

**Using Visual Studio 2005 to Perform Load Testing on a SQL Server 2005 Analysis Services**

**SQL Server Technical Article**

**Writers:** Jaime Basilico, Microsoft Corp.

Dmitri Tchikatilov, Microsoft Corp.

**Technical Reviewers:** Runying Mao, Microsoft Corp.

Len Wyatt, Microsoft Corp.

Dave Wickert, Microsoft Corp.

Jim Carroll, Microsoft Corp.

**Published:** November 2006

**Applies To:** Microsoft® SQL Server™ 2005 Analysis Services and Microsoft Visual Studio® 2005 Team System

**Summary:** Microsoft Visual Studio 2005 Team System includes a Load Test tool that you can use for performance and stress testing of a Microsoft SQL Server 2005 Analysis Services deployment. The Load Test tool runs the tests that you create, and optionally logs the data into a SQL Server database. You can monitor tests as they progress, review performance data after the tests complete, and precisely determine what the threshold is for a specific report server deployment. This article contains step-by-step instructions for installing, creating a unit test, and best practices around stress test methodologies for stress testing Analysis Services. Instructions are also provided for setting up the load test that you use to specify load patterns. The article assumes that you have the AdventureWorks Data Warehouse sample SSAS database so that you can try these steps on your computer.

**Copyright**

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

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

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

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

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

©2006 Microsoft Corporation. All rights reserved.

Microsoft,SQL Server, Windows, and Windows Serverare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

**Table of Contents**

**About Project REAL**

[**Introduction**](#_3znysh7)

Choosing Analysis Workload

Requirements

Visual Studio 2005 Team Edition for Software Testers

[AdventureWorks Sample Database and Cubes](#_4d34og8)

[**Installing Performance Testing Environment**](#_2s8eyo1)

[Steps for Installing AS Query Generator](#_17dp8vu)

[Steps for Installing AS Load Simulator](#_3rdcrjn)

[**Pre-planning of Analysis Services Tests**](#_26in1rg)

[Collecting Load Queries](#_26in1rg)

[Considerations for Analysis Services Caching](#_lnxbz9)

[**Preparing Performance Testing Environment**](#_35nkun2)

[Planning Test Environment Architecture](#_1ksv4uv)

[Setting Analysis Services Server Properties](#_44sinio)

[Validating MDX](#_z337ya)

[Optimizing Cube Aggregations](#_3j2qqm3)

[Performance Testing Environment Checklist](#_1y810tw)

[**Analysis Services Performance Tests Methodology**](#_4i7ojhp)

[Defining Goals for Stress Tests](#_2xcytpi)

[Determining Run Lengths](#_1ci93xb)

[Monitoring Server Performance](#_3whwml4)

[Identifying Analysis Services Bottlenecks](#_2bn6wsx)

[Tuning Analysis Services Server Properties](#_qsh70q)

[Tuning Cube and MDX Query Design](#_3as4poj)

[**Running Performance Tests**](#_1pxezwc)

[Generating MDX Client Files](#_49x2ik5)

[Configuring Load Simulator](#_2p2csry)

Configuring Visual Studio for Load Test Run

[Running Stress Tests](#_3o7alnk)

[**Reporting Performance Test Results**](#_23ckvvd)

[Direct Query of the Database](#_ihv636)

[Leveraging Reporting Services](#_32hioqz)

[**Considerations for Multiple Analysis Services Databases**](#_1hmsyys)

[Generation of MDX Client Files for Multi SSAS DB Environment](#_41mghml)

[Executing Stress Tests for Multi SSAS DB Environment](#_vx1227)

[Reporting Performance Test Results for Multi SSAS DB Environment](#_3fwokq0)

[**Conclusion**](#_1v1yuxt)

[**Appendix - What's in the Visual Studio Team System Load Test Results Store**](#_4f1mdlm)

About Project REAL

Project REAL is an effort to discover best practices for creating business intelligence applications that are based on Microsoft® SQL Server™ 2005 by creating reference implementations that are based on actual customer scenarios. This means that customer data is brought in-house and is used to work through the same issues that the customers face during deployment.

These issues include the following:

The design of schemas for relational databases and for Analysis Services.

The implementation of a data extraction, transformation, and loading (ETL) process.

The design and deployment of client front-end systems, both for reporting and for interactive analysis.

The sizing of systems for production.

The management and maintenance of those systems on an ongoing basis, including incremental updates to the data.

By working with real deployment scenarios, we gain a better understanding of how to work with the tools. Our goal is to address the full gamut of concerns that a large company would face during their own real-world deployment.

This white paper offers a detailed technical discussion on how to perform performance analysis for SQL Server Analysis Services using Visual Studio 2005 Team Edition for Software Testers. The infrastructure discussed in this white paper refers to how the Analysis Services performance and throughput tests were implemented in several field engagements, as well as through needs derived from Project Real’s usage of the predecessor load simulation toolset.

For an overview of Project REAL, see the Project REAL: Technical Overview white paper (http://www.microsoft.com/technet/prodtechnol/sql/2005/projreal.mspx). A number of papers, tools, and samples will be produced over the lifetime of Project REAL. To find the latest information, visit the Project REAL Web site at <http://www.microsoft.com/sql/bi/ProjectReal/>.

Introduction

This article describes how to use Microsoft® Visual Studio® 2005 Team Edition for Software Testers to run performance characterization tests for Microsoft SQL Server™ 2005 Analysis Services. You can use this article as a guideline for capacity planning or to assess performance before rolling out reports on a production server.

This article contains step-by-step instructions for setting up a project, creating Web page and unit tests, creating and configuring a load test, running the test, and evaluating the results. After you create the tests, you can run them on different server configurations to quantify the improvement in performance when you change hardware components or modify a report definition or query, or specify different rendering formats.

