# logo-sql08.gif

Using the Resource Governor

SQL Server Technical Article

**Writer:** Aaron Bertrand, Boris Baryshnikov

**Technical Reviewers**: Louis Davidson, Mark Pohto, Jay (In-Jerng) Choe

**Published:** June 2009

**Applies to:** SQL Server 2008

**Summary:** SQL Server 2008 introduces a new feature, the Resource Governor, which provides enterprise customers the ability to both monitor and control the way different workloads utilize CPU and memory resources on their SQL Server instances. This paper explains several practical usage scenarios and gives guidance on best practices.

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.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, Access, Excel, Hyper-V, MSDN, SQL Server, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

[Introduction 6](#_Toc234040919)

[What Is the Resource Governor? 6](#_Toc234040920)

[Why Use the Resource Governor? 8](#_Toc234040921)

[Consolidation 8](#_Toc234040922)

[Predictable Performance and Preventing the "Runaway" Query 8](#_Toc234040923)

[Meeting Service-Level Agreements 8](#_Toc234040924)

[Chargeback 9](#_Toc234040925)

[Common Usage Scenarios 9](#_Toc234040926)

[Consolidation 9](#_Toc234040927)

[Initial Assessment 10](#_Toc234040928)

[Implementing Controls 13](#_Toc234040929)

[Ongoing Monitoring 14](#_Toc234040930)

[Testing Before Consolidating 15](#_Toc234040931)

[Preventing and Finding the “Runaway” Query 16](#_Toc234040932)

[Meeting Service-Level Agreements 17](#_Toc234040933)

[Response Time and Concurrency for Interactive Users 18](#_Toc234040934)

[Delivery Time for Scheduled Reports 21](#_Toc234040935)

[Limiting Parallelism for Intensive Background Jobs 22](#_Toc234040936)

[Chargeback 25](#_Toc234040937)

[Best Practices 26](#_Toc234040938)

[Test the Dedicated Administrator Connection – Then Test It Again 27](#_Toc234040939)

[Optimize the Classifier Function 27](#_Toc234040940)

[Use the PreConnect Trace Events 27](#_Toc234040941)

[Test the Classifier Function Under Load 27](#_Toc234040942)

[Avoid Lookup Tables 27](#_Toc234040943)

[Optimize Unavoidable Lookups 28](#_Toc234040944)

[Use Local Variables Where Appropriate 28](#_Toc234040945)

[Avoid Expensive String Operations 28](#_Toc234040946)

[Only Classify for User-Defined Workload Groups 28](#_Toc234040947)

[Test the Classifier Function – Then Test It Again 29](#_Toc234040948)

[Do Not Let the Classifier Function Trust Everything 31](#_Toc234040949)

[Place Reasonable Limits on the Default Pool and Group 31](#_Toc234040950)

[Use Memory Constraints with Caution 31](#_Toc234040951)

[Maintain Similar Test and Development Servers 32](#_Toc234040952)

[Test with Adequate Background Workloads 32](#_Toc234040953)

[Plan Ahead, if You Can 32](#_Toc234040954)

[Always Be Monitoring, Evaluating, and Adapting 33](#_Toc234040955)

[Keep Demos Simple 33](#_Toc234040956)

[Conclusion 34](#_Toc234040957)

[Appendix A – Resource Governor in Depth 36](#_Toc234040958)

[Resource Pools 36](#_Toc234040959)

[Workload Groups 38](#_Toc234040960)

[Classifier Function 41](#_Toc234040961)

[Catalog Views 43](#_Toc234040962)

[Dynamic Management Views 44](#_Toc234040963)

[How it All Works Together 45](#_Toc234040964)

[Appendix B – Resource Governor Overhead 46](#_Toc234040965)

[Dynamic Management Views 46](#_Toc234040966)

[Classifier Function 46](#_Toc234040967)

[Tracing and Eventing 46](#_Toc234040968)

[Appendix C – Resource Governor Permissions 47](#_Toc234040969)

[Appendix D – Monitoring the Resource Governor 48](#_Toc234040970)

[Dynamic Management Views 48](#_Toc234040971)

[sys.dm\_resource\_governor\_resource\_pools 48](#_Toc234040972)

[sys.dm\_resource\_governor\_workload\_groups 48](#_Toc234040973)

[Performance Monitor 50](#_Toc234040974)

[Using Performance Monitor 51](#_Toc234040975)

[Showplan and max degree of parallelism 52](#_Toc234040976)

[SQL Trace 53](#_Toc234040977)

[CPU Threshold Exceeded 54](#_Toc234040978)

[PreConnect:Starting 55](#_Toc234040979)

[PreConnect:Completed 55](#_Toc234040980)

[Extended Events 55](#_Toc234040981)

[An Example: Learning About Your System 57](#_Toc234040982)

[Appendix E – Resource Governor Limitations 65](#_Toc234040983)

[No Constraints on I/O 65](#_Toc234040984)

[Database Engine Only 65](#_Toc234040985)

[Single Instance Only 65](#_Toc234040986)

[Enterprise Edition Only 65](#_Toc234040987)

[Resource Usage Anomalies 66](#_Toc234040988)

[Appendix F – Cleaning Up the Resource Governor Configuration 69](#_Toc234040989)

# Introduction

The Resource Governor is a new feature in the Microsoft® SQL Server® 2008 Enterprise. It provides very powerful and flexible controls to dictate and monitor how a SQL Server instance behaves in terms of CPU, memory and response time.

This white paper is targeted at IT professionals and database administrators responsible for managing scale, concurrency, high availability, consolidation, virtualization, or service level agreements for the SQL Server instances in their production environments. In this paper you are shown various ways in which you can use the Resource Governor to analyze and streamline the way resources are utilized within your SQL Server instances, by explaining:

* How the Resource Governor works, including limitations and best practices.
* Methods for building a consolidation plan, and testing those methods.
* Techniques for preventing runaway queries, or at least minimizing their impact.
* Approaches to meeting service level agreements (SLAs) driven by response time and concurrency.
* Basic ideas about implementing a chargeback system to recover resource costs.
* Ways to monitor SQL Server using new Resource Governor functionality.

The paper will highlight several common usage scenarios, help you decide when and where to use the technology, and outline best practices when using the Resource Governor.

**Note:** If you are not familiar with the Resource Governor, it would be helpful to have a basic understanding of how the feature works; the Appendices will help outline the core fundamentals, and point to official documentation where appropriate.

# What Is the Resource Governor?

The Resource Governor is a set of technologies within the Database Engine that allows you to place controls on various facets of CPU and/or memory, either at an aggregate level (one or more groups of users), or at an individual request level. You can think about the limit controls like the governor chip in a car, which cuts power to the engine when a hard-coded speed limit is reached (usually somewhere over 120 mph, varying by manufacturer). The intention is to prevent excess speed, and the motivation behind the feature is obviously safety. While it is true that you can configure the Resource Governor to behave in a similar manner, attempting to prevent targeted requests from exceeding certain resource utilization levels, it is much more than that. You can use it not only to limit maximum usage in certain scenarios, but also to try to guarantee minimums, enabling you to fine tune and balance your workloads, and give the right resources to the right users at the right times. You can also use it to observe resource usage, allowing you to record the results and to tweak your settings periodically to maximize effectiveness.

There are three fundamental components to the Resource Governor: resource pools, workload groups, and the classifier function. Briefly, a resource pool can be thought of as a “slice” of all of the CPU and memory resources available to the SQL Server instance, and a workload group as a set of similar requests (as defined by you). The classifier function enables incoming requests to be associated with a specific workload group (and, in turn, a single resource pool). These requests are then constrained by the Resource Governor; but in several cases, only when there is significant contention for the resource(s).

In the following diagram, we can follow any request type through the classifier function (or in the case of DAC and internal requests, where they bypass the function altogether), then into the workload group assigned by the classifier function (or the default group, if no group is assigned or the group returned by the function is invalid), and then into the appropriate resource pool. Essentially, the classifier function assigns an incoming connection to a specific workload group, which in turn corresponds to exactly one resource pool. After classification has occurred, the Resource Governor knows which workload group and resource pool the session is associated with, and so it can apply the known constraints accordingly, depending on the mix of settings and server state. All requests from that session are then routed to the server so that they can be executed. Note that a non-classified request in this case can mean that (a) the classifier function does not have logic that correctly captures the specific request (or returns an invalid workload group name), or (b) the classifier function is not enabled, in which case the request falls into the default workload group.

 **Figure 1:** Path of requests through classifier function, workload groups, and resource pools

The appendices and associated materials will provide much more detail on the individual components of the Resource Governor and how they work together.

# Why Use the Resource Governor?

There are several primary areas where the Resource Governor can be used to increase performance, reduce costs, or improve manageability. The key areas this paper will focus on are consolidation, predictable performance, meeting service-level agreements (SLAs), and chargeback.

## Consolidation

There is a great deal of pressure on IT professionals and database administrators alike to do more with less. A common trend today – unlike just a few short years ago – is to reduce data center footprint and SQL Server sprawl through consolidation and/or virtualization onto fewer, more powerful and reliable servers, as opposed to scaling out across a seemingly endless stream of smaller, redundant machines. While the concept sounds simple, it is not always easy to take applications running on different servers and make them perform well if they share the same physical resources of a single server. The Resource Governor can help in this case by providing you with the ability to monitor and determine resource usage across your many SQL Server instances long before you consolidate, which is something you cannot do easily with other consolidation solutions such as virtualization, and combining instances onto fewer machines – in those cases you just need to dive in, or procure additional hardware. After you have combined servers, you can continue monitoring CPU and memory utilization and, when needed, you can use Resource Governor constraints to ensure that one workload does not consume too many resources.

## Predictable Performance and Preventing the "Runaway" Query

Today’s database administrator is, for the most part, very reactive when there is a performance problem, such as one session taking too many resources, or another session not getting enough. You can kill SPIDs or applications that have run away or are otherwise monopolizing resources on an instance. You can isolate certain workloads by replicating or splitting your data across multiple instances, and setting different instance-level constraints for each set. You can use QUERY\_GOVERNOR\_COST\_LIMIT at the session or server level, which simply rejects plans that are estimated to exceed the time you specify. Or you can bend over backwards proactively – by juggling SQL Server Agent jobs, taking applications offline during specific windows, or scheduling password or procedure changes – to prevent certain workloads from running during times of the day or week that you know are problematic, or to ensure that specific users get much more predictable use out of the system.

The Resource Governor can help in these scenarios by providing resource minimums to certain groups or applications, and allowing adequate resources such that batch processes can finish within their allocated windows. You can also introduce upper boundaries on CPU and/or memory for different groups, to provide predictable performance and to prevent the "runaway" query.

## Meeting Service-Level Agreements

Many companies are bound by service-level agreements (SLAs), which can dictate that a server or application must meet specific criteria, such as uptime, response time, and concurrency. Outside of preventing a runaway query from bringing down your server, the Resource Governor is not intended to provide uptime guarantees for your hardware and services – regardless of how you limit and prioritize requests, the Database Engine cannot insulate you from a power surge, disk failure, network outage, or faulty maintenance tactics. The Resource Governor can, however, assist greatly in achieving fast response times and concurrency for preferred requests. A preferred request in this case might be a specific application, a specific user or group, or even a scheduled report that needs to be delivered by a specific time or within a defined window.

## Chargeback

In shared hosting scenarios there is typically a single SQL Server instance with dozens or even hundreds of individual databases, each belonging to a unique customer or business unit. While in some cases the performance experienced by any of those customers is based on the old adage, “you get what you pay for,” in other cases, terms of service may dictate that you pay for what you use. What this means is that a customer's monthly or annual fee may be a mixture of a fixed cost, plus an incremental cost for disk space, plus incremental costs for CPU and memory utilization. Using SQL Server today, it can be very cumbersome to audit this resource usage and apply the costing to the appropriate groups or customers. The Resource Governor can allow you to track resource usage among many different workloads, groups, or applications over time, allowing you to not only bill business units appropriately, but also to gauge which of those groups are your biggest resource consumers.

# Common Usage Scenarios

You have been introduced to several scenarios where the Resource Governor could help to improve performance, increase profitability, and maximize productivity. The following sections will treat these scenarios individually and in greater detail.

## Consolidation

Whether you are consolidating to reduce costs, to stay within data center capacity, or simply to minimize the number of SQL Server instances you need to manage, the approach to consolidation can be quite complex. There are various types of consolidation that can be considered:

* Consolidating multiple instances from different servers onto a single server
* Consolidating multiple databases from different instances into a single instance
* Virtualizing multiple servers onto fewer physical hosts

All of these consolidation types will require a similar approach; however, small details may be slightly different. For example, if you are talking about database or application consolidation within a single server, you will not necessarily need an exhaustive inventory of resource utilization for the server as a whole, because the server hardware is not going to change.

You will need to take performance metrics from your various servers and instances to determine which databases are candidates for consolidation, and which are not. You will find some instances that can be combined with ease, and others that would not play well together at all. For example, two instances that both max out CPU on their respective machines throughout business hours are not going to be stellar candidates for consolidation, because even with the help of the Resource Governor, the high demands may still lead to unacceptable performance. On the other hand, an instance that generates reports overnight could very well move to the same server as an instance that handles OLTP activity during business hours, because these schedules and workloads are quite complementary.

### Initial Assessment

The Resource Governor can help by allowing you to collect metrics on each instance to help in the assessment process. Specifically, you can collect real-time data on CPU usage, memory usage, and concurrency. Obviously there are other factors you will want to consider, such as I/O, storage requirements, network utilization, and security implications. You can capture these elements through other means, but the focus here is on how the Resource Governor can assist in a consolidation effort.

**Note:** You do not need to have the Resource Governor enabled or configured to utilize the dynamic management views (DMVs) and Performance Monitor counters. Even when Resource Governor is disabled, the DMVs still collect information for the default and internal workload groups and resource pools. In cases where the Resource Governor is (or has been) enabled, you will need to consider these built-in resource pools and workload groups along with any pools and groups you may have configured.

You should ensure that you are collecting data frequently enough, and that you capture a complete business cycle – a common mistake in consolidation efforts is neglecting to capture data about infrequent spikes in activity such as month-end or year-end reporting. Throughout such a business cycle, you can take frequent snapshots of certain pieces of data from the performance monitor counters. In this specific example you will use the Performance Monitor counters as opposed to the Resource Governor DMVs for two advantages: (1) there are more metrics available in the performance counters than in the DMVs, and (2) unlike querying the DMVs, you can collect data from performance counters with little to no impact on the server. Some of the performance counter metrics that will be interesting include:

**Workload Group Stats**

* Active parallel threads
* Active requests
* Blocked tasks
* CPU usage %
* Max request CPU time (ms)

**Resource Pool Stats**

* Active memory grant amount (KB)
* CPU usage %
* Pending memory grants count
* Used memory (KB)

For more information, including the full set of performance counters relevant to Resource Governor, see the following SQL Server Books Online topics:

**SQL Server, Resource Pool Stats Object**<http://msdn.microsoft.com/en-us/library/cc645958.aspx>

**SQL Server, Workload Group Stats Object**<http://msdn.microsoft.com/en-us/library/cc627354.aspx>

To capture these metrics, you could set up repository tables that hold data for each snapshot you take throughout your business cycle, and a scheduled job that populates the table. For simplicity, in this example you will create a table that collects data locally from the DMV **sys.dm\_os\_performance\_counters**. In a real-world scenario, you would likely set up something more elaborate, where you are querying the performance counters from outside of the database (perhaps using software tools like Systems Center), and storing both DMV and performance monitor data in a different system. The reason you would want to offload this data collection onto another server is that, on a busy server with frequent polling, it can get out of hand fairly quickly. And if the end goal of the project were to optimize the performance on this specific server, bogging it down with additional load would be quite counterproductive, and could also severely taint your analysis.

