUs to process a query. Essentially, parallel query execution involves partitioning of a dataset into smaller approximately equal-sized sets, assigning each set to a group of worker threads that are executing simultaneously, and coordinating across worker threads to process these smaller datasets.

## Methods of Controlling Parallelism in SQL Server

SQL Server provides flexibility so that you can control parallelism by specifying the maximum degree of parallelism (MAXDOP) per query using explicit query hints, or on a per-server basis using the **sp\_configure** max degree of parallelism option. Additionally, by using SQL Server Resource Governor, you can set the value of MAXDOP so that different workload requests can be managed differently. The run-time degree of parallelism (DOP) of a query is the number of logical processors dedicated to that query. Due to the different ways of specifying the value of MAXDOP, deciding which value of MAXDOP to honor within SQL Server is done using the following rules: (i) If a query explicitly uses a Transact-SQL query hint with a MAXDOP value that is greater than 0 (such as OPTION (MAXDOP N)), this value always overrides the MAXDOP value configured through **sp\_configure**. (ii) If MAXDOP is specified using an explicit Transact-SQL query hint, this value overrides the Resource Governor workload group degree of parallelism limit value, if it is less. If the MAXDOP is set using the **sp\_configure** option without using the query hint, and if Resource Governor is enabled, the Resource Governor workload group degree of parallelism limit value is used instead unless the query is set to run in serial mode (MAXDOP=1). This is because queries compiled to execute serially can’t be parallelized, and a MAXDOP=1 hint forces compilation of a serial plan. For more information, see the Resource Governor, **sp\_configure** and explicit query hint topics in SQL Server Books Online in the [MSDN Library](http://msdn.microsoft.com/en-us/library/default.aspx) (http://msdn.microsoft.com/en-us/library/default.aspx).

The two essential components of parallel processing in SQL Server are the *query optimizer* and the *query execution engine*. This section discusses their roles in more detail.

## SQL Server Query Optimizer

The main role of the query optimizer is to produce an efficient query execution plan for an SQL query. Depending on the optimization goals, the query plan can be optimized for response time, throughput, I/O, memory, number of logical processors, or a combination of such goals. These multidimensional goals and the constraints within which the query optimizer has to function make query optimization a challenging task. For a query to be eligible to run in parallel, the query optimizer must explicitly decide to make create a parallel query plan for it, and it must then place parallel operators into the plan.

Query optimization in SQL Server is cost-based and is made up of three main steps: (i) the *plan simplification* step, (ii) the *plan enumeration* step, which generates a set of candidate plans, and (iii) the *plan evaluation* step, which evaluates candidate plans to decide which plan to use.

### Plan Simplification

In the plan simplification step, the SQL Server query optimizer applies rules, such as predicate pushdown and contradiction detection, to produce a trivial plan for some simple queries. For example: In the case of INSERT statements with the VALUES clause, a simple plan is produced.

If the trivial plan is not available, SQL Server loads any statistics that will help it in the cost-based plan enumeration and evaluation steps that follow.

### Plan Enumeration

In the plan enumeration step, the SQL Server query optimizer searches through the plan execution space by considering different execution orders and implementation algorithms for operations such as scans, joins, and group by and union clauses before coming up with a set of good candidate plans. Instead of doing an exhaustive plan search, the query optimizer reduces the optimization cost by using an acceptable upper bound on the effort for each stage in this step. During this effort, the query optimizer gathers all the plans it can from the solution space and produces a plan that is close enough to the optimum plan.

### Plan Evaluation

In the plan evaluation step, the SQL Server query optimizer evaluates candidate plans to come up with the best possible query execution plan. This is done in stages to reduce the complexity and the run-time cost. Candidate plan evaluation and ranking is governed by three factors: (i) cardinality estimation, (ii) cost estimation, and (iii) the amount of time the query optimizer can spend on this step. Cardinality estimation determines how many rows need to be processed by each operator at each step in the query execution plan. Cardinality estimation uses statistics to predict the number of rows that will be output by each operation. Cost estimation determines the CPU and I/O expense as well as the execution time for each operator in the query plan. Cost estimation uses a model of the various execution algorithms and plugs in the results of the cardinality estimation to predict the CPU, I/O, and elapsed time for each operation in the execution plan. During costing, the query optimizer also considers a number of hardware dependent-properties, such as the maximum available server memory on the machine and the number of schedulers available to the query, as well as MAXDOP to estimate how much benefit we can get from running a query in parallel versus serial, and how likely it is that we might spill to disk.

Because longer queries can more effectively amortize the cost associated with parallelism (that is, by initializing, synchronizing, and terminating parallel plans) than shorter queries can, SQL Server uses an **sp\_configure** option called cost threshold for parallelism to determine which queries are considered short. The query optimizer generates serial plans for such queries if their cost is below the threshold this option specifies. During each stage of plan evaluation, the query optimizer decides whether it will be useful to go to the next stage and spend more time further optimizing the plan. If the query optimizer decides not to go into the next stage, it returns the best plan found so far.