Choosing Analysis Workload

This article uses the AdventureWorks DW sample Analysis Services database to illustrate key concepts. You can use the sample SSAS DB if you want to use the sample code and steps provided, or you can work with your own database and modify the code and steps accordingly. When you perform load tests, the queries must be able to run with no user interaction required. If the report prompts for data source credentials or parameter values, you must temporarily modify the report to use stored or integrated credentials and default parameters for the purpose of running the tests.

Requirements

This article assumes that you have the following software and samples installed on a test server:

Microsoft Visual Studio 2005 Team Edition for Software Testers

Microsoft SQL Server 2005 Analysis Services

Microsoft SQL Server 2005 Reporting Services

SQL Server Management Studio

AdventureWorks sample RDBMS database

AdventureWorks sample Analysis Services database

Permission to access the Database Engine, create databases, and retrieve data from the AdventureWorks database. You must also have role assignments that grant access to the Analysis Services database cubes.

Visual Studio 2005 Team Edition for Software Testers

You can install a subset of the Visual Studio components. The following screenshot shows the Team Developer and Test tools that are used in this exercise. The tools that you will use include Performance Tools, Code Analysis Tools, and Testing Tools.

You must also have a language project installed. The sample code provided in this article is in Microsoft Visual C#, but you can use another language if you want to use your own code.

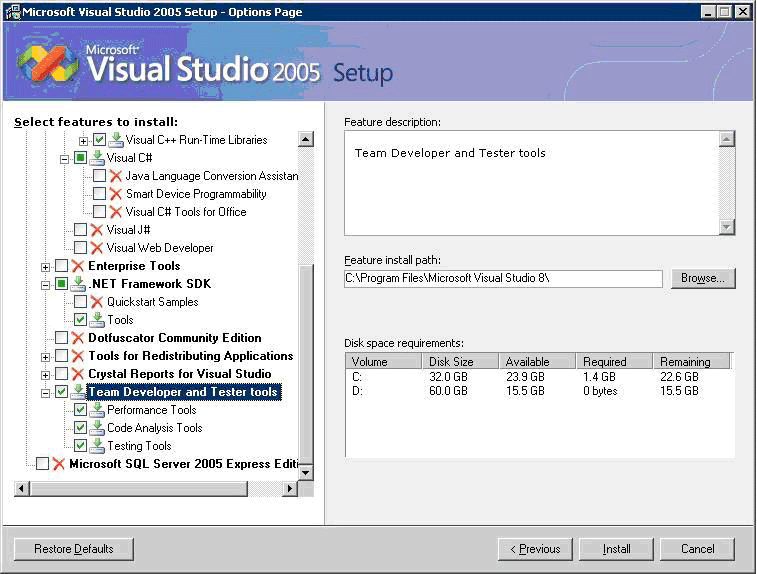


Figure 1. Visual Studio Setup

AdventureWorks Sample Database and Cubes

AdventureWorks DW is a sample relational database and Analysis Services database that is included with SQL Server 2005. If you want to use the AdventureWorks sample database(s), first make sure that the samples are installed. By default, they are located at <*drive*>:\Program Files\Microsoft SQL Server\90\Samples. If not they are not installed, you need to install them. For instructions about how to install and uninstall the samples, see [Installing Samples](http://go.microsoft.com/fwlink/?LinkId=67739) in SQL Server Books Online. You can also download the samples from the [Microsoft Download Center](http://go.microsoft.com/fwlink/?LinkId=31046).

Before you start, verify that you can access the AdventureWorks sample database and run queries against the cubes by starting SQL Server Management Studio and browsing the cubes. We also recommend using “Adventure Works DW” Analysis Services solution, which can be found: *<drive>*:\Program Files\Microsoft SQL Server\90\Tools\Samples\AdventureWorks Analysis Services Project\Enterprise or *<drive>*:\Program Files\Microsoft SQL Server\90\Tools\Samples\AdventureWorks Analysis Services Project\Standard (depending on your choice of Standard or Enterprise Analysis Services deployment). This solution can be deployed on your target Analysis Services Server.

Installing Performance Testing Environment

In order to generate load on Analysis Server you need to set up client site tools which consist of two Visual Studio C# solutions. The first one is Analysis Services Query Generator which purpose is to take specially formatted XML files with MDX queries as inputs and produce XML files with actual MDX queries. These XML files will be used in actual load tests by the Analysis Services Load Simulator tool. You can generate a set of XML files once, during the preparation phase of your load tests and use these files to create the requests from multiple client machines running Load Simulator. Both Query Generator and Load Simulator Visual Studio 2005 solutions (C#) are provided with full source code downloadable with this article.

The “Query Generator” solution (ASQueryGenerator.sln) produces a console application ASQueryGenerator.exe and has dependency on ADOMD.Net libraries (which come as a part of client side install of SQL Server Tools or can be downloaded separately from MSDN) as well as Office 2003 primary interop assemblies PIAs (can be downloaded from Microsoft Office Developer Center).

The “Load Simulator” solution (ASWorkloadTool.sln) produces a .Net assembly ASWorkloadTool.dll which can be then executed from a testing environment inside Visual Studio to run the tests and generate load on Analysis Services server. You can also run the load tests from command line mode by using MSTest.exe. [MSTest.exe](http://msdn2.microsoft.com/en-us/library/ms182489.aspx) utility for running tests from command line comes with Visual Studio 2005 Team Edition for Testers. You will need to deploy this utility to all clients which will be running queries against Analysis Services server.

Based on our test findings, a typical workstation with 2 Gb of RAM and Pentium 4 class processor can generate approximately 200 to 300 concurrent connections to Analysis Services Server without being overloaded (reaching over 75% of CPU utilization and having less than 10% of memory available). This number of concurrent connections is a general observation and may vary depending on your workstation configuration. Therefore, in order to generate large numbers of concurrent users (several hundreds) you may need to use multiple client machines. This can be accomplished by installing Visual Studio Team Edition for Testers and using [Controller and Agent functionality](http://msdn2.microsoft.com/en-us/library/ms182635.aspx). In our tests we simply run command line scripts using MSTest.exe to run the load tests from multiple client workstations. We copied the directory with the ASWorkloadTool solution files on several client workstations and modified it accordingly.