The following table and query can give you a starting point on how to capture performance counter metrics on an interval. Once again for simplicity, the table does not have any constraints, keys, or indexes, and the data types are based mostly on what comes directly out of the performance counters DMV. Also, there is some complex logic in the query that normalizes CPU usage percentage (because it is based in part on another counter), but you could certainly opt to perform these calculations at analysis time as opposed to collection time:

CREATE TABLE dbo.ConsolidationMetrics

(

 SnapshotTime DATETIME,

 [object\_name] NVARCHAR(256),

 counter\_name NVARCHAR(256),

 instance\_name NVARCHAR(256),

 counter\_value BIGINT

);

-- schedule this query:

;WITH rawstats AS

(

 SELECT

 [object\_name],

 counter\_name,

 instance\_name,

 val = cntr\_value

 FROM

 sys.dm\_os\_performance\_counters

 WHERE

 [object\_name] LIKE N'%:[WR]% Stats%'

 AND counter\_name IN

 (

 N'CPU usage %', N'CPU usage % base',

 N'Active requests', N'Blocked tasks',

 N'Active parallel threads',

 N'Max request CPU time (ms)',

 N'Active memory grant amount (KB)',

 N'Pending memory grants count',

 N'Used memory (KB)'

 )

)

INSERT dbo.ConsolidationMetrics

(

 SnapshotTime, [object\_name], counter\_name,

 instance\_name, counter\_value

)

SELECT

 CURRENT\_TIMESTAMP, s1.[object\_name],

 s1.counter\_name, s1.instance\_name,

 CASE WHEN s1.counter\_name <> N'CPU usage %'

 THEN s1.val

 ELSE

 CONVERT(BIGINT, s1.val \* 100.0 / COALESCE(s2.val, 1))

 END

FROM

 rawstats AS s1

LEFT OUTER JOIN

rawstats AS s2

ON

s1.counter\_name = N'CPU usage %'

 AND s2.counter\_name = N'CPU usage % base'

 AND s1.[object\_name] = s2.[object\_name]

 AND s1.instance\_name = s2.instance\_name

WHERE

 s1.counter\_name <> N'CPU usage % base';

Alternatively, you could create a custom data collector for use with data collection and the management data warehouse, described here:

**Data Collection**<http://msdn.microsoft.com/en-us/library/bb677179.aspx>

You will need to decide how often you want to take snapshots; the trade-off is going to be performance vs. accuracy. Obviously if you are writing data to these tables every few seconds, performance will suffer; on the other hand, if the delay between snapshots is too long, you risk losing data that might accumulate just before a failover, service restart, or manual Resource Governor statistics reset. A starting compromise in your scenario might be every five or ten minutes, and that interval can be adjusted as you observe its impact.

After you have collected a representative sample that covers at least a full business cycle, you can inspect the data to guide you on how these instances can be combined. If you are collecting data from the Resource Governor DMVs, you need to be careful that you account for situations where the Resource Governor statistics have been reset – this can occur through a manual command or because the SQL Server service was restarted. If you try to graph this data without accounting for resets, you might see, for example, a misleading dip in total CPU or memory utilization. So in that case, in addition to **SnapshotTime**, you should add a column **LastStatisticsResetTime**, which will carry the DMVs’ **statistics\_start\_time** column.

**Note** There is a wide variety of other metrics you may want to capture in your consolidation assessment, and there are several articles and papers out there that will help you do that. Appendix D, “Monitoring the Resource Governor,” has more information about the metrics you can collect to measure resource utilization in your SQL Server instances; however, performing a complete consolidation guide is out of scope for this paper.

### Implementing Controls

Based on the metrics you have captured, upon consolidation, you can implement controls that will optimize different workloads based on the characteristics you have observed. You might want to ensure that one workload will not take more than 50 percent of the CPU or 60 percent of the memory. For this you will need the Resource Governor enabled; you can easily enable the Resource Governor and configure each workload for these constraints within SQL Server Management Studio or through DDL commands. Here is the interface in SQL Server Management Studio.

 **Figure 2:** Resource Governor Properties dialog box

As with many high availability scenarios, one thing to ensure is that you plan for peak workloads. For the same reason we keep backups and have disaster recovery plans, you should always be prepared for the "worst case scenario.” As an example, it might not be the best idea to consolidate two applications that each max out CPU or memory but at different times, because different factors can lead to those workloads colliding and resulting in poor performance (or even failures). So, when planning to bring multiple workloads together, you will need to be confident that your systems will be able to handle the load when all workloads are at full capacity.

### Ongoing Monitoring

You can and should continue monitoring your workloads after configuration. You can see several ideas for monitoring in Appendix D. It will be important over time to measure whether your initial assumptions about how your workloads consume resources are still accurate. Changes to applications, database schema, or even hardware or network infrastructure can have a great impact on your biggest bottlenecks, and change the way your workloads interact. In these cases you may need to go back and reconfigure, which could mean anything from slightly tweaking your Resource Governor constraints to changing your entire consolidation strategy.

### Testing Before Consolidating

You may want to test how well a consolidation effort will work, prior to actual implementation. An obvious problem in a lot of environments is the inability to test this scenario. It can be quite time consuming, but typically you would set up an additional server, make a test copy of the application and its database(s), combine them onto this one server, and perform load tests. These conditions are typically unrealistic because there is not enough volume, not enough of the right kind of volume, or hardware that is different enough to skew performance. Other blocking factors here could be the availability of hardware and labor, among other things.

An alternative could be to use the Resource Governor separately on both instances, using the constraints available to simulate resources being consumed by other services on the same machine. This would allow you to essentially test consolidation without having to actually move instances between hardware. However, because Resource Governor limitations are only engaged when there is significant contention, you will need to ensure that you are running adequate secondary workloads while testing your constraints. Without sufficient resources being requested by workloads other than the one(s) you are testing, you will not truly be measuring the effect of the constraints, because in many cases they will be ignored. This scenario is covered briefly in the Best Practices section.

While certainly you could argue that playing with the **affinity mask** and **min/max server memory** settings of **sp\_configure** on each individual server could help accomplish this same goal, there are at least three advantages that the Resource Governor can provide; namely, the ability to:

1. Fine-tune constraints per workload group, so that you can measure the effects on individual applications or users, instead of affecting the whole instance. You can tweak these settings in literally hundreds of different ways in order to test all possible scenarios, finding the right balance that will work when consolidating.
2. Quickly revert constraint settings without stopping and restarting the SQL Server service. This will minimize the impact your testing will have on users and other applications, especially if you inadvertently generate an unfavorable constraint with negative consequences.
3. Measure the impact on different workloads without having to run them separately or in isolation. The most notable place where you can see this is the Performance Monitor counter Resource Pool Stats: CPU control effect %, which will tell you exactly what kind of effect consolidation might have on your CPU-bound workloads. You can also run the same (or multiple) workloads over and over again, with slightly different constraints, to see what effect they have on overall CPU. Of course, as demonstrated throughout this paper, there are dozens of other measures you can use to analyze the way your workloads respond to various artificial limitations.

Even if you don’t plan to use Resource Governor constraints in production, you can use the technology in this way to predict how your workloads may perform if and when you consolidate in the future. This kind of testing can guide you on which instances would complement each other in a resource sharing scenario, and it may even lead you to split databases (representing separate workloads) onto different instances in order to minimize the impact of consolidation.

## Preventing and Finding the “Runaway” Query

Because the “runaway” query can come from a variety of sources, such as ad hoc queries, dynamic SQL, user-defined reporting, or Microsoft Office programs such as the Office Access® database software and the Office Excel® spreadsheet software, it is extremely difficult to predict in advance where the next one will come from. Rather than trying to place tight controls on these individual query sources, you can, instead, place a safety net on the default resource pool and workload group to protect your system. Meanwhile, you can create distinct workload groups and resource pools for the controlled applications you recognize (or users that you trust), that you are confident aren’t going to suddenly generate very resource-intensive queries that will take over your instance. This way, you don’t limit everyone unnecessarily; only those that you are not too sure about.

As an example, imagine you have a number of “information workers” that run user-defined queries against the database throughout the day, using a variety of end-user tools. You also have a number of controlled applications that you recognize as well behaved. Just picking numbers arbitrarily, you could place a cap on memory at 20 percent and CPU at 35 percent for the default resource pool, limit degree of parallelism to 1 for the default workload group, and be able to capture any cases where a query runs longer than, say, five minutes. You could also create a separate resource pool and workload group for your applications and other “known” system users, with no resource constraints, allowing them much more freedom to utilize resources. This setup would look something like this:

USE [master];

GO

ALTER RESOURCE POOL [default] WITH

(

 MAX\_MEMORY\_PERCENT = 20,

 MAX\_CPU\_PERCENT = 35

);

ALTER WORKLOAD GROUP [default] WITH

(

 MAX\_DOP = 1,

 REQUEST\_MAX\_CPU\_TIME\_SEC = 300

);

CREATE RESOURCE POOL [Pool\_WellBehaved];

CREATE WORKLOAD GROUP [Group\_WellBehaved] USING [Pool\_WellBehaved];

GO

CREATE FUNCTION dbo.classifier\_PreventRunaway()

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 RETURN

(

 SELECT CASE

 -- BEWARE OF SPOOFING:

 WHEN APP\_NAME() = N'known application name'

 OR SUSER\_SNAME() IN ('list','of','trusted','users')

 THEN N'Group\_WellBehaved'

 ELSE N'default'

 END

);

END

GO

ALTER RESOURCE GOVERNOR WITH

(CLASSIFIER\_FUNCTION = dbo.classifier\_PreventRunaway);

GO

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

Keep in mind that this is just a rough guideline, and the actual settings for your particular environment and workloads may take some experimenting to perfect. You may want to try to lump all of the information workers into their own workload group and resource pool, rather than the method outlined above, depending on whether you find it easier to identify the good or bad set of requests.

To find the bad queries that have run for longer than five minutes, you can simply script a server-side SQL Trace that captures to a file the event Errors and Warnings:CPU threshold exceeded. Note that you can filter on workload group, using the **group\_id** column, so that you only capture data for the default workload group (or the group(s) you are interested in). However, if you have any user-defined workload groups and you filter on **group\_id**, you take the risk that the expensive queries may come from a user or application that bypasses the logic in your classifier function or otherwise ends up in a workload group that you are not capturing. There is more information on using SQL Trace in Appendix D, “Monitoring the Resource Governor.”

The events you find in the trace will not tell you whether a request took longer simply because of the resource constraints you placed on the workload group, but they will at least highlight the most troublesome queries on the system. And the caps you place on the resource pool make sure that, while these requests may take longer, they will not monopolize the CPU or memory of the system in the process.

## Meeting Service-Level Agreements

As mentioned previously, you may be obligated to provide a minimum level of service to your customers, and this level of service can consist of response time and concurrency, as well as other criteria. You may be required not only to meet these SLAs but also to prove that you are meeting them consistently. The Resource Governor can help in various ways, as outlined below.

### Response Time and Concurrency for Interactive Users

In order to provide fast response times for a specific application or set of users, you can easily create a resource pool with guaranteed minimums for both CPU and memory; at the same time, you can place caps on the default and other pools, to maximize the shared percentage of resources for the preferential pool. Imagine you have a set of users that connect to the system all the time, and regardless of what else is going on in the server, you want their requests to finish as fast as possible. You may even be mandated to have all queries complete within a specific timeframe; for the sake of this example, assume this is two minutes. You could set up the default and preferential resource pools in the following way.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Component | Resource Pool | MIN % | MAX % | Effective MAX % | Calculated Shared % |
| CPU | default | 0 | 50 | 50 (MIN (50, 100-30)) | 50 (50-0) |
| Pool\_Preferential | 30 | 90 | 90 (MIN (90, 100-0)) | 60 (90-30) |
| Memory | default | 0 | 40 | 40 (MIN (40, 100-25)) | 40 (40-0) |
| Pool\_Preferential | 25 | 90 | 90 (MIN (90, 100-0)) | 65 (90-25) |

 **Table 1:** Arbitrary settings to attempt to prioritize a resource pool

**Note:** The details on calculating effective and shared maximums for different pools are explained in the following SQL Server Books Online topic:

**Resource Governor Concepts**<http://msdn.microsoft.com/en-us/library/bb934084.aspx>

In addition, you could limit degree of parallelism and concurrency for the default workload group by using the MAX\_DOP and GROUP\_MAX\_REQUESTS settings. (Settings like IMPORTANCE are only worthwhile to use if you are trying to give more resources to requests within a single resource pool, so they would not be useful here.) And finally, you can add a CPU threshold for the preferential workload group, so you can track when any queries exceed your SLA time.

Combining these constraints with the settings on the resource pool should give priority to your preferred users, by ensuring that they get a minimum set of resources, and that no other users can take too much.

In addition to merely providing your preferred users with a positive experience, you may need to *prove* to your customers that you are meeting your SLAs. Consider that you may need to answer questions such as, for any time period:

1. What was the longest time required for any query?
2. What was the highest number of simultaneous queries?
3. How many queries exceeded the SLA threshold time?

Because the DMVs can be reset manually or whenever the service is restarted, you can’t just rely on instance uptime to be sure that the DMVs will always be able to answer the above questions for the relevant time frame. Instead, you will have to periodically take snapshots of the DMVs and then aggregate them on a schedule. In this case, you could set up a SQL Server Agent job that polls the Resource Governor DMVs for key metrics periodically, and stores the data so you can report on it later. As mentioned previously, you could use a custom data collector and the management data warehouse; you could also use Performance Monitor counters.

If you are going to collect the DMV data yourself, you can do this by creating a table that holds the specific data you are interested in, for example:

CREATE DATABASE SLA;

GO

USE SLA;

GO

CREATE TABLE dbo.SLA\_Snapshots

(

 group\_id INT,

 snapshot\_time SMALLDATETIME,

 statistics\_start\_time DATETIME,

 active\_request\_count INT,

 total\_cpu\_limit\_violation\_count INT,

 max\_request\_cpu\_time\_ms INT

 PRIMARY KEY (group\_id, snapshot\_time)

);

(Note that it may make sense to store this repository data in a separate instance, depending on the frequency of collection and other factors.)