## SQL Server Query Execution Engine

The SQL Server query execution engine executes the plan generated by the SQL Server query optimizer. This section describes how resources are allocated to a parallel plan.

### Dynamic Selection of Degree of Parallelism

Figure 1 illustrates the different steps taken by the SQL Server query execution engine for executing a parallel query.

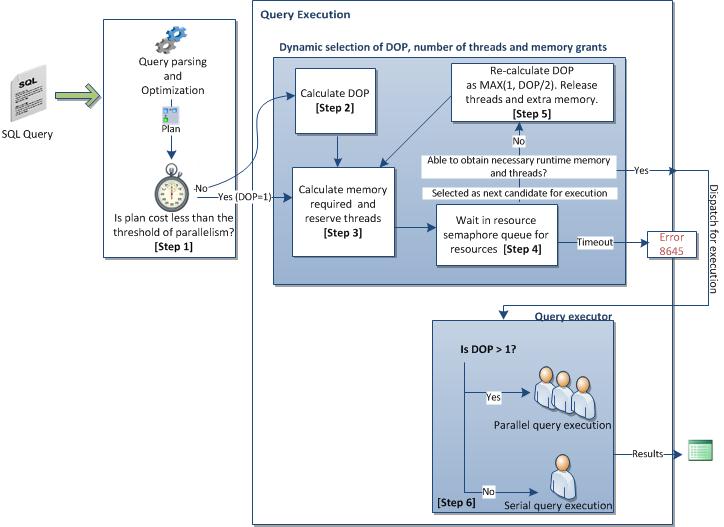


Figure 1: Parallel query processing

The default value of max degree of parallelism in SQL Server is 0, which is interpreted as follows: If MAXDOP = 0 is used for a query, either via the default or through explicit use of a MAXDOP query hint, SQL Server can potentially use all the processor cores to parallelize a query up to a maximum of 64. For example:

1. If MAXDOP is set to 0 on a 32 logical processor machine, the maximum degree of parallelism used is 32 unless another value is explicitly specified.
2. If MAXDOP is set to 0 on a 256 logical processor machine, the maximum degree of parallelism used is 64 unless another value is explicitly specified.

In Step 1, if the MAXDOP value is equal to 1 or if the cost of a query plan is lower than the cost threshold of parallelism, the query optimizer generates a serial plan and the query is executed serially. In the case of a parallel plan, if MAXDOP is set to 0 or to a value greater than 1, the SQL Server query execution engine uses the information from the query plan to calculate an expected degree of parallelism value (Step 2). This degree of parallelism calculation is dependent on the plan shape, the availability of system resources, and the query cost, while honoring the specified MAXDOP value if MAXDOP is greater than 1.