Steps for Installing AS Query Generator

Once you’ve downloaded the source code, go to your download location. For example: C:\ASQueryGenerator

So traverse to folder: C:\ASQueryGenerator. Open the Visual C# Project file named ASQueryGenerator.csproj in Visual Studio 2005. Click menu item “Build”, select “Configuration Manager…”. A window named “Configuration Manager” will be popped up. In “Active Solution Configuration:”, click the drop down menu and change “Debug” to “Release”. Then click “Close”. Finally click menu item “Build” and select “Build ASQueryGenerator”.

The “Output” window will display the build progress. It will indicate where the final binary is located. For example:

------ Build started: Project: ASQueryGenerator, Configuration: Release Any CPU ------

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 /define:TRACE /reference:"C:\Program Files\Microsoft.NET\ADOMD.NET\90\Microsoft.AnalysisServices.AdomdClient.dll" /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Xml.dll /debug:pdbonly /optimize+ /out:obj\Release\ASQueryGenerator.exe /target:exe AssemblyInfo.cs ASQueryTemplateParser.cs ASQueryTemplate.cs ClientFile.cs Tokens.cs ASQueryGenerator.cs TokenKey.cs TupleDictionary.cs

Compile complete -- 0 errors, 0 warnings

ASQueryGenerator -> C:\ASQueryGenerator\bin\Release\ASQueryGenerator.exe

========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========

Steps for Installing AS Load Simulator

Analysis Services Workload Tool is developed on top of Ocracoke (Visual Studio 2005’s Load Testing Tool) framework. Please follow the steps listed below to launch a work load test for Analysis Services.

1. **Install Visual Studio 2005 Team Suite for Software Testers (Ocracoke.)**

<http://msdn.microsoft.com/vstudio/teamsystem/products/test/default.aspx>

<http://msdn.microsoft.com/vstudio/teamsystem/tester/>

1. **Open the project/solution.**

Go to your source depot enlistment. In our team, we use the same client mapping structure. For example: C:\ASWorkloadTool. Then traverse to folder: C:\ASWorkloadTool, then open the Visual C# Solution file named ASWorkloadTool.sln in Visual Studio 2005.

1. **Create load test result repository.**

Connect to local SQL server instance. Run the query file located at C:\ASWorkloadTool\loadtestresultsrepository.sql, which creates a database named “LoadTest” and tables needed to store load test result.

Pre-planning of Analysis Services Tests

Collecting Load Queries

One of the most important things you need for realistic performance tests is a set of Analysis Services queries which would accurately represent the actual users connecting to your server and retrieving the data. You can collect sample queries by using SQL Server Profiler and connecting to an instance of Analysis Services Server. Keep in mind, that usage of the Profiler introduces significant overhead typically is done against test servers, not production servers. After you open a new trace, check “Save to file” check box to save the trace evens for easy query extraction. Then switch to the “Event Selection” tab and select “Query Events – Query Begin and Query End” and clear all other events. After you collected the load queries using Profiler, you can transfer them to the XML template file for generation of the XML files using the Query Generator tool. These XML files will be ultimately used by the Load Simulator.

All queries recorder by the Profiler can be divided in two broad categories: metadata navigational queries which retrieve information about the cube/dimension structure and data reporting queries which return numerical cube data. The fist type is of a much lesser concern from the performance standpoint, in most cases. The expectation is that reporting queries may take some time, at least a couple of seconds, but faster than relational reporting – but that navigation queries should be extremely fast, under a few seconds. Poor navigational queries could result in an end-user experience that is very frustrating. It is one thing if a large report takes 20 seconds to run. It is totally different if every navigation query is 20 seconds. Performance studies tend to emphasize reporting queries; but for acceptance by end-users it is best to also include navigational queries in the mix if you can.

When analyzing the data reporting queries, it is evident that these queries may be quite convoluted and contain named set expressions (using SET keyword). Named sets can be scoped on a specific query (query-scoped) or for the length of the session (session-based). Query-scoped sets can be easily handled within the tool and require no modification. They can be easily parameterized using Tokens as we describe in later section when we go through the process of preparing MDX client files.

The process of using session-based named sets consists of three steps. Fist, creation of the named set by using CREATE SET expressions (see snippet below)

CREATE SESSION SET [Adventure Works].[{FBF95DF5-03D1-43DD-BB93-5899C5819FAB}Pivot12Axis0Set0] AS{{ [Product].[Category].[All Products] }…

Then query of the SET using SELECT statement

SELECT NON EMPTY [{FBF95DF5-03D1-43DD-BB93 5899C5819FAB}Pivot12Axis0Set0] …

And finally dropping of the SET using DROP SET

DROP SET [Adventure Works].[{FBF95DF5-03D1-43DD-BB93-5899C5819FAB}Pivot11Axis1Set0]

We have not tested this version of the Load Simulator with session-scoped named sets and recommend modifying session-based queries to either query-scoped SET or regular SELECT type queries. Since the code for the Load Testing tools is available, it is possible to extend its functionality to support session-scoped named sets.

Considerations for Analysis Services Caching

Since Analysis Services Server caches data in memory, we suggest dropping the cache before each performance test to account for the process of building the cache.

Dropping Analysis Services cache can be dropped by stopping and starting the Analysis Services process, or by leveraging the ASCMD command line tool to clear the data cache (see scenario #8 found on this page when using ASCMD: <http://msdn2.microsoft.com/en-us/library/ms365187.aspx>).