Then you can create this stored procedure, which you could call on demand or from a scheduled job:

CREATE PROCEDURE dbo.SLA\_CaptureSnapshot

 @group\_id INT

AS

BEGIN

 SET NOCOUNT ON;

 INSERT SLA.dbo.SLA\_Snapshots

 (

 snapshot\_time,

 statistics\_start\_time,

 active\_request\_count,

 total\_cpu\_limit\_violation\_count,

 max\_request\_cpu\_time\_ms

 )

 SELECT

 CURRENT\_TIMESTAMP,

 statistics\_start\_time,

 active\_request\_count,

 total\_cpu\_limit\_violation\_count,

 max\_request\_cpu\_time\_ms

 FROM

 sys.dm\_resource\_governor\_workload\_groups

 WHERE

 group\_id = @group\_id;

END

GO

The first two questions above can be answered easily, using the following query:

DECLARE

 @group\_id INT,

 @StartDate SMALLDATETIME,

 @EndDate SMALLDATETIME;

-- populate these values appropriately:

SELECT

 @group\_id = n,

 @StartDate = 'YYYYMMDD',

 @EndDate = 'YYYYMMDD';

SELECT

 longest\_single\_query = MAX(max\_request\_cpu\_time\_ms),

 highest\_concurrent\_query\_count = MAX(active\_request\_count)

FROM

 dbo.SLA\_Snapshots

WHERE

 group\_id = @group\_id

 AND snapshot\_time >= @StartDate

 AND snapshot\_time < @StartDate + 1;

This will return a resultset that shows the longest CPU time and the highest number of concurrent requests observed for the workload group during the specified period. However, for the third question, because the statistics can be reset in the middle of the day, it is not simply a matter of taking a max over the time frame. Because this particular value measures a count, and the value is reset to 0 whenever the statistics are reset, you need to keep a running total for each segment of snapshots for a timeframe that overlaps with one or more statistics resets. To do this you can simply take the maximum value found in the column for each distinct statistics reset date, and then add those maximums together, as demonstrated in this query:

DECLARE

 @group\_id INT,

 @StartDate SMALLDATETIME,

 @EndDate SMALLDATETIME;

-- populate these values appropriately:

SELECT

 @group\_id = n,

 @StartDate = 'YYYYMMDD',

 @EndDate = 'YYYYMMDD';

;WITH maxStats AS

(

 SELECT

 statistics\_start\_time,

 max\_violations = MAX(total\_cpu\_limit\_violation\_count)

 FROM

 dbo.SLAMetrics

 WHERE

 group\_id = @group\_id

 AND snapshot\_time >= @StartDate

 AND snapshot\_time < @StartDate + 1

 GROUP BY

 statistics\_start\_time

)

SELECT

 total\_violations = SUM(max\_violations)

FROM

maxStats;

If your requirements are simply to report on response time violations, you could achieve this simply by running a trace in the background to record whenever the CPU threshold is exceeded. In fact, you could run a much more lightweight trace than the one described previously; if you are only reporting on number of violations per day, for example, you will not necessarily need to correlate the events with specific queries. Instead you could create a trace that simply records the StartTime and GroupID for the CPU Threshold Exceeded event (along with required columns **EventClass** and **SPID**). This will allow you to record a much smaller data set, both in terms of the amount of data per row, and the number of rows.

In your situation, the resource constraints you should place on your workload groups and resource pools may vary greatly (and could change over time), depending on a large number of factors. Your system’s ability to meet stringent response time requirements, for example, relies largely on proper hardware and a well-designed database that is properly indexed and maintained. If you know that a given query can’t meet the dictated response time in isolation, never mind on a busy system, promising that response time does not make much sense in the first place. The above should simply serve as a starting point to test those boundaries and determine the best way to segment and prioritize your workloads for the purposes of meeting response time and concurrency SLAs.

Recording, aggregating, and reporting on these metrics could quite literally deserve a paper all on their own. In addition to the metrics you will need to report to your customers, you may also want to capture additional data for your own benefit, such as total CPU time, number of memory grant timeouts, and other statistics. The goal here is to understand that the DMVs provide a rich set of data that you can collect and fine tune to help you meet your SLAs, and to determine why, if your system is unable to do so.

### Delivery Time for Scheduled Reports

A situation you might have experienced is the need to deliver reports to clients or partners in a timely fashion. You might have a month-end report, for example, that can take several hours to run, and consume a lot of memory in the process, but needs to be delivered by 9:00 A.M. on the first day of every month. A problem with this scenario is that, if the report cannot access all of the memory it needs, it could take longer, and may not be delivered on time.

You might think that you could simply reserve the memory ahead of time, by placing a minimum amount of memory on a certain resource pool, and making sure that only this report ever gets access to the resources allocated to that pool. However, this is quite wasteful; why reserve a chunk of memory all the time, for a task that only runs once a month? Effectively, you would be preventing any other requests from using that memory when they need it most.

One idea to get around this problem is to change the resource pool settings as a part of the job that runs the report, and change them back when the report is complete. So picture a SQL Server Agent job with three steps:

1. Alter the resource pool, increasing the minimum on memory.
2. Run the report.
3. Revert the resource pool to its previous settings.

Doing this will ensure that the job has the required memory to complete the report on time, even if other requests come into the server after the report has started. As long as the report is truly memory-bound, the constraints on the resource pool will prevent any other queries from obtaining any memory until the constraints are removed.

After you have a classifier function enabled, which routes a specific login to the appropriate workload group and ultimately resource pool, you can create a job that has the three steps mentioned above.

If the report needs more than just memory to execute faster, you could also use steps 1 and 3 to change CPU settings for the pool, settings for the Group\_Reporting workload group, and even set tighter constraints on the other workload groups and resource pools in use.

### Limiting Parallelism for Intensive Background Jobs

You might be performing very intensive background tasks, such as index maintenance, and these can interfere with your user-facing processes, sometimes challenging your ability to meet SLAs. Obviously it is not going to be acceptable to eliminate your background tasks, because many of these are necessary. However, it could be appropriate to prevent these jobs from monopolizing all of the CPU resources on the server, with the understanding that they may take longer. This can be valid, both when it is okay that the task takes slightly longer, and also in rare cases where using parallelism actually leads to worse performance. In these cases, you may want to enforce limits on max degree of parallelism (MAX\_DOP). First, you need to create a workload group that will be used for maintenance tasks that you identify:

CREATE WORKLOAD GROUP [Group\_BackgroundTasks]

WITH

(

 MAX\_DOP = 1 -- or 2, or 4, or 8, …

)

USING [default];

Now, because you can’t classify incoming requests based on the command (for example, BACKUP DATABASE or ALTER INDEX), because classification occurs before any command is actually issued, you need some other way to determine whether a new session should be placed in this workload group. Assuming these tasks are started by SQL Server Agent jobs, and they are T-SQL step types, your task is relatively easy: Simply identify the jobs that you want to govern, and have the classifier function detect those jobs accurately, using the APP\_NAME() system function. By reviewing **sys.dm\_exec\_sessions**, you can see that SQL Server captures such job step activity in the following format:

 TSQL JobStep (Job 0xCCFE9FC0BF42334CBCFA64BB5F3040BC : Step 1)

You can correlate this cryptic 0x binary value to the actual job in the **msdb.dbo.sysjobs** table in the following way:

SELECT job\_id, name

 FROM msdb.dbo.sysjobs

WHERE CONVERT(BINARY(16),job\_id) = 0xCCFE9FC0BF42334CBCFA64BB5F3040BC;

So now all you need is a set of strings for the classifier function to look for in APP\_NAME(), using this 0x identifier, to match the jobs you want to constrain by placing them in this workload group. You can create a table in **master** that holds these strings, populated in the following way:

USE [master];

GO

CREATE TABLE dbo.RG\_BackgroundJobs

(

 job\_id UNIQUEIDENTIFIER,

 name SYSNAME,

 match\_string NVARCHAR(256) PRIMARY KEY

);

GO

INSERT dbo.RG\_BackgroundJobs

(

 job\_id, name, match\_string

)

SELECT

 job\_id, name, N'TSQL JobStep (Job '

+ CONVERT(VARCHAR(36), CONVERT(BINARY(16), job\_id), 1) + '%'

FROM msdb.dbo.sysjobs

WHERE -- you need to filter here

;

SELECT job\_id, name, match\_string

 FROM [master].dbo.RG\_BackgroundJobs;

You should see a result like this, depending on how many jobs matched your filter criteria.

 **Figure 3:** Result of population of the dbo.RG\_JobsToGovern lookup table

A lookup table is useful here for two reasons. One is that performing all of these CONVERT operations during the frequent running of the classifier function can be costly; an index seek on a **varchar** column should be much more efficient. The other is that the table will allow you the greatest flexibility in pulling individual jobs in and out of the list, without having to modify the classifier function, or reconfigure the Resource Governor. The only limitation is that, due to the fact that the classifier function requires the WITH SCHEMABINDING option, the table must reside in the same database (**master**).

With a lookup table in place, creating the classifier function that will place the relevant jobs in the right workload group is straightforward. First, check to see whether APP\_NAME() indicates that it is a SQL Server Agent job running; if so, then see whether it is one of the jobs you want to govern:

USE [master];

GO

CREATE FUNCTION dbo.classifier\_BackgroundJobs()

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 DECLARE

 -- BEWARE OF SPOOFING:

 @app NVARCHAR(256) = APP\_NAME(),

 @group SYSNAME = N'default';

 IF @app LIKE N'TSQL JobStep%'

 BEGIN

 IF EXISTS

 (

 SELECT 1

 FROM dbo.RG\_BackgroundJobs

 WHERE @app LIKE match\_string

 )

 BEGIN

 SET @group = N'Group\_BackgroundTasks';

 END

 END

 RETURN (@group);

END

GO

ALTER RESOURCE GOVERNOR WITH

 (CLASSIFIER\_FUNCTION = dbo.classifier\_JobsToGovern);

GO

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

You can make this solution more flexible by adding additional workload groups and different job categories, so that different types of maintenance jobs are subject to different constraints. Another extension to this approach could be to add time frames to the criteria; for example, you could constrain index maintenance jobs during the week, but not on Sundays during your regular weekly maintenance window.

## Chargeback

*Chargeback* typically describes the scenario where a company or a division has invested in hardware and is trying to recoup some of those costs by charging other business units for their usage of that hardware. In some cases, such as the shared hosting scenario, there is a basic "rental" fee that helps offset the original and ongoing hardware costs, with some providers charging extra if, for example, you exceed your allotted disk space. In other cases, the hardware owners may decide to charge for actual resource usage, including CPU, memory, I/O, and network bandwidth.

Disk space and I/O are easy to measure with existing technology external to SQL Server, and most hosting facilities already have the ability to monitor network bandwidth; what is harder to do in the current multi-tenancy model is to charge more precise fees for actual CPU and memory usage. The Resource Governor can help out here, as monitoring CPU and memory utilization specifically can be fairly straightforward using the new DMVs, where otherwise you would have to use extensive SQL Trace or SQL Server Audit mechanisms.

In some chargeback scenarios, the intent is not only to monitor utilization, but also to go so far as to prevent resource usage beyond allotment. This enforcement can be handled through various external mechanisms, such as bandwidth throttling, disk quotas, and tools such as Windows® System Resource Manager and SoftNUMA controls. In the virtualization space there are various tools available as well, including System Center Virtual Machine Manager for the Hyper-V™ technology. However, because the Resource Governor does not effectively provide absolute hard caps on CPU or memory utilization, and because it does not apply to disk usage or network bandwidth, it is more useful in this scenario as a monitoring tool and billing facilitator, aiding in chargeback but not maintaining exclusive responsibility and not enforcing strict controls on resource usage.

In both the multi-tenancy model, and the case where multiple business units utilize the same database, each customer usually has a unique SQL Server Authentication or Windows Authentication account in order to connect to the server and run queries. This makes it easy to create a workload group for each customer, even if they all belong to a single resource pool, and then create a classifier function that identifies each customer by account name. After customers are classified, each customer’s activity is aggregated and/or uniquely represented in DMVs, Performance Monitor counters and SQL Trace events. You can likely see where this is going: If you create central tables with columns for the metrics you want to “charge back” to each customer, such as CPU time and memory usage, you can build a job that polls for this information and stores it in this central location, and then you can build a billing system off of that data.

Note that there are a few limitations that prevent the Resource Governor from providing a complete chargeback solution, at least in its current form. Some have already been mentioned; for example, it cannot track disk space usage, I/O, or network traffic, nor can it place true hard limits even on the metrics it does enable you to monitor. While it can certainly keep track of key metrics, there is no detailed auditing of the data, much of the cumulative data is not persisted between service restarts, and there is no built-in mechanism to prevent you from losing data in certain scenarios.

Because the possibilities for implementing such a feature are virtually limitless, an actual working demonstration will not be covered here. While you can roll your own DMV queries (like those described in the previous section on consolidation), if you are planning to implement a chargeback system, the data collector (management data warehouse) is probably the best way to collect, store, and report on this information. This is because most of your work is already done for you: you can build it out to collect data from DMVs, Performance Monitor counters, and SQL Trace events with built-in scheduling, archival, and windowing functionality. For more information about the data collector, see the following topic in SQL Server Books Online:

**Introducing the Data Collector**
<http://msdn.microsoft.com/en-us/library/bb677248.aspx>

Chargeback often ties into consolidated environments, and it will allow you to recover some or all of your hardware costs if you decide to go this route. Buying newer and more powerful hardware, and offsetting the costs in the right way, can be cost effective ways to reduce both SQL Server sprawl and data center footprint, allowing you to replace your aging servers and significantly improve manageability. The Resource Governor can help in this scenario, as long as you understand some its limitations (more details on Resource Governor limitations are captured in Appendix E).

Microsoft’s Buck Woody has written two excellent articles on chargeback; if you are planning to implement such a system internally or for your customers, you should read these articles as a part of your planning and information gathering:

**SQL Server Chargeback Strategies, Part 1**
<http://www.informit.com/guides/content.aspx?g=sqlserver&seqNum=311>

**SQL Server Chargeback Strategies, Part 2**
<http://www.informit.com/guides/content.aspx?g=sqlserver&seqNum=312>

# Best Practices

There are a number of items that you should keep in mind in order to derive the best possible benefits from the Resource Governor. In no certain order, here are several:

## Test the Dedicated Administrator Connection – Then Test It Again

You should always make sure that the Dedicated Administrator Connection (DAC) is enabled before associating a classifier function with the Resource Governor. Because the function has the potential to slow down or block all external activity coming into the instance, and because the DAC bypasses the function, it will be the easiest and fastest way to get into the server and either correct the classifier function or disassociate it from the Resource Governor altogether. The only other alternative would be to restart SQL Server in single user mode, and this is not always a desirable approach.

Note that DAC is not enabled by default in cluster installations. For more information about using and enabling the DAC, see the following article on MSDN®:

**Using a Dedicated Administrator Connection**
[http://msdn.microsoft.com/en-us/library/ms189595(SQL.100).aspx](http://msdn.microsoft.com/en-us/library/ms189595%28SQL.100%29.aspx)

## Optimize the Classifier Function

The classifier function has the potential to be the biggest bottleneck on your system – if you let it. Timeouts within this function can bring your instance to its knees, because none of your requests will get through to perform their work if they are timing out waiting for classification. Clearly it is not a place where you want to mistakenly deploy an infinite loop or a query that requires scanning a table with 500 million rows. Outside of the obvious things that you would avoid in any piece of code, there are a lot of specific things you can do to minimize the performance impact of the function. The most important is also the most basic: Keep it as simple as possible. Some others include the following.

### Use the PreConnect Trace Events

The two trace events associated with the classifier function, **PreConnect:Starting** and **PreConnect:Completed**, will allow you to measure how long the classifier function takes to run. You should benchmark these events when hitting each path in your classifier function, and make sure to test the performance of the function whenever you make any changes, so that you are not surprised by even the slightest change in response time. These events are also briefly explained in Appendix D.

### Test the Classifier Function Under Load

It may seem trivial, but just because your classifier function returns in a few milliseconds in your isolated tests, this does not necessarily mean it will perform as well under significant volume. You should aim to test classification with a large number of sessions connecting, establishing their workload group name, running a simple query, and then disconnecting. During this time, you can profile the classifier function activity using a server-side trace (as mentioned previously), and ensure that response times are relatively linear as volume increases. If this is not true, you may need to re-evaluate the function to avoid performance problems later on.

### Avoid Lookup Tables

In the best case, the classifier function will only have logic internal to itself to deal with, so no outside data access is required. Not only does this prevent costly read operations and concurrency and locking issues, it also avoids any complication with chaining ownership of objects across databases, because your lookup information is unlikely to live in the **master** database. If the lookup table you want to use is small, it is doubtful that this difference is of any importance; however, if the lookup table is expected to grow, it is definitely something you want to be careful about.

### Optimize Unavoidable Lookups

If you need to derive information from a local lookup table in order to make classification decisions, be sure to have adequate indexes on the table, and keep them maintained. If the data in these tables is not static, they can cause problems over time. Make sure these tables are included in any typical statistics update or index reorganize routines you are performing for other “more visible” tables.