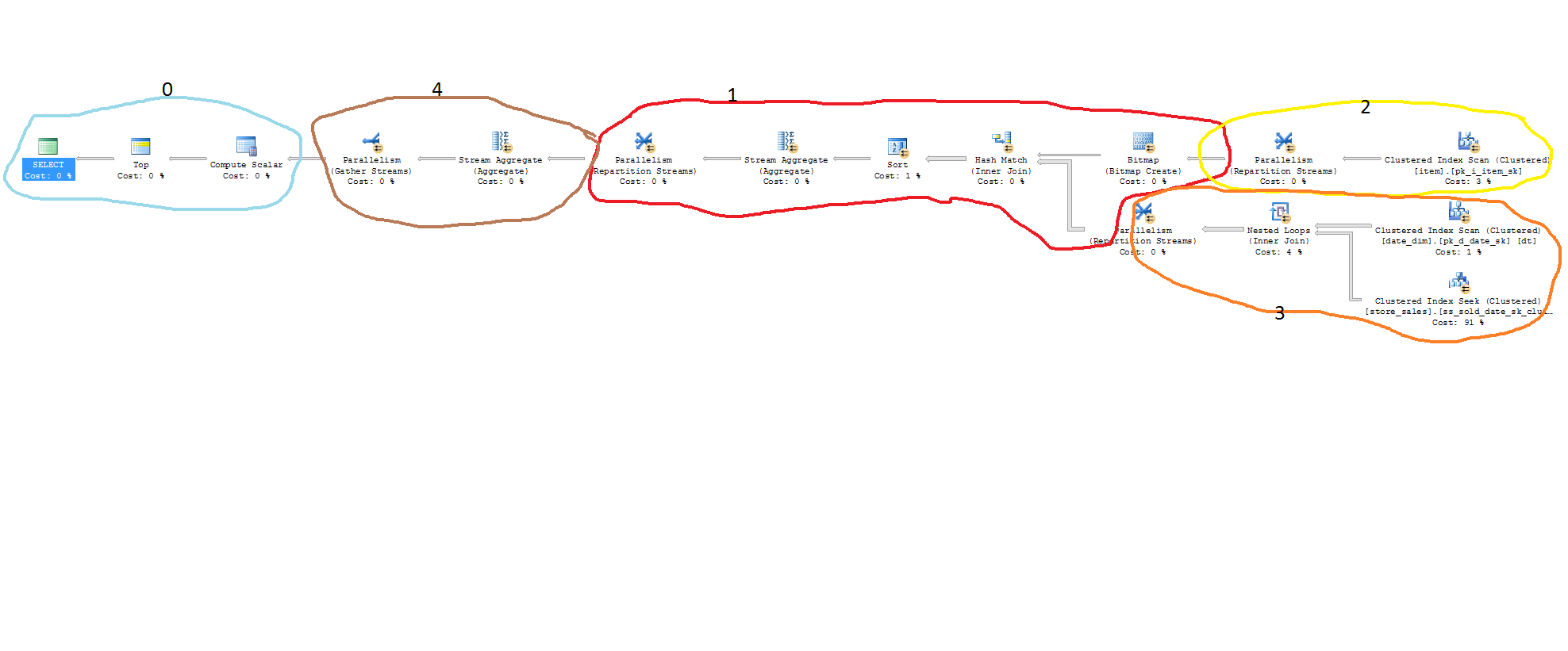
In Step 3, the query execution engine in SQL Server calculates the resources required for a query to execute such as memory and the number of threads. In Step 4, SQL Server uses an internal facility called the resource semaphore to reserve the threads and the memory resources that are required for execution. The resource semaphore ensures that queries are able to satisfy the respective resource reservation values while keeping the overall resource usage within the set limits. First, the memory reservation values for a query are checked to make sure the query needs memory. For example: Simple serial queries without ORDER BY or GROUP BY clauses may not need a memory grant, and in such cases the query is dispatched for execution immediately. Next, SQL Server examines the maximum memory limit for a particular query. This value can be configured through the Resource Governor workload group REQUEST\_MAX\_MEMORY\_GRANT\_PERCENT value. If the memory grant request can be met (that is, if the memory grant request value is less that the per-query maximum limit) and the required thread resources can be obtained, the query is executed immediately. After a query is dispatched for execution, the degree of parallelism cannot change.

If there are not enough resources readily available, the query is put into a resource semaphore queue to wait for resources to become available. The time-out value associated with a query waiting in the resource semaphore queue is proportional to the query cost. The resource semaphore uses an internal query cost estimate to prioritize memory grants and decide which query to run next when memory is available. While the query is waiting in the resource semaphore queue, if the query times out, an error is returned (8657 or 8645). If the query is picked as a candidate for execution from the resource semaphore queue but cannot obtain the required resources, SQL Server reduces the estimated degree of parallelism (DOP) by half (Step 5), and then it recomputes the memory and thread resources for the query (Step 3). SQL Server may ultimately downgrade the parallel plan to a serial plan. Downgrading to a serial plan causes removal of parallelism operators from the plan prior to execution.

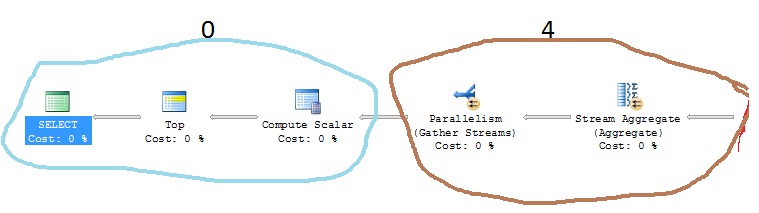
### Dynamic Selection of Number of Worker Threads

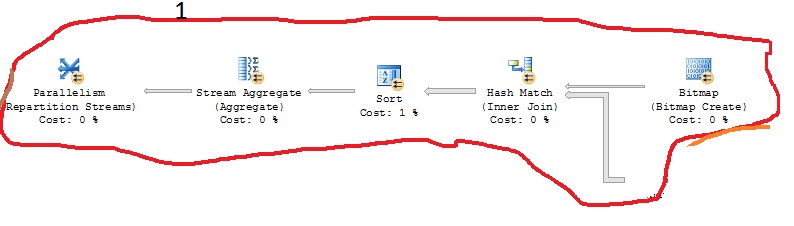
In SQL Server, the max worker threads **sp\_configure**configuration value determines the maximum number of worker threads created for handling all simultaneous SQL Server requests. The default value of max worker threads in SQL Server 2008 is 0. If the default value (0) is used for max worker threads, SQL Server automatically configures the number of worker threads at startup. In this case, the value depends on the machine architecture (32-bit vs. 64-bit) and on the number of logical processors.