Preparing Performance Testing Environment

Planning Test Environment Architecture

Before you begin stress testing your Analysis Services server, you’ll need to create the test environment. Planning the test environment goes beyond just installing the QueryGenerator and LoadSim tools on your Visual Studio 2005 workstation; you’ll need to consider the server architecture for which you’ll run your test against. This server architecture includes more than just the Analysis Services server component. At the very least, your test environment should include at minimum one server that contains the SQL Server 2005 technology components for Analysis Services, Reporting Services, and the Database Engine. This environment may be suitable if you plan to share the server resources across not only Analysis Services, but perhaps other SQL Server services, and/or non-SQL Server 2005 applications. However, for a more scalable environment, you’ll most likely want to separate SQL Server technology components from each other. When testing the capacity of Analysis Services, a typical architecture would be to separate the RDBMS from the Analysis Services engine.

Below is an example architecture of how SQL Server 2005’s technology components can be separated for a load simulation test.

|  |
| --- |

Figure 2. Analysis Services 2005 Properties Page

Setting Analysis Services Server Properties

Several key server wide Analysis Services settings should be made to SQL Server 2005 while stress testing your scenarios. The following list describes the some of the configuration settings that should be focused on:

1. **MaxThreads** is a server property that controls how many threads will be spawned, at maximum for either query or process operations.
2. **MinThreads** is a server property that controls the minimum number of threads that will be spawned for either query or process operations.
3. **TempDir** is a property that specifies where Analysis Server places temporary files that it uses during processing operations.
4. **TotalMemoryLimit** is a server property that controls the total amount of physical memory that can be used by all SSAS operations. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 80, which indicates that a maximum of 80 percent of the total physical memory can be used by SSAS. The total amount of memory available to Analysis Server depends on the hardware platform Analysis Server runs on. A 32-bit operating system limits Analysis Server to maximum of 3 gigabytes (GB) with /Gb switch enabled in the boot.ini file. On a 64-bit platform, Analysis Server can address all memory available.
5. **DataDir** is a server property that specifies where Analysis Server data is stored per server instance.

An easy method to change Analysis Services properties can be accomplished through the SQL Server Management Studio (SSMS). Analysis Server enables you to change its server properties through a properties page exposed through SSMS. To open this properties page, follow these steps:

1. Open SQL Server Management Studio.
2. Ensure that you open a connection to Analysis Server (not Database Engine).
3. In the Object Explorer pane, right-click the server to which you have connected.
4. Click the Properties menu item.
5. Ensure the General page is selected, as it will be by default.
6. Click the Show Advanced (All) Properties check box. You’ll see the screen shown in the figure below.

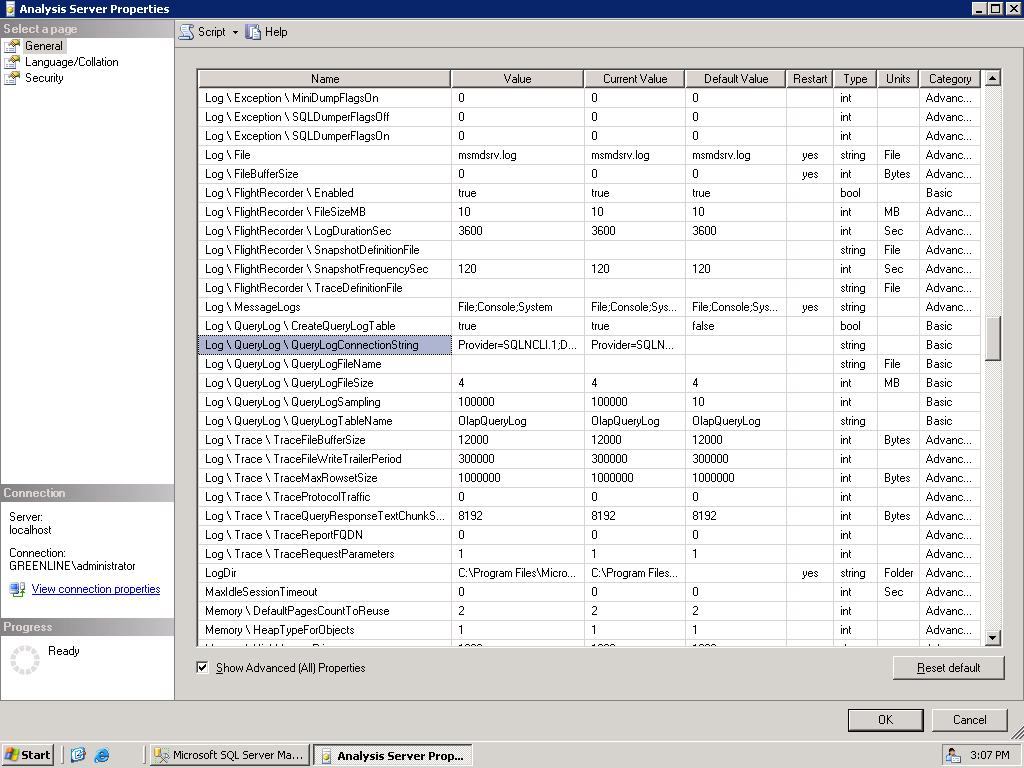


Figure 3. Analysis Services 2005 Properties Page

For additional settings and insight into Analysis Services Properties, and their effect on performance, please refer to the “Tuning Server Resources” section, found on page 91 of the SQL Server 2005 Performance Guide (<http://download.microsoft.com/download/8/5/e/85eea4fa-b3bb-4426-97d0-7f7151b2011c/SSAS2005PerfGuide.doc>).

Validating MDX

Once your stress test MDX statements are collected through means identified above in this paper, you’ll want to ensure that the MDX queries generated by the QueryGenerator run correctly.