### Use Local Variables Where Appropriate

Because it makes the code much easier to read and explain, many classifier function examples you see at user group presentations, and in code samples on the Internet, use methodology similar to this:

IF SUSER\_SNAME() = N'anna'

 RETURN(N'anna');

IF SUSER\_SNAME() = N'bedecs'

 RETURN(N'bedecs');

-- and sometimes many more, depending on the logic

In cases like this, it is better to store the result of a repeated function call or lookup in a local variable, and then reference the local variable repeatedly, instead of performing the slightly heavier work every time:

DECLARE @u NVARCHAR(256);

SELECT @u = SUSER\_SNAME();

 IF @u IN (N'anna', N'bedecs')

 RETURN(@u);

Arguably, this difference is minimal, especially with a low number of calls – but, as with any high-volume piece of code, every little bit helps.

### Avoid Expensive String Operations

Another technique you might be tempted to use, simply because it is convenient, is to classify users based on a pattern in their username string or another similar property. For example, at your company, you might place all outside contractors into a specific workload group, and identify them by parsing their individual user names, looking for SUSER\_SNAME() LIKE 'YourDomain\v-%'. The function will be more efficient if you check group membership, for example using IS\_MEMBER('YourDomain\contractors').

### Only Classify for User-Defined Workload Groups

Because all sessions not explicitly classified by the classifier function end up in the default workload group, there is very little value in explicitly defining logic to route sessions to that group. This just adds extra logic to the classifier function, making it more difficult to debug, and certainly not making it any faster.

## Test the Classifier Function – Then Test It Again

In addition to performance, you will also want to ensure that the classifier function will return predictable results in every scenario you are trying to capture. Before writing your classifier function, you should have a firm grasp on exactly what you want it to accomplish. Flow charts can be a very worthwhile starting point to map out which situations will yield which workload group name as the result.

The most direct way to test changes is to remove the classifier function from the Resource Governor configuration, change the function code, and then re-associate the function with the Resource Governor; then, in order to test your changes, you will have to log in to SQL Server using the different methods that you are utilizing for classification. Depending on how complex your classification logic has become, this can be quite cumbersome. Because the workload group is tied to the user for the entire session, you can’t log in as a sysadmin and then expect EXECUTE AS to allow you to run in the context of a different user that should end up in another workload group – you are still the sysadmin in the eyes of the Resource Governor.

Also, using this methodology, you run the risk of introducing bugs directly into your code, which can have a wide range of negative impacts – from improper classification of workloads to an unresponsive server.

In order to really test your changes without affecting real-time operations, you can test the classifier function independently, by creating a similar function that accepts inputs for the values that you typically obtain within the function itself. For example, if you have a classifier function that looks like this:

CREATE FUNCTION dbo.classifier()

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 RETURN

 (

 SELECT CASE

 WHEN SUSER\_SNAME() = 'ceo' THEN N'top'

 WHEN HOST\_NAME() LIKE 'dev-%' THEN

 CASE WHEN DATEPART(DW, CURRENT\_TIMESTAMP) = 1

 THEN N'dev-highimportance'

 ELSE N'dev-lowimportance'

 END

 END

 );

END

GO

You can create a mirror function, with most of the same properties and the exact same logic as the classifier function, with the minor difference that you are testing the input parameters instead of intrinsic and other functions:

CREATE FUNCTION dbo.classifier\_mirror

(

 @suser\_sname NVARCHAR(256),

 @host\_name VARCHAR(128),

 @day\_of\_week TINYINT

)

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 RETURN

 (

 SELECT CASE

 WHEN @suser\_sname = 'ceo' THEN N'top'

 WHEN @host\_name LIKE 'dev-%' THEN

 CASE WHEN @day\_of\_week = 1

 THEN N'dev-highimportance'

 ELSE N'dev-lowimportance'

 END

 END

 );

END

GO

This will allow you to test the classifier function logic (but not the Resource Governor limits) by simply calling this function with different parameters, without having to test from multiple machines, log in to SQL Server with different accounts, change the system clock on the server, or any other cumbersome workarounds you might need to cover all of your potential use cases. It will also allow you to troubleshoot any logic problems before they appear in actual Resource Governor usage, so you can keep the existing classifier function logic in place, and avoid causing any performance or functionality issues while testing changes. The maintenance of this extra function is clearly a trade-off, because it should be updated in parallel with any changes to the real classifier function, but it will be more valuable in the long run as your classification logic gets more complex.

In addition, it is preferential to use fewer branches, IN() lists, and CASE expressions, rather than a series of independent IF statements. Using independent IF statements that each assign a group name to a variable that you eventually return can lead to situations where certain requests fall through the cracks and are classified incorrectly. Using a CASE expression, ordering from most likely to least likely match (or based on priority of your matching rules), and then ending with the catch-all scenario (ELSE ‘default’), you can be sure that you won’t accidentally give the ad hoc workload full run of the instance.

Finally, remember that the return value of the function is case sensitive, if your instance is running a case-sensitive collation (or if you might ever deploy your code to such an instance). If you have a workload group that you name ‘my group’ and your function returns ‘My Group’, rather than being classified into the correct workload group, those requests will drop into the default workload group. Incorrect case issues can be hard to spot, so use a sensible naming scheme for your workload groups, and be consistent. Depending on the complexity of your classifier function, it may make sense to declare variables for your possible workload group names at the top of the function and treat them like constants; in the extreme case of dozens or more groups, you could pull the correct names from the **sys.resource\_governor\_workload\_groups** view. You should always code defensively in this case, and assume that at least one of your target instances will have a case-sensitive collation. And if you want to go a few more steps to force you into the habit, you could try developing on an instance that uses a case-sensitive collation – you will learn the importance of case pretty quickly.

## Do Not Let the Classifier Function Trust Everything

A few of the items that you can choose to use for classification (such as HOST\_NAME() and APP\_NAME()) are relatively easy to spoof. If your developers discover that you have given your CEO’s machine name better resource availability than their own, they may be tempted to spoof the host name they are passing from their own query tools or applications, so that they can get a better slice of the resources. The reverse can also happen; if you place all requests from a specific application into a workload group in order to severely limit the resources that that application can get, it would be trivial for someone using the application to change the value in the connection string to something you didn’t predict, so that they get thrown into the default pool instead. Depending on how your applications and network are set up, in some cases you may just have to resign yourself to the fact that this spoofing is unavoidable; but when you can, put multiple checks in place, with at least one of them being a setting they can’t mess around with, such as login information. Another thing to keep in mind is that the names of workload groups and resource pools are public (and are visible in certain error messages), so this means that you should not include any sensitive or telling information in their names. For example, “ceo\_machine\_gets\_min\_50\_percent\_of\_cpu” is not likely a good choice, because if a user notices this, they can easily take advantage of the CPU settings by attempting to spoof the CEO’s HOST\_NAME() when they connect to SQL Server.

## Place Reasonable Limits on the Default Pool and Group

The default resource pool and default workload group are your safety nets. With all the testing in the world, your classifier function could still let sessions fall through the cracks into the default workload group. If you leave too little resources for the default pool, because you thought your classifier function would successfully allocate all requests to your user-defined workload groups, you will be in for a big surprise when a session slips through your logic and into the resource-starved default workload group.

## Use Memory Constraints with Caution

Because the minimum memory allocated to a resource pool will not be easily returned to the system even if the pool has no more requests in it, this setting should be used only if deemed absolutely necessary, and only if you are sure that this resource pool will need to use that memory a large percentage of the time. Likewise, placing maximum constraints on a pool should be isolated to cases where there is a very good reason to do so. Giving too much memory to one resource pool will often lead to suboptimal plans and longer query durations in all of the other resource pools.

## Maintain Similar Test and Development Servers

In order to avoid being surprised when you move from one environment to another, you should – to the best of your ability – attempt to maintain test and development servers that match the production environment as closely as possible. This refers not only to Resource Governor settings but also to the number of CPUs and amount of memory allocated to the instance. If these conditions differ significantly between environments, your resource constraints that balanced workloads effectively on the test server may yield vastly different query plans and performance in the production environment.

## Test with Adequate Background Workloads

It is very difficult to determine what kind of effect your resource constraints might have on a system if you develop in a bubble. Because several Resource Governor settings do not apply when there is no resource contention, it can be hard to tell if a query is slow because it is being constrained, or if it is slow because of other problems; also, you really can’t tell how much the settings actually slowed the query down. So be sure to simulate realistic load on the server, by replaying traces from production, or using any of a number of stress testing tools against your application or directly against SQL Server, before deciding that the constraints you are placing on your workloads are adequate. Ideally you would be replaying realistic traffic loads against your test environment that accurately reflect the type of workloads that are typically running against your production servers, but this is not always practical or even possible. There are several tools out there that can help simulate fictitious load against your servers; here are a couple:

 **Adam Machanic’s SQLQueryStress**
 <http://www.datamanipulation.net/sqlquerystress/>

 **SQL Load Test**
 <http://sqlloadtest.codeplex.com/>

 **Quest Benchmark Factory**
 <http://www.quest.com/benchmark-factory/>

There are also several tools available out there that will simulate load against your database via your Web or Windows application(s), which may provide much more realistic tests. There are also many articles online which will walk you through setting up a load testing environment.

## Plan Ahead, if You Can

It is hard to take this advice seriously if you are new to the technology, but try to plan your resource pool and workload group layout, at least at a high level, as early as possible. Dropping workload groups or resource pools, or switching groups between pools, is easy if you have isolated access to the instance; on busy servers, however, there are obstacles. For example, if you try to drop a resource pool that has workload groups associated with it, you will get this error:

Msg 10916, Level 16, State 1, Line 1

Cannot drop resource pool '<pool>' because it contains workload group '<group>'. Drop or remove all workload groups using this resource pool before dropping it.

And although dropping resource pools or workload groups (or changing the pool a group is using) while relevant users are still connected will appear to work initially, when you run the ALTER RESOURCE GOVERNOR RECONFIGURE command, you will see the following error, as described previously:

Msg 10904, Level 16, State 2, Line 1

Resource governor configuration failed. There are active sessions in workload groups being dropped or moved to different resource pools. Disconnect all active sessions in the affected workload groups and try again.

To get around this, you have to do just what the error messages state – move or drop workload groups before dropping resource pools, and clear out any active sessions (or wait them out) before running the ALTER RESOURCE GOVERNOR RECONFIGURE command.

You should understand your workloads before starting out with configuring the Resource Governor. Think about each type of workload you have (OLTP, CRUD, reporting), and the parts of each workload that require CPU (calculations, compilations, looping) and memory (joins, sorting, logical reads). You may already know these things, or you may have to perform some research in order to discover them; in any case, this should give you some high-level view of each workload, whether it is CPU-bound or memory-bound, and how they might interact with each other and the constraints around them. If you blindly apply memory limits to a workload that is mainly CPU-bound, you will be left with a poor impression of the Resource Governor and how it works. Most importantly, don’t believe in the impossible – that you can satisfy hungry applications on limited hardware just by constraining resources. The truth is that there is always a trade-off. This technology can help you balance finite resources on an instance, but it will not make a blade server behave like a massive 64-way server with 2 terabytes of RAM.

## Always Be Monitoring, Evaluating, and Adapting

The characteristics of your workloads can change over time: Scheduled jobs can change, an office can open in a new time zone, servers can be migrated to new hardware, new reports can be created, new applications can be deployed, new service packs can be installed; the list goes on and on. Any of these things can cause a major change to the way your workloads perform and how they interact with Resource Governor constraints. Do not be tempted to treat the technology as a “set it and forget it” switch. You need to periodically assess the effectiveness of your constraint settings to ensure that they are doing what you expect. You will find more details about this in the Monitoring section.

## Keep Demos Simple

If you are planning to demonstrate the Resource Governor to co-workers, user groups and the like, the best way to illustrate the impact of constraints on CPU and memory, especially on today’s typically low-powered but multi-core laptop, are to set **instance affinity** to 1 and **min/max memory** to something stable for the instance. The main reason for this is that with multiple CPUs, it is impossible to dictate how two concurrent processes will land on the schedulers. In some cases users have been quite surprised to see both requests use a single CPU while the other CPU sits idle; and have a hard time explaining the behavior to their audience. Boris Baryshnikov has published two blog posts that go into much more detail about why it is difficult to show off the technology on multi-core machines, and how to ensure a predictable demonstration:

**Part 1: Anatomy of SQL Server 2008 Resource Governor CPU Demo**
<http://blogs.technet.com/sqlos/archive/2007/12/14/part-1-anatomy-of-sql-server-2008-resource-governor-cpu-demo.aspx>

**Part 2: Resource Governor CPU Demo on multiple CPUs**
<http://blogs.technet.com/sqlos/archive/2008/01/18/part-2-resource-governor-cpu-demo-on-multiple-cpus.aspx>

If you do anything more complex than this, you are likely to spend more time explaining the behaviors you see in Performance Monitor, than explaining the technology itself. For longer and deeper presentations, you will need to derive some ways to show more complex interactions of workloads and constraints using some of the techniques you will find in Appendix D.

# Conclusion

Enterprise applications require predictable performance while optimizing the use of available resources. They require ways to prioritize, monitor, and analyze workloads sharing many of the same underlying resources. The Resource Governor is a unique and valuable new feature that allows administrators to identify, separate and control the way workloads utilize resources on their SQL Server 2008 instances. The technology offers much finer controls than the limited set of instruments available in previous versions. As SQL Server builds on providing innovative new features and capabilities at low cost with easy manageability for mission-critical applications customers will benefit. This is one more example that allows efficient deployment of mission critical applications at scale.

**For more information:**

[http://msdn.microsoft.com/en-us/library/bb895232.aspx](http://msdn.microsoft.com/en-us/library/bb895232%28SQL.100%29.aspx): Introducing Resource Governor

[http://msdn.microsoft.com/en-us/library/bb964723.aspx](http://msdn.microsoft.com/en-us/library/bb964723%28SQL.100%29.aspx): Resource Governor How-to Topics

[http://msdn.microsoft.com/en-us/library/bb933866.aspx](http://msdn.microsoft.com/en-us/library/bb933866%28SQL.100%29.aspx): Managing SQL Server Workloads with Resource Governor

<http://msdn.microsoft.com/en-us/library/bb933944.aspx>: Resource Governor Workload Management Scenarios

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback.

# Appendix A – Resource Governor in Depth

It is important to recognize that the Resource Governor can manage the CPU and memory that has been allotted to the instance, not the total CPU and memory of the physical machine. Also note that Resource Governor memory management only covers so-called single page allocators. Single page is a dominant mechanism of memory allocation inside SQL Server, and it accounts for the majority of memory usage in most cases; however, some components also use allocations that are not tracked by SQL Server’s memory manager. Examples in SQL Server 2008 that are not managed by the Resource Governor are extended stored procedures, linked servers, COM objects, and certain SQLCLR objects (depending on garbage collection). In addition, the Resource Governor does not manage memory used by the buffer pool for data pages; these mechanisms remain the same as in SQL Server 2005. Future versions will likely expand this coverage to include some of these additional components and features. Finally, the Resource Governor cannot apply constraints to requests that are in a pre-emptive waiting state for other tasks, such as I/O or locks.

Another important concept to understand is that everything you do with the Resource Governor can be considered a trade-off. Limiting the memory for a request may make the query take longer by increasing CPU or I/O requirements, and likewise, limiting the CPU may force the query to request more memory. Because these trade-offs can make individual queries take longer, there is always a precarious balance between conserving resources in general and helping individual queries finish as quickly as possible.