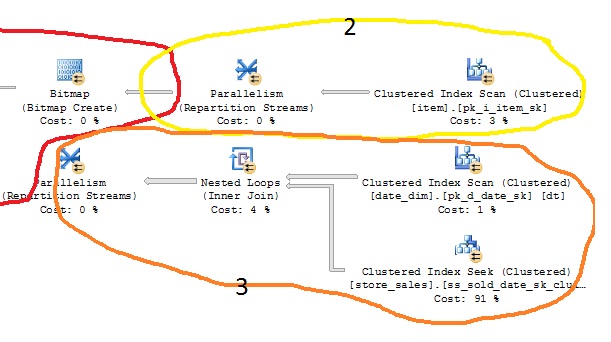
Depending on the degree of parallelism value and the different possibilities of data flow and execution order in a parallel query plan, the number of threads calculated by the query execution engine for running a query is estimated as:



Query Plan 1 : Parallel query plan with 2 concurrent branches (see below for zoomed-in view)







For example, consider a graphical query plan, Query Plan 1. Note: this would all appear on one line in the query plan, but it is being wrapped so that you can clearly read each step in this document.

Query Plan 1 has five branches, but at any point there are only two concurrent branches at work – branches (1) and (2) or branches (1) and (3). This is because by the time the hash-join operator finishes the hash-table build, branch (2) is no longer needed.

If a degree of parallelism value of 4 is used, the number of estimated threads for executing this query is 4 \* 2 = 8.

The query execution engine calculates the estimated number of worker threads required for executing a query before it actually executes the query. Because this is just an estimate of the number of threads, the actual thread consumption can be different depending on the system status. For example: The finished thread may not exit immediately when the system is under load. On the other hand, some plans may have some branches that are serially executed and in such cases the plans use fewer threads than estimated.

### Query Execution Memory Requirements

In addition to the memory requirements of the query compiler and plan cache, SQL Server requires memory for query execution. The memory grant for a SQL Server query is the total amount of memory required for a query to execute and is divided into two components: (i) *required memory* and (ii) a*dditional memory*. Required memory is the minimum amount of memory the query must have in order to execute. Additional memory is the memory that the query needs in order to fit additional data structures such as SORT tables completely in memory during query execution. Calculating the total amount of memory that might be required for a query is a complex but important task. It is necessary to be accurate because overestimating the amount of memory required can reduce the number of concurrently executing queries, and underestimating the amount of memory required can result in performance degradation due to lack of sufficient memory.

For example, consider a simple query that needs to sort 1 million rows, each 10 bytes in size. The required memory for this query is 512 KB because this is the minimum amount SQL Server needs to construct internal data structures to handle one sort. Because it would take 10 MB to store all rows, the additional memory would be 10 MB (slightly higher when overhead is included). This calculation becomes more complex if the compiled plan has multiple sorts and joins because SQL Server also considers the lifetime of each operator for more efficient memory usage. You would generally see a smaller estimate than the sum of all sorts and joins.

In the case of parallel query plan execution, the parallelism operator (also known as exchange) facilitates parallelism by connecting parallel source operators to destination operators. The optimizer places exchanges at boundaries between other operators and during query execution the exchange moves rows between threads for parallel processing. As illustrated in Figure 2, the exchange operator is internally composed of two iterators – producer and consumer. Buffers are used for flow control between consumers and producers, and for processing a batch of rows at a time rather than individual rows. Each producer reads rows from its subtree into its local buffer, assembles the rows into packets, and routes these packets to appropriate consumer buffers. In the case of a repartition stream exchange operator [7], the number of producer and consumer threads for each exchange operator is equal to the run-time degree of parallelism.

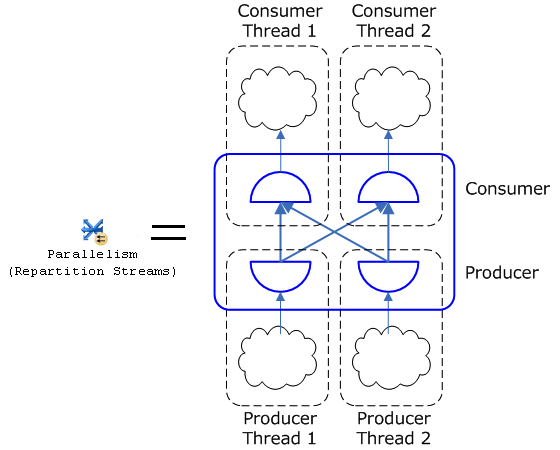


Figure 2: A repartition exchange operator running at degree of parallelism =2

For a given degree of parallelism, the number of buffers needed is proportional to . Each buffer requires a fixed amount of memory and thus the memory requirement of complex parallel plan can be large.

# Benefits of Parallel Query Processing in SQL Server

SQL Server harnesses the power of multicore processors to speed query execution time significantly. Parallel table scan is a common query plan operator in most large-scale data warehouse and general reporting application workloads. The following query finds the number of line-items whose shipping date is less than '1992-08-06'.