You can Spot check some of the MDX queries that were generated by AS QueryGenerator, by using SQL Server Management Studio MDX Query tool. Simply copy one (or a series) of MDX statements contained within the XML worksheet (as generated by AS QueryGenerator), and paste the MDX into the SSMS MDX query tool windows (as seen below.

|  |
| --- |
| Figure 4. Analysis Services 2005 Query via Management Studio |

Optimizing Cube Aggregations

Before running your stress tests, you’ll want to ensure that you OLAP design has optimized aggregations. To optimize your aggregations, you’ll need to first turn on SQL Server Analysis Services query logging, then run the cube optimization wizard.

SQL Server Analysis Services uses query logs to log statistical information about queries, and stores this statistical query information in a relational database. This information is used in Usage-Based Analysis.

There are only a couple differences between Analysis Services in Microsoft® SQL Server™ 2000 and Analysis Services in SQL Server 2005. However, these two differences are quite important. Following is a brief description of each of these important differences.

By default, Analysis Services 2005 does not log queries into the query log table. To log queries, you need to adjust Analysis Services properties.

Analysis Services 2005 does not support Microsoft Office Access databases as a hosting technology for the query log table. You must point Analysis Services to the SQL Server database if you want a to create a query log table and log queries.

Now, before you go off an start optimizing your stress test cube, you may also want no aggregations at all, thus testing with “zero aggregations” (this will lead to least amount of cube processing elapsed time and minimized disk storage, but also worst possible query performance). Most likely your stress test will contain various levels of aggregation scenarios, and will then test with multiple levels of aggregation to determine aggregations effect on query response time. For example, you may stress test a single cube with no aggregation, 30% aggregation, 60% aggregation, and then 100% (full) aggregation.

| Figure 5. Analysis Services 2005 Usage Based Optimization Wizard |
| --- |

Here is a suggested aggregation design cycle that you should follow when running your stress tests.

* Use Storage Design Wizard (~20% perf gain) to design initial set of aggregations
* Enable query log and run pilot workload (beta test with limited set of users)
* Use Usage Based Optimization (UBO) Wizard to refine aggregations
* Use larger perf gain (70-80%)
* Reprocess partitions for new aggregations to   
  take effect

For additional information on Analysis Services Aggregation, and their effect on performance, please refer to the “Maximizing the Value of Aggregations” section, found on page 24 of the SQL Server 2005 Performance Guide (<http://download.microsoft.com/download/8/5/e/85eea4fa-b3bb-4426-97d0-7f7151b2011c/SSAS2005PerfGuide.doc>).

Performance Testing Environment Checklist

1. Collect MDX statements (data queries and navigational queries) using Profiler
2. Test MDX
3. Setup/Configuration of SQL Server DB Engine, Analysis Services, and Reporting Services instances
4. Creation of LoadTest database
5. Configure Analysis Services Query Logging
6. Modify Default Analysis Services Server Properties as neded
7. Optimize Analysis Services Cube Aggregations

Analysis Services Performance Tests Methodology

Defining Goals for Stress Tests

As with any stress test, there needs to be defined goals for the test(s) itself. This is no different for AS LoadSim using Visual Studio 2005 Team Test. Since AS LoadSim records average query response time for any given SSAS workload, the goal of the stress test will be determined by the thresholds that are identified prior to the start of the stress test.

For example, a reasonable goal may be average response per MDX query time must be below ten (10) seconds. Using this goal and the results from the stress test, you’ll be able to indicate the number of simultaneous users that can be supported from Analysis Services.

Additionally, the stress test may have include other goals as well, such as determining number of SSAS servers needed in production to support user load, or determining maximum memory used by SSAS during maximum user concurrency.

Determining Run Lengths

When executing a stress test against Analysis Services, proper determination around actual stress execution needs to be discussed and finalized by stress team.

A stress run duration should be long enough in length to surface any resource contention on the server, be it either caused by SSAS itself or by applications / processes interacting with SSAS when query stress runs as executed. Therefore, you’ll want to select a stress run during within Visual Studio 2005 Team Test that spans enough time to capture real value in the numbers.

In addition, determining the warming period of Analysis Services query cache is just as important, since without a proper warmup period, query stress results will be negatively skewed.

One consideration for the stress test duration comes from the number of queries stored in XML files as well as from the time interval between the query executions. The time interval (in seconds) is driven by the XML elements <ThinkTimeMin> and <ThinkTimeMax> in the configuration file (configuration.xml) for the Load Simulator tool. For example, if you recorded 3 XML files with 10 test queries each (which may for example require 2 second response time), given <ThinkTimeMin> and <ThinkTimeMax> of 10 and 30 respectively (average 20), you can expect the entire test to complete in approximately 3 \* 10 \* (2 + 20) = 660 seconds. Given this information, you can plan your “warmup” time and other test activities accordingly.

Normally, a fifteen (15) minute run time, with a three (3) minute “warm-up” period should suffice for most stress test scenarios. However, you’ll want to closely review how long each query may run, and how many simulated users your stress test allows, since long running queries with large “think times” and low number of concurrent users, may not properly warm the SSAS Query cache as planned.

If you’ll be processing SSAS OLAP cubes at same time, ensure that run duration is at least as long as processing. This will ensure that maximum resource usage across processing and querying actions is seen.