## Resource Pools

You can consider a resource pool to be a “slice” of an instance’s CPU and memory. You might want a resource pool to corral all requests from a specific application or from the users in the engineering department. A resource pool has two configuration aspects: CPU and memory. You can set a MIN and MAX for each (without any customizations, these are 0 and 100 respectively). For CPU, this does not mean traditional raw CPU percentage like you see in Task Manager, but rather a per-scheduler percentage of work performed by active threads. Nor does it mean that you can use the Resource Governor to reserve specific CPUs or to segment them in any specific way; it treats the entire set the instance can see as a whole. In addition, it is a guideline only, with the intention of keeping an average at or near that level over longer periods of time (meaning is it not a hard cap, and brief bursts above the limit are possible). For memory, one of the most significant architectural changes made for the Resource Governor is that, rather than having memory brokers for the entire instance, memory brokers and two resource semaphores in each resource pool handle memory management.

By default there are two resource pools: default and user-defined. The internal pool cannot be altered in any way, and reserves what SQL Server needs for internal processes, making the rest available to the default and user-defined pools. You can define up to 18 user-defined pools, for a total of 20. For more information, including a syntax diagram, see the following SQL Server Books Online topic:

 **CREATE RESOURCE POOL (Transact-SQL)** <http://msdn.microsoft.com/en-us/library/bb895329.aspx>

When a pool is created, you can specify a name and the following four options:

* MIN\_CPU\_PERCENT

When you apply a minimum CPU setting for a pool, you are essentially saying, “When requests in this pool are running, and there is CPU contention from other pools, make sure they collectively get this percentage of overall scheduler time.” It is important to note that this does not guarantee a certain percentage of CPU bandwidth to any single request; the Resource Governor uses fair scheduling to make a best effort to balance the CPU across all of the active threads, according to this and other Resource Governor settings, and it only enforces them if there is CPU contention. MIN\_CPU\_PERCENT can be between 0 and 100, and the sum of MIN\_CPU\_PERCENT across all pools cannot exceed 100.

* MAX\_CPU\_PERCENT

When you apply a maximum CPU setting for a pool, you are saying, “When requests in this pool are running and there is CPU contention, make sure they get no more than this percentage of overall scheduler time.” Again, this is a best effort, and a soft cap, so you will occasionally see the maximums exceeded. In any pool, the value MAX\_CPU\_PERCENT must be between MIN\_CPU\_PERCENT and 100 (but this value cannot be 0).

* MIN\_MEMORY\_PERCENT

When you apply a minimum memory setting for a pool, you are saying, “Ensure that this percentage of memory will be available for any requests that might run in this resource pool.” This is an important differentiator compared to MIN\_CPU\_PERCENT, because in this case memory may remain in the given resource pool even when the pool does not have any requests in the workload groups belonging to this pool. Therefore it is crucial that you be very careful when using this setting, because this memory will be unavailable for use by any other pool, even when there are no active requests. Like with MIN\_CPU\_PERCENT, the sum of MIN\_MEMORY\_PERCENT across all pools cannot exceed 100, and the defined value must be between 0 and 100.

* MAX\_MEMORY\_PERCENT

When you apply a maximum memory setting for a pool, you are saying, “When requests are running in this pool, make sure they get no more than this percentage of overall memory.” In any pool, MAX\_MEMORY\_PERCENT must be greater than or equal to MIN\_MEMORY\_PERCENT, and the value must fall between 1 and 100. Note that the memory referenced here is query execution grant memory, not buffer pool memory (for example, data and index pages).

It can be complex to think about how these four settings across multiple resource pools will interact with each other, and how that will affect your workloads. In a nutshell, at least under heavy load, the effective maximum percentage of a resource that a specific pool can consume is the maximum for that pool less the sum of the minimum resources defined for all of the other pools, and the shared percentage is calculated as the effective maximum less the minimum. So assuming you have four resource pools with the following settings, we can calculate the effective maximum percentage (best case for that pool) and shared percentage (best case for all pools). Note that neither the effective maximum percentage nor the shared percentage will always represent what will actually happen under normal operation; also, the sums of either calculation will not necessarily add up to 100 percent.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pool | Min % | Max % | Effective Max % | Shared % |
| default | 0 | 100 |  25 (MIN (100, 100-(20+50+5))) |  25 (25-0) |
| Pool\_A | 20 | 100 |  45 (MIN (100, 100-(50+5)) |  25 (45-20) |
| Pool\_B | 50 | 70 |  70 (MIN (70, 100-(20+5)) |  20 (70-50) |
| Pool\_C | 5 | 100 |  30 (MIN (100, 100-(20+50))) |  25 (30-5) |

**Table 2:** Effective maximum and shared percent for CPU or memory settings

Note that when the server is not under load, it is always possible that requests in a specific resource pool can consume all of the resources on the server, even if other resource pools have established minimums. And it is also likely that, when the server is under heavy load, the set of requests in a specific resource pool could never reach their effective maximums. Finally, one of the reasons you will typically want to avoid creating too many resource pools is that your cache size can increase dramatically if the same database is shared between multiple resource pools.

For more information about calculating effective and shared maximums for different pools, see the following SQL Server Books Online topic:

**Resource Governor Concepts**<http://msdn.microsoft.com/en-us/library/bb934084.aspx>

## Workload Groups

You can think about a workload group as a container or bucket for a set of similar queries or tasks, and the purpose of the group is to route requests in the group to a specific resource pool. The separation is there because, while with a resource pool you can control CPU and memory, adding a layer for workload groups allows you to add additional controls for different sets of requests across the same “slice” of the instance’s resources. For example, you might have a resource pool for the engineering department, but you might want to create a separate workload group for testers or quality assurance folks so they don’t inadvertently take too many resources, or for your boss so that his queries finish faster than yours (or vice versa).

Like resource pools, there are two predefined workload groups: default and internal. Each workload group is assigned to exactly one resource pool, though you can have several groups associated with one pool; if not specified, the workload group will be allocated to the default resource pool. You can have an unlimited number of workload groups, but this will make administration quite complex; typically you would have no more than double the number of pools (and often less than that). Note that you cannot assign user requests to the internal workload group, nor can you modify its settings; also, you cannot associate the default workload group with the internal or any user-defined resource pool. For more information, including a syntax diagram, see the following SQL Server Books Online topic:

**CREATE WORKLOAD GROUP (Transact-SQL)**<http://msdn.microsoft.com/en-us/library/bb934146.aspx>

In addition to the group name and the pool\_name that the workload group should be associated with, the following settings are available for configuration:

* IMPORTANCE

While it may sound like priority, IMPORTANCE is not really priority in the traditional sense. It is simply relative importance, and it allows the scheduler to balance work across the scheduler for the individual requests in the workload group. Note that this setting is effective within all of the workload groups in a specific resource pool; however, importance is not compared between pools. The possible values for IMPORTANCE are LOW, MEDIUM, and HIGH. The default value is MEDIUM if not otherwise specified.

To understand how the scheduler balances tasks with different importance, consider that a numeric coefficient is applied to each importance, where LOW is 1, MEDIUM is 3, and HIGH is 9. Now imagine that two requests come in, one LOW and one HIGH. What will happen is that if the scheduler has 10 quantum units available to queue, it will give 9 of them to the HIGH task, and 1 of them to the LOW task. Note that they can run concurrently given available schedulers, so it is not technically a queue, but rather what Microsoft’s engineers loosely refer to as “quantum sharing.” For a much deeper treatment of this queuing mechanism and quantum sharing, see Bob Dorr’s explanation in the following blog post:

 **SQL Server 2008 - Resource Governor Questions**
 <http://blogs.msdn.com/psssql/archive/2008/01/10/sql-server-2008-resource-governor-questions.aspx>

* MAX\_DOP

The MAX\_DOP setting corresponds to the maximum degree of parallelism allowed for any query in the group. The default is 0, which translates to allowing global settings (**sp\_configure**) to dictate whether parallelism can be used. When this value is non-zero, the scheduler will prevent a parallel plan from exceeding this number of parallel threads when it queues a request. This is similar, but not identical, to how it would work if OPTION (MAXDOP n) were applied as a query hint. The major difference is that in the case of a query that is limited via the MAX\_DOP setting of the workload group, the graphical execution showplan will still show parallelism, so it will appear as if the constraint is not working. This situation will be treated in Appendix D, “Monitoring the Resource Governor.”

* REQUEST\_MAX\_MEMORY\_GRANT\_PERCENT

REQUEST\_MAX\_MEMORY\_GRANT\_PERCENT is the portion of the total memory available to the underlying resource pool that can be granted to any query in the workload group. The default, if not specified, is 25. If a query requires more memory than this setting dictates, then for the default workload group, the server will grant the memory required by the query, as long as it is available in the pool and not reserved by other pools; otherwise, a timeout error will eventually occur:

Msg 8645

A timeout occurred while waiting for memory resources to execute the query in resource pool '%s' (%d). Rerun the query.

For user-defined pools, the server will resort to lowering the degree of parallelism until the memory limit is satisfied. If the requirement is still too high, or parallelism is already 1, then the following error will occur:

Msg 8657

Could not get the memory grant of %d KB because it exceeds the maximum configuration limit in workload group '%s' (%d) and resource pool '%s' (%d). Contact the server administrator to increase the memory usage limit.

Increasing the value of this parameter allows a single query to consume larger amounts of memory, and reducing it allows for higher concurrency of large, memory-bound queries, although they may run more slowly as a result. Under the default setting (25%), and with no other memory consumption occurring, you can expect to run up to three large memory queries simultaneously. You might expect this number to be four, but there is overhead due to hidden internal reservation, which does not allow the sum of query memory consumption to reach exactly 100 percent. On SQL Server 2005, this threshold was not controllable by the user, and it was approximately equal to 20 percent per request, which allowed about four large queries to be running simultaneously. It is not recommended that you set the REQUEST\_MAX\_MEMORY\_GRANT\_PERCENT value above 65-70 percent, because the server may be unable to set aside enough free memory if other concurrent queries are running. This may eventually lead to the query time-out error 8645, mentioned above.

* REQUEST\_MEMORY\_GRANT\_TIMEOUT\_SEC

The REQUEST\_MEMORY\_GRANT\_TIMEOUT\_SEC setting allows you to specify the number of seconds that a query can wait for a memory grant. If the server is unable to satisfy the requirement within the defined duration, and there is little memory pressure, the memory granted to the query will be lower than what was requested, yielding lower performance. However, if there is sufficient memory pressure within the pool, a timeout error will occur:

Msg 8651

Could not perform the operation because the requested memory grant was not available in resource pool '%s' (%d). Rerun the query, reduce the query load, or check resource governor configuration setting.

The default value is 0, which means that the SQL Server 2005 logic applies (the wait time will be determined based on the cost of the query, and it will not exceed 24 hours).

* GROUP\_MAX\_REQUESTS

The GROUP\_MAX\_REQUESTS setting lets you dictate how many concurrent requests can be running at any time within the workload group. Unlike with IMPORTANCE, in this case, the scheduler does treat the set of active requests similar to a queue, where no work is assigned to tasks that came in after the limit was reached by existing requests. Note that if it determines that doing so could avoid blocking or a deadlock scenario, the Database Engine may decide to ignore this limit temporarily. The default value is 0, which means no limit.

* REQUEST\_MAX\_CPU\_TIME\_SEC

The REQUEST\_MAX\_CPU\_TIME\_SEC setting indicates how much CPU time any single query within the group can take before a CPU Threshold Exceeded event is raised. Unlike the query governor cost limit option, which refuses to run a query at all if the plan estimates that it will exceed the defined limit, this setting does not actually prevent a query from continuing execution and exceeding the time specified. Instead, the trace event is raised the first time a query is observed to exceed the limit. The default value for the setting is 0, which means there is no limit and no event will ever be raised. Keep in mind that there is a polling mechanism used to check violations, and this runs about every 5 seconds. So, there can be cases where you set a maximum value to 5 seconds, a query runs for 7 seconds, and no alert is ever fired, simply because the polling interval did not overlap with the violation. This does not mean you should avoid setting the limit lower to 5 seconds; you can set it to one second, and still catch queries that run for 1.0001 seconds.

## Classifier Function

This function is effectively the glue that holds together the binding of user requests to workload groups (and, in turn, resource pools). It is a user-defined function that lives in the **master** database, and it contains logic that returns the name of a workload group. The function runs after authentication and logon triggers have fired, and the workload group name is used to instruct the Resource Governor on how this session should be tracked throughout its lifetime. There are several ways you can determine which workload group a request should belong to; you can segment requests by database, user, Windows group, server role, application name, host name, time of day, day of the week or month, or your own custom classification rules. You can achieve this by inspecting the values in built-in functions such as LOGIN\_NAME(), HOST\_NAME(), APP\_NAME(), CONNECTIONPROPERTY(), LOGINPROPERTY(), and IS\_MEMBER():intrinsic information such as DATEPART() and CURRENT\_TIMESTAMP; and finally information you store in your own user tables.

You should note that your goals for the classifier function are that it be accurate and, equally as important, that it be as efficient as possible. This function has the potential to be the biggest bottleneck on your system, because it is executed for every new connection to the server. As such, adding even milliseconds to its execution time can have drastic effects for systems with high connection rates.

One important fact to note here is that, in a case-sensitive collation, the return value of the classifier function is case-sensitive according to the Resource Governor; so, take care to use the correct case whenever you reference Resource Governor entity names. As a best practice, and not just in the case of the return value of the classifier function, you should always use consistent case (the predefined entities are always in lowercase). Also remember that the classifier function should always return SYSNAME (NVARCHAR(128)).

Creating the function should be in the form:

USE [master];

GO

CREATE FUNCTION dbo.fn\_classifier()

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 RETURN (SELECT N'case\_sensitive\_group\_name');

END

GO

And to tell the Resource Governor that this is the function you want to use, you need to specify the name in the options, and then use ALTER RESOURCE GOVERNOR RECONFIGURE to apply the new setting:

ALTER RESOURCE GOVERNOR WITH

 (CLASSIFIER\_FUNCTION = dbo.fn\_classifier);

GO

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

In addition to case sensitivity, there are some other rules that should be kept in mind when creating the classifier function. You can’t assign requests to the internal group; returning ‘internal’ or any nonexistent group name will end up using the default workload group. You also cannot access external or pre-emptive operations such as extended procedures or WAITFOR. The function also must be created WITH SCHEMABINDING, which means you cannot easily change underlying objects after the function is in place. And you cannot change the function while it is configured for use with the Resource Governor; you must change the configuration to use a different function (or NULL, which means that all new connections should be placed in the default workload group) in order to apply changes.

Msg 10920, Level 16, State 2, Procedure <name>, Line 6

Cannot alter user-defined function '<name>'. It is being used as a resource governor classifier.

While the function can access tables, views, and even DMVs, it cannot use four-part names, so if you plan to use a central server for lookup data used by the function, you will need to investigate replicating it in some way to every server that will use it for classification decisions. (This has more to do with the SCHEMABINDING requirement than with the classifier function specifically.) If you try to access a linked server object from the classifier function, you will get a compile-time error:

Msg 2014, Level 16, State 1, Procedure <name>, Line <n>

Remote access is not allowed from within a schema-bound object.

And no, you cannot fool the system with synonyms, because they are not allowed in schema-bound objects either:

Msg 2788, Level 16, State 1, Procedure <name>, Line <n>

Synonyms are invalid in a schemabound object or a constraint expression.

You can keep multiple functions around in **master**; however, only one can serve as the classifier function for the Resource Governor at any point in time. The function works similar to a logon trigger, in that it is automatically invoked whenever a new session is established (right after any logon triggers are fired, and before the debugging session is enabled). The classification that is returned for a specific session at this time is fixed for the life of a session; you cannot reclassify a session after it has started (for example by running ALTER FUNCTION and changing the workload group logic), even though you can affect its resource usage dynamically by applying changes to workload group and/or resource pool settings.