Select COUNT(\*) from LINEITEM where L\_SHIPDATE < '1992-08-06'

Query 1: Sample test query to illustrate benefit of parallel processing

As an example, we present a parallel scan test that was done on in-memory data. In this scaling experiment, we used two different parallel machines (Red and Blue) each with ample memory, and multiple Intel Hex Core Xeon X7460 processors, one at 2.66 GHz and the other at 2.13 GHz respectively. The I/O-subsystem configurations are not described because all of the tests were done on in-memory data.

Figure 3: Decrease in execution time as the number of cores increases

As shown in Figure 3, as we increased the number of cores from 1 to 12, the time taken for an in-memory parallel scan operation to finish dropped significantly. For the 1 to 8 case, the speedup is close to linear.

At higher degree of parallelism values, up to 32, adding additional logical processors to handle a single query typically does improve query response time. For some queries, benefits from parallelism can be seen up to a degree of parallelism of 64. Beyond a degree of parallelism of 64 there doesn’t tend to be significant improvement in query run time. Beyond a degree of parallelism of 16, each additional logical processor assigned to the query tends to make a smaller marginal contribution to overall query response time.

**Note:** This discussion applies to SQL Server 2008 R2.

For multi-user workloads, running multiple queries at a degree of parallelism equal to ½ or ¼ of the number of logical processors on the machine can give an excellent balance of throughput and response time, and fully utilize the resources on a 4-way or 8-way SMP. The SQL Server query execution team historically has improved parallel query execution with each release, so these performance characteristics can be expected to change with successive releases.

# Problems

Through extensive testing, we have found that there are a small fraction of cases at higher MAXDOP settings where parallelism no longer improves performance. In most such cases, the prominent root cause of performance degradation is tone of the following: (i) insufficient memory during execution of parallel query plans, (ii) synchronization overhead of parallel operators, (iii) statistical estimation errors, or (iv) data skew issues.

### Insufficient Memory During Execution of Parallel Query Plans

Depending on the complexity of the query execution plan, a parallel query plan can have one or more exchange operators. Because exchange operators require memory buffers and the memory requirement for an exchange operator is proportional to , the memory grant request value for a complex parallel query can be quite large.

### Synchronization Overhead of Parallel Operators

In the case of exchange operators, synchronization is required between producer and consumer threads that are accessing the same exchange buffers. The CXPACKET lock is used for this purpose, and it provides exclusive buffer access to requesting producer and consumer threads. In most parallel execution cases, the synchronization between producer and consumer threads seems to perform well. However, there are cases (for example, when either the producer or consumer thread is waiting for a resource such as I/O, CPU, or memory grants) in which excessive CXPACKET waits can occur. Other possible causes for high CXPACKET waits are underestimation of the number of rows flowing through an operator and highly skewed data. In such cases, as illustrated in Figure 4, the query optimizer may divide the total work and assign it to only a few threads, which results in a few threads doing most of the work and the remaining threads being idle. This can cause longer waits for acquiring the CXPACKET lock. Waiting for the CXPACKET lock is a consequence of thread parallelism. You should not worry about these types of waits unless they are too long and cause performance degradation.

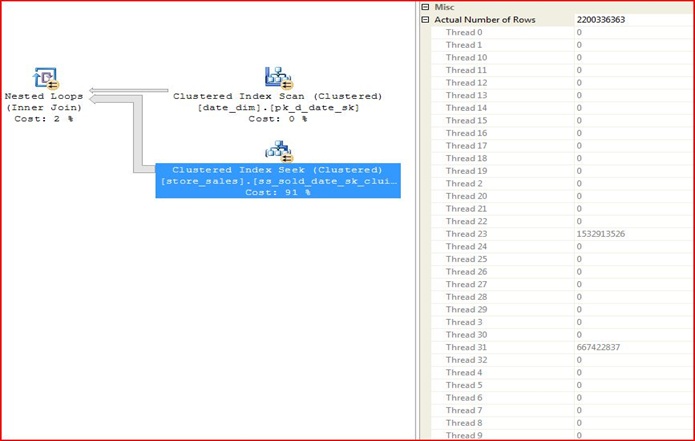


Figure 4: Imbalance in thread load distribution

### Statistical Estimation Errors

SQL Server collects statistical information about indexes and column data stored in the database. The SQL Server query optimizer uses these statistics to estimate the selectivity of expressions and determine the size of intermediate and final query result sets. Keeping the statistics up to date can provide the query optimizer with accurate information to access the cost of different query plans and enable the query optimizer to select better plans. Because the additional memory component of the memory grant request of a query is dependent on statistical estimates, lack or inaccuracy of statistical information can result in poor memory grant estimates. Underestimating the memory requirements can cause more data spills and thus hurt performance. On the other hand, overestimating the memory requirements can prevent other queries from running concurrently, because the memory grant requirements of the later-arriving queries cannot be met. Additionally, lack or inaccuracy of statistical estimates can also result in a poor plan choice such as selecting a parallel plan when a serial plan would execute faster or vice versa.