Monitoring Server Performance

When running the Analysis Services load simulation, you will need to capture not only query response times, but also various System Monitor performance counters. By default, the Visual Studio 2005 Test Tool only capture basic server level performance counters. You will want to either add some (or all) of the following counters to the load simulation test inside Visual Studio, or simply run System Monitor during the execution of the stress simulation.

| **Performance counter name** | **Definition** |
| --- | --- |
| **MSAS 2005:Memory\Memory Limit Low KB** | Displays the Memory\LowMemoryLimit from the configuration file |
| **MSAS 2005:Memory\Memory Limit High KB** | Displays the Memory\TotalMemoryLimit from the configuration file. |
| **MSAS 2005:Memory\Memory Usage KB** | Displays the memory usage of the server process. This is the value that is compared to Memory\LowMemoryLimit and Memory\TotalMemoryLimit. Note that the value of this performance counter is the same value displayed by the Process\Private Bytes performance counter. |
| **MSAS 2005:Memory\Cleaner Balance/sec** | Shows how many times the current memory usage is compared against the settings. Memory usage is checked every 500ms, so the counter will trend towards 2 with slight deviations when the system is under high stress. |
| **MSAS 2005:Memory\Cleaner Memory nonshrinkable KB** | Displays the amount of memory, in KB, non subject to purging by the background cleaner. |
| **MSAS 2005:Memory\Cleaner Memory shrinkable KB** | Displays the amount of memory, in KB, subject to purging by the background cleaner. |
| **MSAS 2005:Memory\Cleaner Memory KB** | Displays the amount of memory, in KB, known to the background cleaner. (Cleaner memory shrinkable + Cleaner memory non-shrinkable.) Note that this counter is calculated from internal accounting information so there may be some small deviation from the memory reported by the operating system. |
| **MSAS 2005: Threads\Query pool job queue length** | The number of jobs in the queue of the query thread pool. A non-zero value means that the number of query jobs has exceeded the number of available query threads. In this scenario, you may consider increasing the number of query threads. However, if CPU utilization is already very high, increasing the number of threads will only add to context switches and degrade performance. |
| **MSAS 2005: Threads\Query pool busy threads** | The number of busy threads in the query thread pool. |
| **MSAS 2005: Threads\Query pool idle threads** | The number of idle threads in the query thread pool. |
| **MSAS 2005: Cache\Current KB** | The current number of bytes also allows you to examine memory usage for the server data cache. Monitored over a period of time, this value can indicate whether or not you have additional available memory for expanding other buffers, such as the process buffer or the read-ahead buffer, to increase processing performance. |
| **MSAS 2005: Cache\Direct hits/sec** | The number of data slices that are fully completed by the server cache. The higher this number is, the better the perceived performance. |
| **MSAS 2005: Cache\Misses/sec** | The number of data slices that are not completed by the server cache, requiring retrieval from disk storage. This number should be as low as possible. |
| **MSAS 2005: Connection\ Current Connections** | Each connection has a worker thread assigned to it, so this performance counter is an excellent metric for observing current worker thread allocations. |
| **MSAS 2005: Connection\Requests/sec** | A basic performance counter, this indicates the total number of connection requests made to the Analysis server per second. This counter is generally useful as a basic measure of work performed by the Analysis server. |
| **Memory: Pages/sec** | This performance counter indicates the number of I/O operations needed to support virtual memory. Ideally, this number should be as low as possible; a high number demonstrates too little available physical memory. Increasing available physical memory should reduce the number of page faults, and therefore reduce the amount of virtual memory used to support active processes such as Analysis Services. |
| **Memory: Available bytes** | This performance counter indicates the amount, in bytes, of available physical memory. Combined with the Pages/sec system counter, this counter can be used to further quantify the amount of available physical memory. |
| **PhysicalDisk: Current Disk Queue Length** | This counter represents the current number of queued disk operations. If this number spikes during poor processing performance, especially during the base or aggregating phases, then the current disk storage solution may not be adequate to support the needs of Analysis Services. Ideally, the value of this performance counter should be as low as possible at any given time. |

Identifying Analysis Services Bottlenecks

When trying to determine Analysis Services bottlenecks, one must take into consideration the size of the cube(s), number of dimensions, complexity of the MDX queries being run against the cube/server, hardware involved, etc. Basically, there are many different factors that could represent the bottleneck. The method to reveal these potential bottleneck culprits is to sift through the available information provided by SQL Server Profiler and System Monitor.

To start with, you should focus on where the bottleneck is as shown in System Monitor data. That could be memory pressure, high CPU utilization, elevated disk I/O, or network latency for example. From this insight, your next step is to determine if the largest percentage of time (aka the bottleneck) is going into the Formula Engine (FE) or Storage Engine (SE) of Analysis Services. This will indicate whether or not you’ll need to modify your cube and MDX, or perhaps just add some additional memory to your server (or adjust SSAS server properties).

For additional information on identifying Analysis Services bottlenecks, please refer to the “Enhancing Query Performance” section, found on page 8 of the SQL Server 2005 Performance Guide (<http://download.microsoft.com/download/8/5/e/85eea4fa-b3bb-4426-97d0-7f7151b2011c/SSAS2005PerfGuide.doc>).

Tuning Analysis Services Server Properties

Now that you’ve identified potential Analysis Services bottlenecks while running the stress tests, your next step will be to make adjustments to Analysis Services server properties. Making these server property adjustments will ensure that you’ve got a correctly configured server environment before making any adjustments to your overall cube design and query code.