When the Resource Governor is enabled, all user requests must go through classification when their session starts, unless there is no classifier function associated – in which case, all user requests automatically fall into the default workload group. Such sessions are still subject to any constraints you place on that group or the default resource pool. The only exceptions where a request can bypass the classifier function are when using the dedicated administrator connection (DAC) and when starting SQL Server in single user mode (-m -f). So, it is a good idea to thoroughly test your classifier function for both logic and performance before hooking it up with the Resource Governor configuration; and also, to be sure that the DAC is enabled and working correctly. Otherwise, if you have a slow function, it can cause login timeouts, and if the function actually hangs your server, and you can’t troubleshoot it using the DAC—you will need to bring the service down and back up in single user mode in order to connect again. These and other issues with the classifier function will be treated in the “Best Practices” and “Common Pitfalls” sections.

## Catalog Views

There are three new catalog views containing the configuration settings: one for the Resource Governor in general, one for workload groups, and one for resource pools. The catalog views show the configuration metadata about the Resource Governor, but not necessarily the runtime metadata. This will be explained in further detail in the next section. For more information about the columns and data types, see the following SQL Server Books Online topic:

**Resource Governor Catalog Views (Transact-SQL)**
<http://msdn.microsoft.com/en-us/library/bb934093.aspx>

## Dynamic Management Views

Like the catalog views, there are three similar Dynamic Management Views (DMVs) that have been added for tracking Resource Governor usage. For the columns that are the same as those in the catalog view, instead of configuration values, they hold run-time values. The way this works is that you can change a setting, and it only alters the value in the catalog view, until you issue an ALTER RESOURCE GOVERNOR RECONFIGURE command, at which point the corresponding value in the DMV is updated as well. The additional columns contain statistics for the workload group or resource pool or, in the case of the configuration view, whether there are configuration changes waiting that have not been applied with ALTER RESOURCE GOVERNOR RECONFIGURE.

Note that the RECONFIGURE command used to apply settings of **sp\_configure** is not the same as ALTER RESOURCE GOVERNOR RECONFIGURE; these are separate, independent commands.

You can use the Resource Governor DMVs to see aggregate statistics for each resource pool and workload group over time. If you make a change to the configuration and then want to start from a clean baseline, you can run a reset command to clear several of the aggregates that have been collected and reset them to zero. Resetting the statistics between runs will allow you to tweak configuration settings optimally through many short cycles. The command is:

ALTER RESOURCE GOVERNOR RESET STATISTICS;

Currently, this action will reset the **statistics\_start\_time** value to CURRENT\_TIMESTAMP for every resource pool and workload group. The fact that this column exists in the individual DMVs, rather than as a single data point in the configuration view, suggests that a future version might support an extension to the above command that will allow you to pick and choose which specific statistics to reset (so you could test a configuration change to a workload group, for example, without destroying your statistics for other entities). However, be aware that not all statistics can be reset; for example, any column that reports current activity will still report the truth, such as **active\_request\_count**. Also, several of the core metrics that reflect targets or effective maximums will not be reset, because these are a reflection of configuration settings and server state, not of individual query activity. For more information about t columns and data types of these new DMVs, see the following SQL Server Books Online topic:

**Resource Governor Related Dynamic Management Views (Transact-SQL)**<http://msdn.microsoft.com/en-us/library/bb934218.aspx>

In addition to these new views, several of the DMVs first introduced in SQL Server 2005 have been updated to include information specific to the Resource Governor. These new entities will be covered Appendix D.

## How it All Works Together

It would be useful at this point to discuss again the lifecycle of a request in terms of the Resource Governor. Essentially, the classifier function assigns an incoming session to a specific workload group, which in turn corresponds to exactly one resource pool. Once classified, the connection is bound to that workload group and resource pool for the life of the session (in other words, no reclassification occurs). The Resource Governor now knows exactly how to apply constraints (depending on current Resource Governor settings and the state of the server) before it passes each request for that session onto the query optimizer.

Note that if you do not enable a classifier function, the engine still uses the technology, but the behavior is much the same as it is in SQL Server 2005: all non-internal requests go into a single default workload group and resource pool, and share the same set of instance resources. And while you may not be explicitly classifying individual requests, you can still change the constraints on the default workload group and resource pool, which will affect all user requests. For example, you may still want to capture CPU Threshold Exceeded events for any query running longer than a minute, or cap all user requests at a maximum of 10 percent of available memory.

As mentioned previously, while the workload group and resource pool to which a session belongs cannot be changed for the life of the session, some of the settings can be affected in real time. In general, changes to CPU settings can take effect whenever the scheduler makes its next decision (in current versions of SQL Server, this is about every 4 milliseconds), or when the next request is sent. Memory changes will not affect the memory requested by tasks already in progress (even if the memory has not yet been granted), but will take effect whenever a new memory request is made. Because classification only occurs at the beginning of a session, changes to the classifier function obviously cannot take effect until a new session is created, and no existing sessions will be affected. Changes to group pool assignments are not permitted while there are any active sessions in the affected group; this can make dynamic pool assignment challenging on busy systems. You can make the changes to the configuration, but when you try to reconfigure, active sessions affected will cause the following error:

Msg 10904, Level 16, State 2, Line 1

Resource governor configuration failed. There are active sessions in workload groups being dropped or moved to different resource pools. Disconnect all active sessions in the affected workload groups and try again.

In this case, there is not much you can do other than kill the problem sessions, wait them out, or schedule the assignment change for non-peak hours.

# Appendix B – Resource Governor Overhead

The Resource Governor infrastructure itself does not add significant overhead (in most systems and configurations this will be under 1 percent). Altering the Resource Governor controls will have an effect on individual workloads, but this is not the overhead of the feature. However, you should keep the following in mind when configuring the Resource Governor.

### Dynamic Management Views

Maintaining the aggregate statistics for each workload group and resource pool in the dynamic management views is quite likely the biggest inherent overhead of the Resource Governor. Resource pools are limited to 20, so this overhead is relatively insignificant; and while workload groups can theoretically reach a much higher number, Microsoft’s own internal testing revealed no performance degradation in DMV overhead with over a thousand groups. In any case, if you are creating more than a handful of groups, you should probably consider re-evaluating whether you are making the best use of Resource Governor in the first place.

### Classifier Function

While not inherently a performance problem, the classifier function certainly has the potential to become a major issue if you are not careful. This is primarily because every new connection will need to go through the function; so, if your SQL Server instance has a high rate of short connect/disconnect transactions, the overhead will correspond to the volume and the complexity of the function. This is an important consideration when using connection pooling or other technologies where the rate of disconnect/reconnect is high, as each reconnect will once again invoke the classifier function. There are several ways to mitigate the performance costs associated with the classifier function, and these are treated in the main content of this paper. Note, however, that if the application keeps the connection open and just submits more queries on the same connection, the classifier function is not executed for each request.

### Tracing and Eventing

Depending on how many traces you are running, and how many events are being generated, you can expect some overhead due to this activity. If your system is capturing and/or triggering a lot of events that are handled by Extended Events, SQL Trace, or even Profiler, these activities will obviously require some of the resources from the instance. This is no different from any other feature using the events, but it is important to keep in mind when using the Resource Governor, as resources used by the internal resource pool will impact the amount available for enforcement.

# Appendix C – Resource Governor Permissions

Much of the permission scheme required to manage and monitor Resource Governor falls into existing permission schemes at the server level. To configure Resource Governor settings, the user must have CONTROL SERVER permissions; to monitor Resource Governor activity through the DMVs, the user must have VIEW SERVER STATE permissions.

One exception is the classifier function; this object is subject to the same permissions and ownership rules as other user-defined objects. To modify the object, the user needs GRANT ALTER permissions; and to view the definition requires GRANT VIEW DEFINITION permissions. Logically it should follow that, in order to execute the function, the user would need to have GRANT EXECUTE permissions; however, this is not true. The system ensures that, even if you have been explicitly denied EXECUTE permissions on the classifier function, the permissions check on the function itself is bypassed, allowing classification to proceed. (However, the function still has user context, and it does not allow for permissions elevation. In other words, the user will not get magic access to restricted objects because of a bug or loophole in the classifier function.)

# Appendix D – Monitoring the Resource Governor

The Resource Governor provides many ways for you to monitor the performance and run-time aspects of your workload groups and resource pools, whether or not you are actually employing any constraints. Simply by classifying incoming requests into buckets segregated by whatever means you choose, you automatically get the ability to peek at what each set of requests is doing in the system. Even if you are not classifying incoming requests, you can get an aggregate view of your entire workload.

You can do this through the DMVs, Performance Monitor, SQL Trace, and even the new Extended Events infrastructure. Using these tools, you can gauge the effectiveness of your Resource Governor configuration, or simply gather information about your workloads so that you can understand them better.

## Dynamic Management Views

Looking at the main Resource Governor DMVs, **sys.dm\_resource\_governor\_workload\_groups** and **sys.dm\_resource\_governor\_resource\_pools**, you can get a very good feel for how each pool or group is utilizing resources, and gain insight into what you can do to better balance the usage across your system. Along with a general feel for the number of requests and total amount of memory and CPU time being used by each group or pool, you can learn facts about your workloads that can lead to corrective action. Some examples include the following.

### sys.dm\_resource\_governor\_resource\_pools

* The columns **total\_cpu\_usage\_ms** and **used\_memory\_kb** can give you a quick clue as to how your each of your resource pools are consuming these two resources relative to the others, allowing you to focus on performance problems in a particular pool.
* A high value in the **out\_of\_memory\_count** column can tell you that queries are failing due to memory constraints.
* The **memgrant\_waiter\_count** column can indicate that the pool might benefit from an increase in memory caps, so that queries can get grants quicker (or that you should investigate queries that may be poorly formed and require too much memory).
* The **total\_memgrant\_timeout\_count** column can indicate that many queries are timing out waiting for memory grants, which may indicate that memory constraints on this pool are too low (or the minimums on other pools are too high).

### sys.dm\_resource\_governor\_workload\_groups

* Unexpected values in the **queued\_request\_count** or **total\_queued\_request\_count** columns indicate that the restriction on GROUP\_MAX\_REQUESTS may be too low.
* A high number in the **total\_suboptimal\_plan\_generation\_count** column means that a lot of “bad” query plans are being forced due to the group’s constraints; this group may be a candidate for re-evaluating those constraints, or analyzing individual queries.
* High values in the **total\_query\_optimization\_count** table might indicate memory pressure.
* The **active\_parallel\_thread\_count** column can tell you whether queries in your workload group are actively using parallelism.
* A high value in the **total\_cpu\_limit\_violation\_count** can point out that queries are taking longer to run than your threshold, which may mean that your threshold is too low; however, note that elapsed time does not necessarily mean that the query was monopolizing CPU.

In addition to the new Resource Governor DMVs, several of the existing DMVs have been altered to include information about resource pools and workload groups:

* **sys.dm\_exec\_sessions**

Using this DMV, you can break out sessions by **group\_id** to see memory usage, CPU usage, **scheduled\_time**, reads, logical reads, and writes, either individually or aggregated to the group.

* **sys.dm\_exec\_requests**

Similar to **sys.dm\_exec\_sessions**, for active requests you can get **cpu\_time**, **total\_elapsed\_time**, **wait\_time**, wait types and other metrics using the **group\_id** column to see data either for each request or for the group as a whole. High CXPacket waits across the group, for example, can indicate that a lower MAX\_DOP setting for the group may be appropriate.

* **sys.dm\_exec\_cached\_plans**

With this DMV, you can see cached plans per resource pool, using the **pool\_id** column, for compiled plans (procedures, ad hoc, and prepared) as well as parse trees for views. Low (or zero) rowcounts for a particular pool can help you identify workloads that are not able to cache query plans effectively. Note that DBCC FREEPROCCACHE now takes optional parameters allowing you to clear plan cache for a specific resource pool. For more information ,see the following Books Online topic:

**DBCC FREEPROCCACHE**[http://msdn.microsoft.com/en-us/library/ms174283(SQL.100).aspx](http://msdn.microsoft.com/en-us/library/ms174283.aspx)

* **sys.dm\_exec\_query\_memory\_grants**

Using this DMV you can see detailed memory grant information for each active request, including how long a grant took (difference between **request\_time** and **grant\_time**), how much memory was granted compared to requested (**granted\_memory\_kb**, **requested\_memory\_kb**), and what Resource Governor considers to be the “best” amount of memory for the request (**ideal\_memory\_kb**). You can aggregate the data in the DMV by **pool\_id** or by **group\_id**.

* **sys.dm\_exec\_query\_resource\_semaphores**

For each pool, you can see memory details for each of the pool’s two resource semaphores, including the target, total, available, granted, and used memory, the number of requests that have been granted memory, the number of requests waiting on a grant, the number of timeout errors, and the number of forced memory grants. In short, this can tell you whether your resource pools are able to acquire enough memory.

* **sys.dm\_os\_memory\_brokers**

Here you can see allocations, allocation rate, limits, and last notification status for cache, steal, and reserve for each pool. This information can provide a detailed breakdown of how the memory usage of each pool is changing over time.

* **sys.dm\_os\_memory\_cache\_entries**

This DMV provides very low-level details about memory allocated and distributed amongst the different pools. In addition to buffer pool memory, you can also see stats about memory usage within each pool for non-buffer pool objects like extended procedures.

Finally, in addition to querying against the DMVs in real time, you can take advantage of another SQL Server 2008 feature, the data collector (management data warehouse), to maintain a running history of Resource Governor metrics. By archiving key DMV data periodically, your statistics can survive manual reset events, as well as service restarts or failovers. While setting up a custom collection set is beyond the scope of this paper, you can see this topic in SQL Server Books Online to get started:

**How to: Create a Custom Collection Set That Uses the Generic T-SQL Query Collector Type**
<http://msdn.microsoft.com/en-us/library/bb677277.aspx>

## Performance Monitor

SQL Server 2008 installs two new sets of Performance Monitor counters to enable monitoring of Resource Governor entities. They are SQL Server: Resource Pool Stats and SQL Server: Workload Group Stats, and they are also available in the DMV **sys.dm\_os\_performance\_counters**. The individual counters in each object are described in the following SQL Server Books Online topics:

**SQL Server, Resource Pool Stats Object**<http://msdn.microsoft.com/en-us/library/cc645958.aspx>

**SQL Server, Workload Group Stats Object**<http://msdn.microsoft.com/en-us/library/cc627354.aspx>

### Using Performance Monitor

These counters are segregated by SQL Server instance in Performance Monitor and in the DMV, so even if you have multiple instances on the same machine, and they share the same names for resource pools or workload groups, you will still be able to tell them apart. When you add them through the Performance Monitor user interface, you will be able to pick the counters from each SQL Server instance separately (not to be confused with the counter instance, which actually reflects the name of the workload group or resource pool). In the following screen shot, you can see that the workload group “foo” exists on both the SQL2008 instance and the SQL2008A instance.

 **Figure 4:** Adding Workload Group counters using Performance Monitor

You may have noticed that most of the information available through these new Performance Monitor counters seems to overlap with the data available from the Resource Governor DMVs. While this is true in general, there are some exceptions. For example, the resource pool counter for CPU control effect % will tell you how much impact a CPU constraint has on the pool as a whole, when compared to how the pool would be utilizing CPU without any constraints. It can be interesting to correlate this information with any observable increases in query times to see if your constraints are helping or hurting performance overall.

The advantage of using the Performance Monitor user interface when analyzing your workloads (and how the Resource Governor impacts them) is that, unlike with the DMVs, you get visual graphs that clearly show deltas. In the following screen shot, you can see the CPU usage for two concurrent CPU-intensive queries in two different workload groups in the same resource pool, with no constraints (Group1 in red, and Group2 in blue). Up until point (A), you can see that each task is using about the same amount of CPU. However at point (A) the relative importance within the pool was set to IMPORTANCE = LOW for Group1 and IMPORTANCE = HIGH for Group2. You can see at that point that Group2 was able to use a lot more of the CPU (nearly half) and that Group1 was limited to roughly 20%. At point (B) both groups were set back to equal importance (MEDIUM) and, as expected, the CPU usage became more balanced.