### Data Skew

Underestimation due to data skew can seriously hurt performance. Most data warehousing star-join queries access a fact table based on the values of fields in dimension tables. Generally the fact table is quite large compared to the dimension table, and a dimension table might have a particular row (known as a gatekeeper row) that is referenced by majority of rows in the fact table. In such cases, a star-join query expression could possibly output the majority of the rows from the fact table. If the gatekeeper row is absent, only a few rows are output.

For example, consider the query (Q), where A, B, and C are relations and σp is a filter that applies predicate p on relation B. Row b is a gatekeeper row in relation B:

Q = (A join σp(B) ) join C

If row b satisfies the predicate p, assume the number of rows in Q’s result set is 100 million. If b does not satisfy the predicate p, assume the number of rows in Q’s result set is 1. Such a data distribution can make the cardinality estimation process hard, and the large run-time cardinality estimation variance can result in in-accurate memory grant estimation and suboptimal query plan choices that cause performance degradation.

# Problem Identification and Mitigation

The previous section described problems that could limit parallelism. There is a huge amount of information available [1, 3] for troubleshooting performance problems. This section introduces some of the possible approaches that you can use to identify parallelism performance problems and provides some guidelines on what you can do to mitigate some of the issues.

## Degree of Parallelism and Memory Grants

Query degree of parallelism and memory grants can be monitored using the XML query plan. The **MemoryFractions** element in the run-time XML showplan gives the relative memory usage among operators in the query plan. The **MemoryGrant** attribute of the **QueryPlan** element also provides information regarding memory grants for a specific query. After a query is executed, the **DegreeOfParallelism** attribute of the **QueryPlan** element in the actual XML showplan, provides information of the actual degree of parallelism used.

In addition to using the XML query plan, SQL Server also provides a few dynamic management views (DMVs) and Performance Monitor counters to help with monitoring of query memory during run time. A few DMVs and Performance Monitor counters that can be useful are:

* sys.dm\_exec\_query\_memory\_grants*:*ThisDMV shows all query consuming memory grants including those waiting in the memory queue. Waiting queries have NULL values in the grant\_time column. The is\_next candidate column shows the next candidate query to wake up when memory is available. You can use the following query to find those queries that have not been granted memory yet and are waiting:

SELECT \*  
FROM sys.dm\_exec\_query\_memory\_grants  
WHERE is\_next\_candidate in (0,1)  
ORDER BY is\_next\_candidate desc, queue\_id, wait\_order;

* sys.dm\_os\_wait\_stats:ThisDMV shows wait statistics for all server objects. Since memory grants use RESOURCE\_SEMAPHORE type waits, you might notice significant waits on this wait type for complex memory intensive queries.
* The Memory Grants Pending Performance Monitor counter can tell you the number of waiting queries. The Memory Grants Outstanding Performance Monitor counter can tell you the number of queries that have grants and are currently executing.

A typical debugging scenario for a query that is timing out due to failure to obtain a memory grant may look like the following:

1. Check overall system memory status using sys.dm\_os\_memory\_clerks, sys.dm\_os\_sys\_info, and various performance counters. The Memory Grants Pending and the Memory Grants Outstanding Performance Monitor counters can be used for this purpose. If any of these counters are greater than or equal to 1 constantly, use the following steps to investigate further.
2. Check for query-execution memory reservations in sys.dm\_os\_memory\_clerks where type = 'MEMORYCLERK\_SQLQERESERVATIONS'.
3. Check for queries waiting for grants using sys.dm\_exec\_query\_memory\_grants.
4. Check query wait statistics for resource semaphores using sys.dm\_os\_wait\_stats where type=’RESOURCE\_SEMAPHORE’.

Further examine memory-intensive queries using sys.dm\_exec\_requests. If a runaway query is suspected, examine the showplan from sys.dm\_exec\_query\_plan and the batch text from sys.dm\_exec\_sql\_text.

## Worker Thread Balance

SQL Server parallelizes queries by horizontally partitioning the input dataset into equal-sized subsets, assigning each set to a particular worker thread, and then performing the same operation (for example, aggregate or join) on each set. In order to maximize processor utilization, it is important to balance computation and inter-thread communication. By monitoring the work queue count value, you can verify that all worker threads have been used. It might sound obvious that changing the value of max worker threads can solve the problem of lack of worker threads, but changing the value of max worker threads alone is not sufficient; other factors come into play. In fact, changing the value of max worker threads without adequate preproduction testing and system monitoring can actually have negative effects on performance. If you increase the number of worker threads, you also increase the amount of virtual memory that must be kept aside for thread stack space. This can deprive core components such as the buffer pool and execution engine of memory resulting in significant performance degradation. Additionally, due to more threads, there is also more contention and there needs to be more synchronization effort between threads. On the other hand, reducing the number of worker threads can cause multiple parallel incoming queries to stall waiting for sufficient threads to execute.