| **AggregationMemoryLimitMax** is a server property that controls the maximum amount of memory that can be used for SSAS aggregation processing. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 80, which indicates that a maximum of 80% of the total physical memory can be used for SSAS aggregation processing. | | Property Name | General Page: OLAP \ Process \ AggregationMemoryLimitMax | | --- | --- | | Default Value | 80 | | Unit of Measure | Percentage if between 0 and 100.  Bytes if greater than 100. | | Data Type | Double | | Minimum Value | 0 – SSAS will build aggregations for a single partition segment at a time, swapping rest of the segments to disc. | | Maximum Value | For Percentage: 100 – SSAS will use up to 100% of total physical memory.  For Bytes: 1.79E + 308 is the maximum value that can be specified. | | Requires Restart | No | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **AggregationMemoryLimitMin** is a server property that controls the minimum amount of memory that will be used for SSAS aggregation processing. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 10, which indicates that a minimum of 10% of the total physical memory will be used for SSAS aggregation processing. | | Property Name | General Page: OLAP \ Process \ AggregationMemoryLimitMin | | --- | --- | | Default Value | 10 | | Unit of Measure | Percentage if between 0 and 100.  Bytes if greater than 100. | | Data Type | Double | | Minimum Value | 0 – SSAS will not guarantee that memory gets allocated for Aggregation processing. | | Maximum Value | For Percentage: 100 – SSAS will use up to 100% of total physical memory.  For Bytes: 1.79E + 308 is the maximum value that can be specified. | | Requires Restart | No | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **BufferMemoryLimit** is a server property that controls the amount of memory that is used to store/cache data coming from relational data sources and for processing cubes, dimensions and partitions. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 60, which indicates that a maximum of 60 percent of the total physical memory can be used as the buffer memory limit. | | Property Name | General Page: OLAP \ Process \ BufferMemoryLimit | | --- | --- | | Default Value | 60 | | Unit of Measure | Percentage if between 0 and 100.  Bytes if greater than 100. | | Data Type | Double | | Minimum Value | 0 – SSAS will limit memory available for processing to the absolute minimum. | | Maximum Value | For Percentage: 100 – SSAS will use up to 100% of total physical memory for buffer memory.  For Bytes: 1.79E + 308 is the maximum value that can be specified. | | Requires Restart | No | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **DatabaseConnectionPoolMax** is a server property that controls the size of the connection pool. Opening connection to the relational database is expensive operation. It might take up to several tens of seconds to get a single relational database connection fully initialized. Analysis Server will not close established connections immideately but will keep them in the pool till next operation comes. | | Property Name | General Page: OLAP \ Process \ DatabaseConnectionPoolMax | | --- | --- | | Default Value | 50 | | Unit of Measure | Number of connection objects | | Data Type | Integer | | Minimum Value | 1 | | Maximum Value | 2147483647 | | Requires Restart | Yes | | Special Notes | Increase the property if many processing jobs run in parallel. Decreasing the value will cause SSAS to be more responsive in cases when databases referenced by connections are no longer available. | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **LowMemoryLimit** is a server property that specifies the low threshold of physical memory available to the server. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. For more information on memory, see the “Memory” section of this paper. | | Property Name | General Page: Memory \ LowMemoryLimit | | --- | --- | | Default Value | 75 | | Unit of Measure | Percentage if between 0 and 100.  Bytes if greater than 100. | | Data Type | Double | | Minimum Value | 0 – SSAS cleaner will be allowed to clean any cached query results. | | Maximum Value | For Percentage: 100 – SSAS will use up to 100% of total physical memory.  For Bytes: 1.79E + 308 is the maximum value that can be specified. | | Requires Restart | No | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **MaxThreads** is a server property that controls how many threads will be spawned, at maximum for either query or process operations. | | Property Name | General Page: ThreadPool \ Process \ MaxThreads  General Page: ThreadPool \ Query \ MaxThreads | | --- | --- | | Default Value | 10 for Query  64 for Process | | Unit of Measure | Threads | | Data Type | Integer | | Minimum Value | 1 – A maximum of one thread is spawned. | | Maximum Value | 2147483647, although this number is not practical for threading. The practical number is determined by the operating system. | | Requires Restart | No | | Special Notes | This is server-wide setting that affects all queries and processing operations.  Regardless of the number of processors in a server, the default values apply. If your server has more than one processor, you should increase the value. | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **MinThreads** is a server property that controls the minimum number of threads that will be spawned for either query or process operations. | | Property Name | General Page: ThreadPool \ Process \ MinThreads  General Page: ThreadPool \ Query \ MinThreads | | --- | --- | | Default Value | 1 for Query and Process | | Unit of Measure | Threads | | Data Type | Integer | | Minimum Value | 1 – A minimum of one thread is spawned. | | Maximum Value | 2147483647, although this number is not practical for threading. The practical number is determined by the operating system. | | Requires Restart | No | | Special Notes | This is a server-wide setting that affects all queries and processing operations. Increase the value to make more operations run in parallel. | | Security Implications | None | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **TempDir** is a property that specifies where Analysis Server places temporary files that it uses during processing operations. | | Property Name | General Page: TempDir | | --- | --- | | Default Value | Determined during setup, but if not specified, this directory is the same as the value specified in the DataDir folder. | | Unit of Measure | N/A | | Data Type | String | | Minimum Value | N/A | | Maximum Value | N/A | | Requires Restart | Yes | | Special Notes | Analysis Services does not check the validity of the path you enter for this server property. Ensure the path exists when changing this property. | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

| **TotalMemoryLimit** is a server property that controls the total amount of physical memory that can be used by all SSAS operations. This value is expressed as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 80, which indicates that a maximum of 80 percent of the total physical memory can be used by SSAS. For more information on memory, see the “Memory” section at the beginning of this paper. The total amount of memory available to Analysis Server depends on the hardware platform Analysis Server runs on. A 32-bit operating system limits Analysis Server to maximum of 3 gigabytes (GB) with /Gb switch enabled in the boot.ini file. On a 64-bit platform, Analysis Server can address all memory available. | | Property Name | General Page: Memory \ TotalMemoryLimit | | --- | --- | | Default Value | 80 | | Unit of Measure | Percentage if between 0 and 100.  Bytes if greater than 100. | | Data Type | Double | | Minimum Value | 0 – SSAS will not use any memory for caching. | | Maximum Value | For Percentage: 100 – SSAS will use up to 100% of total physical memory.  For Bytes: 1.79E + 308 is the maximum value that can be specified. | | Requires Restart | No | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Tuning Cube and MDX Query Design

Once you have identified potential Analysis Services bottlenecks, and have made the proper adjustments to Analysis Services server properties and perhaps added more memory/CPU power to the server, the next phase is to look into tuning the OLAP cube design and the related MDX queries used against the cube.