 **Figure 5:** Illustrating Resource Governor changes using Performance Monitor

While you can get the same information from the DMVs, you would have to poll them continuously, and it would be difficult to pinpoint the exact moment when a metric changed significantly. Using the management data warehouse with a very high polling frequency, and then querying against the data after the fact, would be the only feasible way to capture changes like this other than real-time monitoring via Performance Monitor.

## Showplan and max degree of parallelism

When you prevent a workload group from running tasks in parallel using the MAX\_DOP = 1 setting, you may want to validate this by running a query and examining the query plan. You will surely be surprised to find that the graphical execution showplan will still display parallel operators. If you look closer, the showplan XML (actual, not estimated) will have <QueryPlan DegreeOfParallelism="0" ... />, and only one thread listed inside of the RunTimeInformation node:

<RunTimeInformation>

 <RunTimeCountersPerThread Thread="0" …

</RunTimeInformation>

Whereas a query that actually uses parallelism will have <QueryPlan DegreeOfParallelism="2" ... /> and show multiple threads:

<RunTimeInformation>

 <RunTimeCountersPerThread Thread="2" …

 <RunTimeCountersPerThread Thread="1" …

 <RunTimeCountersPerThread Thread="0" …

</RunTimeInformation>

If the **max degree of parallelism** setting for **sp\_configure** is any number other than 1, SQL Server will always consider parallel plans for queries. If parallelism is limited to 1 by a Resource Governor setting, these plans will run serially, and even though the query plans will still show parallelism operators, you can confirm that the setting is in effect as described above. If the **max degree of parallelism** setting for **sp\_configure** is 1, SQL Server will not attempt to generate parallel plans, and they will not contain parallelism operators, unless this setting is overridden by a MAXDOP query hint or a Workload Group MAX\_DOP setting greater than 1 (as described in the documentation below). In the case where there is no query hint and the Resource Governor MAX\_DOP setting is 1, these queries will run serially as long as the serial plan is in the plan cache, even if the max degree of **parallelism** and/or Resource Governor settings are changed later. When the plan is next recompiled, SQL Server will use the combination of **sp\_configure** and Resource Governor settings to determine whether a parallel plan could be used. For more information about how Resource Governor settings interact with **sp\_configure** and the MAXDOP query hint, see the notes in the following SQL Server Books Online topic:

**CREATE WORKLOAD GROUP**
<http://msdn.microsoft.com/en-us/library/bb934146.aspx>

There are other ways you can validate whether a query is running with parallel threads; for example, if the query is long-running, you should see periodic **wait\_type** values of CXPACKET in the DMV **sys.dm\_exec\_requests**, while you will not see this wait type if the query is bound to a single CPU. You can also see that there are parallel tasks executing in the group by looking at the **active\_parallel\_thread\_count** column in the **sys.dm\_resource\_governor\_workload\_groups** DMV.

## SQL Trace

Another way to derive information about your workloads is to use SQL Trace. In SQL Server 2008, there were three new events added specifically to capture Resource Governor activity.

### CPU Threshold Exceeded

If you use the REQUEST\_MAX\_CPU\_TIME\_SEC setting at the workload group level, you can capture an event every time a query runs longer than this setting. The event will fire exactly once for each offending query, and it can happen as long as five seconds after the threshold is reached. The important columns exposed to this event are CPU (CPU usage in ms), **GroupID**, **OwnerID**, **SessionLoginName**, and **StartTime**. Note that there is a **SPID** column, but it is not very interesting; this represents the background SPID that detected and reported the violation, not the SPID that actually ran the query that caused the violation. So even though it may not seem intuitive, you should look at the **OwnerID** column, not the **SPID** column, to determine the SPID that exceeded the threshold. As an example, if you have a workload group with settings as follows:

ALTER WORKLOAD GROUP [default]

WITH

(

 REQUEST\_MAX\_CPU\_TIME\_SEC = 1

);

GO

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

Then create a profiler trace with events **TSQL/SQL:StatementCompleted** and **Errors and Warnings/CPU threshold exceeded**. Finally, run a query that you know will take longer than a few seconds to run; you may have big tables prepared for this purpose, or you could use something like this:

SELECT \*

 FROM sys.objects AS s1

CROSS JOIN sys.objects AS s2

CROSS JOIN sys.objects AS s3;

 **Figure 6:** Profiler output for a statement causing a CPU threshold exceeded event

As you can see in the SQL Server Profiler output, the CPU threshold exceeded event occurred first; then, when the statement actually finished, the **StmtCompleted** event was recorded. Unfortunately, the text data for the query does not get reported along with the violation event, so you have to manually correlate the original query to the event. On a busy system, or when the timeout period is long, this is not likely to be something you’re going to want to handle with Profiler; instead, you should script the trace as above and capture the data to a file or table.

### PreConnect:Starting

You can use this event to capture whenever classification starts; this can tell you how often connections are running the classifier function. The useful columns available are **SPID**, **StartTime**, **ObjectID**, and **ObjectName**. The **ObjectID** and **ObjectName** columns refer to the classifier function in **master**, so in most cases this information can be ignored, unless you are constantly switching out the active classifier function.

### PreConnect:Completed

You can use this event to benchmark your classifier function. The interesting columns are **SPID**, **StartTime**, **EndTime**, **Duration**, **CPU**, **Reads**, **Writes**, and **GroupID**. There are also columns you can capture to help diagnose problems with the classifier function, such as **Error** and **State**. However, because the classifier function is created WITH SCHEMABINDING, this makes it difficult to generate typical errors, such as referencing an object or column that does not exist. If you really go out of your way, you can force an error in the function; you will find that the session goes into the default workload group as expected, and that both **PreConnect** events still fire, as shown in this screen shot.

 **Figure 7:** Profiler output for PreConnect events and a classifier function error

## Extended Events

Extended Events is an extremely powerful but lightweight eventing architecture introduced in SQL Server 2008. One way that you can use Extended Events (XEvents) to monitor Resource Governor activity is to utilize the “system\_health” event session (in concept, similar to the default trace), which is running by default in SQL Server 2008, and records several classes of events that Microsoft has determined are indicative of server health. These are typically events that users never see, and unless you are specifically watching for them, you will not be aware that they are happening. Unlike SQL Trace, which just reports the event in isolation, the XEvents session also records the query that triggered the event. In a completely contrived example, you could:

* Configure your instance with a very low min/max server memory setting (for example, 128 MB).
* Configure the default pool with MAX\_MEMORY\_PERCENT = 1.
* Run a very large query, such as **sys.objects** cross-joined to itself multiple times.

In this case, you should get an error message like this:

Msg 701, Level 17, State 123, Line 1

There is insufficient system memory in resource pool 'default' to run this query.

If you were not running a trace during this time (and we are not running traces all the time), it can be difficult to determine which query caused the error. The SQL Server error log will have an entry similar to the above, as well as some diagnostic output (much like DBCC MEMORYSTATUS):

<date/time> spid51 Error: 701, Severity: 17, State: 123.

<date/time> spid51 There is insufficient system memory in resource pool 'default' to run this query.

The system\_health XEvents session will have much more detail, and in a more usable format; run the following query in SQL Server Management Studio, with Results to Grid (Ctrl+D) set:

SELECT [XML] = CAST(t.target\_data AS XML)

FROM sys.dm\_xe\_session\_targets AS t

INNER JOIN sys.dm\_xe\_sessions AS s

ON s.[address] = t.event\_session\_address

WHERE s.name = 'system\_health';

In the grid result, you can click on the XML output to see the raw XML in a new window. You might see a large number of events, depending on the relative health of your system and how long it has been running, but part of the output will contain a corresponding event, such as the following (trimmed for brevity):

<event name="error\_reported" package="sqlserver" id="100" version="1" timestamp="**2009-03-27T02:42:15.285Z**">

 <data name="error">

 <type name="int32" package="package0" />

 <value>**701**</value>

 <text />

 </data>

…

<data name="message">

 <type name="unicode\_string" package="package0" />

 <value>**There is insufficient system memory in resource pool 'default' to run this query.**</value>

 <text />

 </data>

 <action name="callstack" package="package0">

 …

 </action>

 <action name="session\_id" package="sqlserver">

 <type name="uint16" package="package0" />

 <value>**51**</value>

 <text />

 </action>

 <action name="sql\_text" package="sqlserver">

 <type name="unicode\_string" package="package0" />

 <value>**SELECT \* FROM sys.objects AS s1 CROSS JOIN**

 **… [rest of big query that caused the memory error]**</value>

 <text />

 </action>

 <action name="tsql\_stack" package="sqlserver">

 <type name="unicode\_string" package="package0" />

 <value>&lt;frame level='1' handle='**0x020000007492133342425DC7D4D76E4FE99BB5D01609E593**' line='1' offsetStart='0' offsetEnd='-1'/&gt;</value>

 <text />

 </action>

 </event>

(Those who know their way around XPath or XQuery should be able to make quick work of parsing this result for interesting events, or even automating the process so alerts are sent when a new event is discovered.)

There are some other interesting details that XEvents can provide that are not easily obtained elsewhere. Paul S. Randal recently published a TechNet article about troubleshooting with Extended Events, and one of the examples showed how to aggregate I/O usage by resource pool. The article is here:

**Advanced Troubleshooting with Extended Events**
<http://technet.microsoft.com/en-us/magazine/dd314391.aspx>

Because there is no user interface for XEvents in SQL Server Management Studio, if you would like to explore it more interactively, you should have a look at Jonathan Kehayias’ Extended Event Manager project on CodePlex. This will give you a head start on creating and viewing XEvents metadata with an easy-to-use graphical interface:

**SQL Server 2008 Extended Events Manager**<http://extendedeventmanager.codeplex.com/>

Finally, Adam Machanic has written an Extended Events Code Generator (currently in beta), which makes it much easier to consume Extended Events data without having to know XQuery:

 **Extended Events Code Generator**
 <http://sqlblog.com/files/folders/beta/entry14175.aspx>

If you would like to explore XEvents further, you should consult both SQL Server Books Online and Kalen Delaney’s new title, *Microsoft SQL Server 2008 Internals*, from Microsoft Press; both have very thorough introductions to the technology.

## An Example: Learning About Your System

It can be very useful to leverage the Resource Governor for simply learning about the workloads on your system. A lot of times there is a SQL Server instance in use by an array of users and applications, and it is difficult to get a stable handle on exactly how the instance is being used, by whom, and when. Because you can determine quickly through a random sampling of **sys.dm\_exec\_sessions** exactly which application names, user names, and host names are running queries against SQL Server, you should be able to build a representative list in a short amount of time. This will allow you to build a small set of resource pools and workload groups that represent each set that you consider “similar,” so that you can analyze them in groups, rather than treating every session and request distinctly. A classifier function can use this data to divert the requests into the different workload groups, analyze the performance characteristics of each workload independently – to learn about your workloads both in general and, potentially, in the event of a performance issue.

The plan of action in this case would be:

1. Take random samplings to get a good grasp of the different computers, users, and applications that are currently connecting to the instance:

SELECT [program\_name],[host\_name],[login\_name]

 FROM sys.dm\_exec\_sessions

 WHERE session\_id > 50;

1. For each obvious segment within that set, plan out how you want to analyze each segment. As an example, imagine you had the following output.

|  |  |  |
| --- | --- | --- |
| program\_name | host\_name | login\_name |
| Microsoft SQL Server Management Studio | WORKSTATION2 | Domain\Bill |
| Microsoft SQL Server Management Studio – Query | WORKSTATION2 | Domain\Bill |
| .NET SqlClient Data Provider | APPSERVER1 | AppAccount |
| Windows® Operating System | DBSERVER | Domain\ServiceAccount |
| Internet Information Services | WEBSERVER1 | WebAccount |
| SQLAgent - TSQL JobStep (Job 0x10D… : Step 3) | DBSERVER | Domain\ServiceAccount |
| SQLAgent – TSQL JobStep (Job 0x14B… : Step 1) | DBSERVER | Domain\ServiceAccount |
| OSQL-32 | APPSERVER1 | AppUser |
| SQLCMD | APPSERVER2 | AppUser |
| SQL Query Analyzer | WORKSTATION1 | Domain\Bob |
| MS SQLEM | WORKSTATION1 | Domain\Bob |

 **Table 3:** Fictitious output from sys.dm\_exec\_sessions

Now you can see some patterns here: you have Windows applications, Web applications, end users, and SQL Server Agent jobs, all running workloads against your database server. You could decide to segment them in this way.

**
 Figure 8:** Sample layout for organizing workload groups and resource pools

1. Create a resource pool for applications in general, and under that pool, create a workload group for each relevant application (or group of applications). In this case, there will be workload groups for:

	1. Requests from Web server(s)
	2. Requests from application server(s)
	3. Requests from applications or hosts you do not recognize

Here is sample code for creating this resource pool and its workload groups:

CREATE RESOURCE POOL [Pool\_Apps];

GO

CREATE WORKLOAD GROUP [Group\_Apps\_WebServers]

 USING [Pool\_Apps];

GO

CREATE WORKLOAD GROUP [Group\_Apps\_AppServers]

 USING [Pool\_Apps];

GO

CREATE WORKLOAD GROUP [Group\_Apps\_Unknown]

 USING [Pool\_Apps];

GO

1. Create a resource pool for ad hoc queries, and within that pool, create workload groups for:
2. Requests from SQL Server Management Studio / SQL Server Query Analyzer / SQL Server Enterprise Manager
3. Requests from command-line hosts, such as SQLCMD / OSQL

Here is sample code for configuring these workload groups and their resource pool:

CREATE RESOURCE POOL [Pool\_AdHoc];

GO

CREATE WORKLOAD GROUP [Group\_AdHoc\_UI]

USING [Pool\_AdHoc];

GO

CREATE WORKLOAD GROUP [Group\_AdHoc\_CMD]

USING [Pool\_AdHoc];

GO

1. Create a resource pool for all SQL Server Agent jobs, and within that pool, create a workload group for requests from SQL Server Agent jobs. Here is the code for creating this resource pool and workload group:

CREATE RESOURCE POOL [Pool\_SQLAgent];

GO

CREATE WORKLOAD GROUP [Group\_SQLAgent]

 USING [Pool\_SQLAgent];

GO

1. Create a classifier function that will route the requests to the proper workload group. This can be a little more daunting, because you do not want any of the unrecognized requests landing in the wrong workload group:

CREATE FUNCTION dbo.classifier\_Learning()

RETURNS SYSNAME

WITH SCHEMABINDING

AS

BEGIN

 DECLARE

 @host SYSNAME,

 @app SYSNAME,

 @group SYSNAME;

 SELECT

 @host = UPPER(HOST\_NAME()),

 -- BEWARE OF SPOOFING:

 @app = LOWER(APP\_NAME());

 SELECT @group = CASE

 -- SQL Server Agent job

 WHEN @app LIKE 'sqlagent%tsql%jobstep%'

 THEN N'Group\_SQLAgent'

 -- someone using SQL Server Management Studio, SQL Server Enterprise Manager/Query Analyzer

 WHEN @app LIKE '%management studio%'

 OR @app IN ('sql query analyzer', 'ms sqlem')

 THEN N'Group\_AdHoc\_UI'

 -- someone running osql or sqlcmd

 WHEN @app LIKE 'osql%' OR @app = 'sqlcmd'

 THEN N'Group\_AdHoc\_CMD'

 -- Web app on a Web server

 WHEN @app = 'internet information services'

 AND @host LIKE 'WEBSERVER[0-9]%'

 THEN N'Group\_Apps\_WebServers'

 -- .NET app on an app server

 WHEN @app = '.net sqlclient data provider'

 AND @host LIKE 'APPSERVER[0-9]%'

 THEN N'Group\_Apps\_AppServers'

 -- if it's not IIS from a Web server,

 -- and it's not .NET from an app server,

 -- but it's coming from those servers,

 -- what is it?