We recommend that database administrators understand your system’s workloads and adequately test your systems by monitoring the number of idle schedulers, CPU/Disk/Network utilization, and distribution of data before changing the value of max worker threads.

## Distribution of Data

To understand why distribution of data is an important factor that affects parallelism, consider the following part of an XML plan for a parallel sort operator that occurs as part of a CREATE INDEX operation.

<RelOp NodeId="**2**" PhysicalOp="**Sort**" LogicalOp="**Sort**" EstimateRows="**2.57398e+008**" EstimateIO="**2744.91**" EstimateCPU="**1958.46**" AvgRowSize="**35**" EstimatedTotalSubtreeCost="**7260.16**" Parallel="**1**" EstimateRebinds="**0**" EstimateRewinds="**0**">

[**-**](file:///\\dsdb\pss\cases\SRX090304600324\0317\XMLPLANX64.txt.xml) <OutputList>

<ColumnReference Column="**RowRefSrc1011**" />

</OutputList>

<MemoryFractions Input="**1**" Output="**1**" />

[**-**](file:///\\dsdb\pss\cases\SRX090304600324\0317\XMLPLANX64.txt.xml) <RunTimeInformation>

<RunTimeCountersPerThread Thread="**7**" ActualRows="**16059608**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**6**" ActualRows="**15888692**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**8**" ActualRows="**8269991**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**5**" ActualRows="**16257841**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**4**" ActualRows="**16733855**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**3**" ActualRows="**15253016**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**1**" ActualRows="**152841862**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**2**" ActualRows="**16092988**" ActualRebinds="**1**" ActualRewinds="**0**" ActualEndOfScans="**1**" ActualExecutions="**1**" />

<RunTimeCountersPerThread Thread="**0**" ActualRows="**0**" ActualRebinds="**0**" ActualRewinds="**0**" ActualEndOfScans="**0**" ActualExecutions="**0**" />

</RunTimeInformation>

The plan shows the actual number of rows processed by each thread using the ActualRows attribute. Thread 1 ended up sorting almost 152 million rows, whereas the remaining threads did considerably less work. We discovered that the root cause of this behavior was that the table had really skewed data. Out of 250 million rows, over 150 million rows contained empty strings for the column they were trying to build the index on. Because of skewed data distribution, the sort operator could not be truly parallelized and increasing the number of parallel worker threads did not improve performance.

However, if you run into a similar situation, you can mitigate this problem using table partitioning and building an index on a per-partition basis. Additionally, you can use filtered indexes to filter the skewed data values from the result set.

## Individual Threads Statistics and CXPACKET Waits

By using information available in the XML query plan information such as the DegreeOfParallelism attribute of the QueryPlan element and the RunTimeCountersPerThread element, monitoring on a per-thread basis can be done for individual queries.

CXPACKET type waits can be monitored using the sys.dm\_os\_wait\_statsDMV. Here is a sample SQL query that uses the DMV:

SELECT \* , (wait\_time\_ms - signal\_wait\_time\_ms) as resource\_wait\_time\_ms

FROM sys.dm\_os\_wait\_stats

ORDER BY resource\_wait\_time\_ms DESC

--ORDER BY wait\_time\_ms DESC

--ORDER BY signal\_wait\_time\_ms DESC

--ORDER BY waiting\_tasks\_count DESC

--ORDER BY max\_wait\_time\_ms DESC

Through research and previous investigation [9], we have learned that in the case of parallel queries on OLTP systems, the majority of the wait statistics are attributable to CXPACKET waits.

Figure5: Total wait statistics and CXPACKET waits before and after changing MAXDOP settings from 0(default) to 1(serial)

As shown in Figure 5, after the MAXDOP value was changed to 1, the CXPACKET waits observed were negligible compared to the number of CXPACKET waits when MAXDOP was set to 0.

If you reduce the degree of parallelism (setting MAXDOP to an explicit value greater than 0), you reduce the number of producer and consumer buffers by a quadratic factor. This change results in fewer CXPACKET locks and consequently fewer CXPACKET waits. The ideal MAXDOP setting for a system is highly dependent on a number of factors such as workload, data access, and data distribution patterns of the query. For OLTP workloads, generally, a MAXDOP value of 1 is reliable. For data warehousing workloads, MAXDOP values ranging between ¼ and ½ the number of logical processors provided by the machine typically works well. In addition, this again depends on a number of factors such as workload, data access, and data distribution. We recommend that you not modify the degree of parallelism value on production systems without sufficient testing.

As an example, if you are experiencing unpredictable multi-user parallel query performance on a machine with 32 logical processors, consider setting the max degree of parallelism **sp\_configure** option to 16 or even 8. For large queries that need the best possible response time, consider reducing the number of concurrently running queries (for example, during a batch window that you control) and running them with a MAXDOP value of 32.

## Indexes and Statistics

Indexes are vital to improving SQL Server performance. It is crucial to maintain up-to date statistics on tables to get good performance benefits.

The AUTO\_UPDATE\_STATISTICS option is bound by a 20 percent threshold limit, which means that statistics will be updated only if more that 20 percent of the table has changed. You should explicitly update statistics on tables where the cardinality estimates are off. For more information about how to improve query performance using better statistical estimates, see the white paper [Statistics Used by the Query Optimizer in Microsoft SQL Server 2008](http://msdn.microsoft.com/en-us/library/dd535534.aspx) (http://msdn.microsoft.com/en-us/library/dd535534.aspx).

# Recommendations

Start with the default value of MAXDOP and reduce it only if you identify performance problems. A good approach is to progressively test with powers of two values. If you only have one or a few concurrent queries running, the default for MAXDOP (0) will often be optimal. Following are some techniques you can use to understand problems related to parallel processing in SQL Server and tips for mitigating them.

1. If you observe excessive CXPACKET waits, reduce the value of MAXDOP:

* In an OLTP system, use a MAXDOP value of 1 (serial execution), if that is sufficient. Consider explicitly setting it to a higher value for individual queries that require parallelism.
* In a data warehousing system, where you need high parallelism for better overall response time, setting MAXDOP between ¼ the number of logical processors and ½ the number of logical processors generally works well. Experiment in preproduction environments to decide the MAXDOP value that gives you the best combination of throughput and response time for your environment.

1. Higher degree of parallelism queries generally require more memory to run. If several concurrently running complex queries each execute in parallel with a degree of parallelism that is greater than 1, the memory requirement may be significant. Consider using Resource Governor to throttle the degree of parallelism and total number of parallel queries by workload using the guidelines in [8].
2. If there is a lack of worker threads:

* Reduce MAXDOP.
* Use Resource Governor to throttle the degree of parallelism and total number of parallel queries by workload using the guidelines in [8].

1. If you are observing what appears to be a suboptimal query plan due to cardinality estimation errors, consider updating table statistics using the guidelines in [4].
2. Consider rewriting some queries so that they perform well in parallel environments using the guidelines in [2].
3. Remember that parallel query performance is multifaceted; there is no silver-bullet solution that can solve all parallel performance issues.

# Parallel Data Warehouse

In SQL Server Parallel Data Warehouse, data is horizontally partitioned across a set of nodes and data management is handled in parallel. The benefits of this approach are:

* The ability to use a larger number of processors via massively parallel processing (MPP).
* Improved performance with very large numbers of logical processors via a “divide and conquer” approach where data is hash partitioned across nodes, and further within nodes. This reduces contention for resources between logical processors working to solve a single query, improving overall scalability.

By partitioning the data among multiple smaller SMP nodes, and again within those nodes, potential problems such as excessive CXPACKET waits and scheduling of an imbalanced number of threads on the same core can be reduced when compared to running the same workload on a large SMP.

For more information about SQL Server 2008 R2 Parallel Data Warehouse, see the overview [here](http://www.microsoft.com/sqlserver/2008/en/us/parallel-data-warehouse.aspx) (http://www.microsoft.com/sqlserver/2008/en/us/parallel-data-warehouse.aspx).

# Conclusions

Parallel performance problems in SQL Server can occur due to a number of reasons. There is no single solution that works in all cases. By using some of the approaches discussed in this white paper, you can better understand the problems that affect you systems, and you can tune your system to create more predictable execution times even at higher degrees of parallelism.

# Bibliography

[1] *Performance Tuning Using Waits and Queues*, 2006. http://technet.microsoft.com/en-us/library/cc966413.aspx

*[2] Scaling Up Your Data Warehouse with SQL Server 2008*, 2008. http://msdn.microsoft.com/en-us/library/cc719182.aspx

*[3] Troubleshooting Performance Problems in SQL Server 2008*, 2009. http://msdn.microsoft.com/en-us/library/dd672789.aspx

[*4] Statistics Used by the Query Optimizer in SQL Server 2008*, 2009. http://msdn.microsoft.com/en-us/library/dd535534.aspx

*[5] Fast Track Data Warehouse 2.0 Architecture*, 2009. http://technet.microsoft.com/en-us/library/dd459178.aspx

*[6] Data Warehousing 2.0 and SQL Server 2008, 2009*. http://msdn.microsoft.com/en-us/library/ee730351.aspx

*[7] Repartition Streams Showplan Operator*, 2009. http://msdn.microsoft.com/en-us/library/ms190783.aspx

*[8] Using the Resource Governor, 2009.* http://msdn.microsoft.com/en-us/library/ee151608(SQL.100).aspx

*[9]Case Study - CXPACKET Wait Stats and ‘max degree of parallelism’ option, 2008.* http://blogs.msdn.com/b/jimmymay/archive/2008/12/02/case-study-part-2-cxpacket-wait-stats-max-degree-of-parallelism-option-suppressing-query-parallelism-eliminated-cxpacket-waits-liberated-30-of-cpu.aspx