 WHEN @host LIKE 'WEBSERVER[0-9]%'

 OR @host LIKE 'APPSERVER[0-9]%'

 THEN N'Group\_Apps\_Unknown'

 ELSE

 N'default'

 END;

 RETURN (@group);

END

GO

1. Enable the Resource Governor with this classifier function:

ALTER RESOURCE GOVERNOR

 WITH (CLASSIFIER\_FUNCTION = dbo.classifier\_Learning);

GO

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

1. Verify the Resource Governor configuration:

SELECT

 c.classifier\_function\_id,

 function\_name = OBJECT\_SCHEMA\_NAME(c.classifier\_function\_id)

 + '.' + OBJECT\_NAME(c.classifier\_function\_id),

 c.is\_enabled,

 m.is\_reconfiguration\_pending

FROM

 sys.resource\_governor\_configuration AS c

CROSS JOIN

 sys.dm\_resource\_governor\_configuration AS m;

SELECT \* FROM sys.resource\_governor\_resource\_pools;

SELECT \* FROM sys.resource\_governor\_workload\_groups;

Results:


 **Figure 9:** Verifying Resource Governor configuration

1. Then, periodically, run queries against the DMVs to take a pulse of the system, and answer any questions you might have about how your workloads are interacting. (Keep in mind that your own queries are included, and that any “to date” questions simply mean, “since the last time the statistics were reset.”) Here are a few that might be useful:
2. How many users from each workload group are currently connected, and which of them are currently running queries?

SELECT

 g.group\_id,

 GroupName = g.name,

 ConnectedSessions = COALESCE(s.SessionCount, 0),

 ActiveRequests = g.active\_request\_count

FROM

 sys.dm\_resource\_governor\_workload\_groups AS g

LEFT OUTER JOIN

(

 SELECT group\_id, SessionCount = COUNT(\*)

 FROM sys.dm\_exec\_sessions

 GROUP BY group\_id

) AS s

ON

 g.group\_id = s.group\_id;

1. Is any resource pool experiencing a high number of query optimizations or suboptimal plan generations?

 SELECT

 p.pool\_id,

 p.name,

 g.group\_id,

 g.name,

 g.total\_query\_optimization\_count,

 g.total\_suboptimal\_plan\_generation\_count

 FROM

 sys.dm\_resource\_governor\_workload\_groups AS g

 INNER JOIN

 sys.dm\_resource\_governor\_resource\_pools AS p

 ON

 g.pool\_id = p.pool\_id

 ORDER BY

 g.total\_suboptimal\_plan\_generation\_count DESC

 --g.total\_query\_optimization\_count DESC

;

1. What is the average CPU time per request in each resource pool to date?

SELECT

 p.pool\_id,

 p.name,

 total\_request\_count = COALESCE(SUM(t.total\_request\_count), 0),

 total\_cpu\_usage\_ms = COALESCE(SUM(t.total\_cpu\_usage\_ms), 0),

 avg\_cpu\_usage\_ms = CASE

 WHEN SUM(t.total\_request\_count) > 0 THEN

 SUM(t.total\_cpu\_usage\_ms)

/ SUM(t.total\_request\_count)

ELSE

 0

 END

 FROM

 sys.dm\_resource\_governor\_resource\_pools AS p

 LEFT OUTER JOIN

 (

 SELECT

 g.pool\_id,

 g.total\_request\_count,

 g.total\_cpu\_usage\_ms

 FROM

 sys.dm\_resource\_governor\_workload\_groups AS g

 WHERE

 g.pool\_id > 1

 ) AS t

 ON

 p.pool\_id = t.pool\_id

 GROUP BY

 p.pool\_id,

 p.name;

1. How is the system utilizing cache, compile and total memory within each pool and across all pools?

;WITH poolmemory AS

(

SELECT

 pool\_id,

 cache\_memory\_kb,

 compile\_memory\_kb,

 used\_memory\_kb

FROM

 sys.dm\_resource\_governor\_resource\_pools

),

totalmemory AS

(

SELECT

 cache = SUM(cache\_memory\_kb),

 compile = SUM(compile\_memory\_kb),

 used = COALESCE(NULLIF(SUM(used\_memory\_kb), 0), -1)

FROM

 poolmemory

)

SELECT

pool\_id,

cache\_memory\_kb,

compile\_memory\_kb,

used\_memory\_kb,

 -- % of cache/compile/total this pool is using among total:

 -- (100% is the sum going \*down\* any of these columns)

 cache\_across\_pools = cache\_memory\_kb \* 100.0 / cache,

 compile\_across\_pools = compile\_memory\_kb \* 100.0 / compile,

 used\_across\_pools = used\_memory\_kb \* 100.0 / used,

 -- % of this pool's memory being used for cache/compile:

 -- (100% is the sum going \*across\* these two columns)

 pool\_cache = cache\_memory\_kb \* 100.0 /

 COALESCE(NULLIF(used\_memory\_kb, 0), -1),

 pool\_compile = compile\_memory\_kb \* 100.0 /

 COALESCE(NULLIF(used\_memory\_kb, 0), -1)

FROM

 poolmemory

CROSS JOIN

 totalmemory;

There are easily and quite literally dozens of other questions you could ask, that you could answer by querying the DMVs. And of course you have other avenues discussed previously, such as SQL Trace and Performance Monitor. Like learning about performance in general, you can get as much out of this exercise as you want – you can use it just to satisfy your curiosity, to find “bad” workloads that could benefit from optimization, or to help guide how you will use the Resource Governor to control and balance your workloads.

# Appendix E – Resource Governor Limitations

Like any other feature, the Resource Governor has both pros and cons. Before you get into leveraging the technology for specific usage scenarios, you should be familiar with some of the drawbacks.

## No Constraints on I/O

With the reduced cost and constantly increasing speed, power, and quantity of today's processors and RAM, it is often quite easy to overcome CPU and memory bottlenecks with very little effort and financial investment. Hard drive technology, on the other hand, has not enjoyed the same advances. Solid State Drives (SSDs) have been making headlines for months, but the prices are still high enough to make them impractical for regular commercial usage. And high performance storage area networks (SANs) are still prohibitively expensive, not just in terms of the initial capital expenditure, but also the ongoing maintenance and support contracts. For these reasons, disk subsystems are often the most complex hardware component to replace or improve, and as a result, I/O can often be the most prevalent bottleneck in a system. For workloads that are primarily I/O-bound, the current implementation of the Resource Governor does not provide controls for I/O activity. Note that there are definitely plans to address this scenario in a future version of SQL Server.

## Database Engine Only

Currently the Resource Governor only provides control over the CPU and memory that have been made available to the Database Engine. For broader, server-wide controls, and to employ control over other components such as Analysis Services, Reporting Services, and Integration Services, you may want to use an external tool, such as Windows System Resource Monitor (WSRM), SoftNUMA, or other external mechanisms. For some great information on resource controls, see Slava Oks' blog:

**Slava Oks**
<http://blogs.msdn.com/slavao/>

## Single Instance Only

The Resource Governor is designed to work with each SQL Server instance independently; there is no way to use the technology alone to balance resource allocation between two or more instances. In this case you are limited to the hard caps you can place on each instance, such as **sp\_configure** settings for min/max memory and processor affinity, and then within each instance you could further segregate workloads with individual Resource Governor configurations.

## Enterprise Edition Only

The Resource Governor is one of the features that have been designated as a differentiator between the Enterprise edition and the other SKUs such as Standard, Workgroup, Web, and Express. The functionality is available in both the Evaluation and Developer editions, so you will be able to configure and test your configuration outside of your production environment. However, you will need to make sure that you aren't testing the feature on a development box, and later deploying to an instance that is unable to actually use the technology because it is running the Standard or Workgroup edition.

## Resource Usage Anomalies

There are several cases where CPU and memory cannot be managed at all, or might not be managed as you expect; understanding some of these limitations can be crucial when configuring or validating Resource Governor settings:

* Lack of contention

One of the more dominant scenarios is one where resource constraints do not kick in at all, due to lack of contention. It is important to understand that the Resource Governor is designed in such a way that it will not inhibit a request from using as much of the instance's CPU bandwidth as possible, if no other requests are demanding CPU. It is typical to see systems where CPU has been capped at 30 percent, but because there is only one active request, it can freely dominate all of the CPU on the machine. For memory, the enforcement is "hard", that is, if the resource pool is capped at 50 percent of memory, it won't be allowed to exceed that amount. The remaining memory will be consumed by the data cache (buffer pool) instead.

* Short requests vs. long requests

The way that the Resource Governor responds to requests of varying durations can make it seem like the constraints are not working correctly. There are two scenarios where this is true:

* The thread that captures CPU threshold exceeded events runs roughly every five seconds; due to the timing, it is not guaranteed that it will report a query that hits the limit but does not exceed it by five seconds. This does not necessarily mean that you should avoid setting a low threshold for this event; in the best case, you could set the limit to 10 seconds and catch a query that runs for 10.00001 seconds. But you should be aware that you will not always capture events that run for durations very close to the threshold you set.

**Note:** Another complication with the CPU threshold exceeded event is that the event is reporting a session that hit the limit at that time. It may be tempting to react to these events by killing the session, or even to write a process that automatically kills the SPID involved. You should keep in mind that the session may no longer exist or, worse, by the time you run your KILL command, it may be doing something else entirely. In fact it may have even been recycled and assigned to a new connection. This kind of knee-jerk reaction to a long-running query can potentially cause greater system instability than the original query might have without any intervention.
* Internally, SQL Server uses CPU quantums for scheduling decisions (note that this is unrelated to operating system quantum). Essentially, scheduling decisions are made every 4 milliseconds, or when the query completes within the 4 ms window. If your workload combines short OLTP-type queries (< 4 ms) with longer queries (> 4 ms), the Resource Governor will, in effect, favor longer queries that can continue uninterrupted on the CPU without being affected by new decisions. This may make it appear that your CPU controls are not affecting the longer queries. The suggestion is to not rely solely on CPU controls to constrain both short- and long-running queries in a mixed workload, but to also employ other settings, such as parallelism, concurrency, and memory utilization constraints.
* I/O-bound workloads

You may observe scenarios where one workload is using CPU and another is waiting on disk I/O. In this case the constraints will not appear to be in place, because Resource Governor cannot apply controls to tasks that are in a waiting or pre-emptive state. So even though the CPU-bound workload is supposed to be constrained to a certain percentage of CPU (or a lower importance), the constraints will not be enforced because, in this simple example, it only sees one “eligible” workload. This is one reason why the current implementation of the Resource Governor may not be appropriate for workloads that are often I/O-bound.

* CPU usage across schedulers

CPU enforcement applies on a per scheduler basis. There is a scheduler per CPU on the machine. The algorithm modifies scheduling of a task based on the settings of the workload group (IMPORTANCE) and resource pool (CPU\_\*\_PERCENT) and does not limit actual CPU usage of the SQL Server process. In order to observe the effect of CPU governance, you will need multiple tasks in different workload groups/resource pools sharing the same scheduler.

Sometimes you may find that two requests could end up on different schedulers, so it will appear as if the Resource Governor is not working correctly. In fact this is why, in demonstrations of the technology, you will see that the instance has been configured using **sp\_configure** to use exactly one CPU. Without having to predict or analyze which scheduler a request ends up on, the demos are much more predictable, because everything is normalized to exactly one CPU. In the production environment this is not likely a problem as the load is evenly distributed across all schedulers or queries are running in parallel (that is, utilizing multiple schedulers).

* Importance vs. priority

As described earlier, many users have a natural instinct to equate importance with priority. With the Resource Governor, the importance gives a numeric weight to the request. Consider the case where two tasks are vying for CPU within the same resource pool, with one (task A) assigned a medium importance, and the other (task B) assigned a low importance. This does not mean that task A goes first, and when it is finished, task B can start. What it means is that the scheduler will work on both tasks, but give a greater percentage of its cycles to task A. In this simple example, with no other contention on that scheduler, it will give 3 out of every 4 units of work to task A. The scheduler will do its best to maintain this ratio throughout the duration of the queries.

* Pre-emptive and external tasks

There are several types of requests that are off limits to the Resource Governor. These include extended procedures, COM/OLE automation (including the sp\_OA set of procedures), linked server queries, database mail, CLR objects that include their own garbage collection, and any multi-page memory allocations. As with the I/O limitation, there are at least cursory plans to deal with some of these exclusions in future versions; in the meantime, be aware that if you try to constrain workloads that use these features, you might not experience the level of control that you would like.

* The “runaway” query is still possible

Because of some of the situations described above, the dreaded "runaway" query is always a possibility, even in the most meticulously configured instance. What the Resource Governor allows you to do in this case is not necessarily make the runaway query impossible, but rather reduce its likelihood, and ensure that its impact on the rest of the system is minimized.

# Appendix F – Cleaning Up the Resource Governor Configuration

If you are planning on running any of these examples on a test system, it is useful to understand how to clear pre-existing Resource Governor settings, so that you are starting fresh each time. One of the easiest ways to do this without dealing with existing sessions is to disable the Resource Governor, restart the SQL Server service, and then drop all of the user-defined workload groups and resource pools. More explicitly, the steps are as follows:

1) Run the following code to disable the Resource Governor and reset statistics:

ALTER RESOURCE GOVERNOR WITH (CLASSIFIER\_FUNCTION = NULL);

ALTER RESOURCE GOVERNOR DISABLE;

GO

2) Restart the SQL Server service.
3) Run the following to generate the DROP commands for workload groups:

SELECT 'DROP WORKLOAD GROUP ' + QUOTENAME(name) + ';'

 FROM sys.resource\_governor\_workload\_groups

WHERE name NOT IN ('internal', 'default');

4) Copy and paste the output from 3) into a new query window, and then execute it.
 5) Run the following to generate the DROP commands for the resource pools:

SELECT 'DROP RESOURCE POOL ' + QUOTENAME(name) + ';'

 FROM sys.resource\_governor\_resource\_pools

 WHERE name NOT IN ('internal', 'default');

6) Copy and paste the output from 5) into a new query window, and then execute it.
7) Reconfigure the Resource Governor:

ALTER RESOURCE GOVERNOR RECONFIGURE;

GO

You may want to keep your various classifier functions around for later reference, especially if the classification rules you are testing are complex. Otherwise, you can just drop any existing classifier functions manually after the above steps are complete. If you are going to keep them, obviously a good practice here would be to use a consistent naming scheme, such as dbo.classifier\_<description>, so they are easy to find and organize.

You may also find it useful to create a view that you can reference to examine all of your user-defined workload groups and resource pools and how they relate to each other. This can be a quick way to validate your configuration or runtime settings without having to write all those pesky JOINs. Here are two examples: one matches pools and groups using the configuration values, and the other uses the DMVs, in case you have made configuration changes that are not yet published to the Resource Governor:

USE [master];

GO

CREATE VIEW dbo.RGObjects\_Config

AS

 SELECT

 p.pool\_id,

 PoolName = p.name,

 g.group\_id,

 GroupName = g.name

 FROM

 sys.resource\_governor\_resource\_pools AS p

 LEFT OUTER JOIN

 sys.resource\_governor\_workload\_groups AS g

 ON

 p.pool\_id = g.pool\_id;

GO

CREATE VIEW dbo.RGObjects\_Runtime

AS

 SELECT

 p.pool\_id,

 PoolName = p.name,

 g.group\_id,

 GroupName = g.name

 FROM

 sys.dm\_resource\_governor\_resource\_pools AS p

 LEFT OUTER JOIN

 sys.dm\_resource\_governor\_workload\_groups AS g

 ON

 p.pool\_id = g.pool\_id;

GO