The next few sections will give you best practice insight into fine-tuning your cube and query design, thus making your OLAP cube as efficient as possible for the queries being sent to it for results.

**Cube Design Best Practices - Dimensions**

* Consolidate multiple hierarchies into single dimension (unless they are related via fact table)
* Avoid ROLAP storage mode
* Use role playing dimensions (e.g. OrderDate, BillDate, ShipDate) - avoids multiple physical copies
* Use parent-child dimensions prudently
  + No aggregation support
* Set Materialized = true on reference dimensions
* Use many-to-many dimensions prudently
  + Slower than regular dimensions, but faster than calculations
  + Intermediate measure group must be “small” relative to primary measure group

**Cube Design Best Practices – Attributes/Hierarchies**

* Define all possible attribute relationships!
* Remove redundant attribute relationships
* Mark attribute relationships as rigid where appropriate
* Use integer (or numeric) key columns
* Set AttributeHierarchyEnabled to false for attributes not used for navigation (e.g. Phone#, Address)
* Set AttributeHierarchyOptimizedState to NotOptimized for infrequently used attributes
* Set AttributeHierarchyOrdered to false if the order of members returned by queries is not important
* Use natural hierarchies where possible

**Cube Design Best Practices – Measures**

* Use smallest numeric data type possible
* Use semi-additive aggregate functions instead of MDX calculations to achieve same behavior
* Put distinct count measures into separate measure group (BIDS does this automatically)
* Avoid string source column for distinct count measures

**Cube Design Best Practices – OLAP Partitions**

* No more than 20M rows per partition
* Specify partition slice
  + Optional for MOLAP – server auto-detects the slice and validates against user specified slice (if any)
  + Must be specified for ROLAP
* Manage storage settings by usage patterns
  + Frequently queried → MOLAP with lots of aggregations
  + Periodically queried → MOLAP with less or no aggregations
  + Historical → ROLAP with no aggregations
* Alternate disk drive - use multiple controllers to avoid   
  I/O contention
* Remote partitions for scale out – VLDB

**Cube Design Best Practices – Aggregations**

* Define all possible attribute relationships
* Set accurate attribute member counts and fact table counts
* Set AggregationUsage to guide aggregation designer
  + Set rarely queried attributes to None
  + Set commonly queried attributes to Unrestricted
* Do not build too many aggregations
  + In the 100s, not 1000s
* Do not build aggregations larger than 30% of fact table size (agg design algorithm doesn’t)

**MDX Query Design Best Practices**

* Use calculated members instead of calc cells where possible
* Use .MemberValue for calculations on numeric attributes
  + Filter(Customer.members, Salary.MemberValue > 100000)
* Avoid using CalculationPassValue
  + Rely on auto recursion resolution using scopes and assignments
* Avoid redundant use of .CurrentMember and .Value
  + (Time.CurrentMember.PrevMember, Measures.CurrentMember ).Value can be replaced with Time.PrevMember
* Avoid LinkMember, StrToSet, StrToMember, StrToValue
* Replace simple calculations with computed columns in DSV
  + Calculation done at processing time is always better
* Many more at:
  + <http://sqljunkies.com/weblog/mosha>
  + [http://sqlserveranalysisservices.com](http://sqlserveranalysisservices.com/)

Running Performance Tests

Generating MDX Client Files

Analysis Services Query Generator command line utility source code (C#) is available for your reference. No modifications to the project are necessary to work with ASQueryGenerator.exe. After you collected sample MDX queries, the next step is to generate an XML template file to feed into Query Generator. A sample template file (template.xml in solution directory) is demonstrated below:

| **XML Element and Value** | **Description** |
| --- | --- |
| <ASQueryTemplate> | Root element of the query template file |
| <Server>TargetSSASServer</Server> | Target Analysis Services server name |
| <Database>Adventure Works DW</Database> | Target database name |
| <Cube>Adventure Works</Cube> | Target cube name |
| <Queries> | Collection of queries to be executed |
| <Query>  SELECT  {[Product].[Category].Children} ON COLUMNS,  {[Geography].[Country].Children} ON ROWS  FROM [Adventure Works]  WHERE ( |CalendarYear| )  </Query> | Query template to be used for the generation of XML files  Please note a token |CalendarYear| in “WHERE” clause. Tokens can be placed anywhere in the query. Multiple tokens per query are also allowed.  Token value will be generated by the Query Generator tool directly from Analysis Services database  Example: [Date].[Calendar Year].&[2004] |
| <Tokens> | Collection of tokens for parameterized queries |
| <Token>  <Name>CalendarYear</Name>  <MinQuantity>1</MinQuantity>  <MaxQuantity>2</MaxQuantity>  <Tuple Set="[Date].[Calendar Year].Members"  OPTIONAL  File= C:\ASQueryGenerator\tuplesinfile.txt />  </Token> | <Name> is Token name without vertical bar  <MinQuantity> and <MaxQuantity> are Min and Max numbers of Tuples used to replace the Token (CalendarYear in this case)  <Tuple Set> - the Tuple to replace the Token with. The tool will connect to the Server/Database/Cube to get a list of Tuples.  OPTIONAL  If you configure optional File attribute of the <Tuple> element you can provide a specific set of Tuples to substitute for the token. Tuples will be selected from a text file with Tuples value which you need to create, for example: (contents of tuplesinfile.txt)  [Date].[Calendar Year].&[2001]  [Date].[Calendar Year].&[2004]  However, if you use the optional file notation, the “<Tuple Set” still needs to be used but can be equal to “ “. Also, keep in mind that in this case the <Server> and <Database> settings do not need to be available if files are used. |
| <RandomSeed> 3 </RandomSeed> | Seed to use in random number generation for Tuple selection |
| <ClientFile Name="C:\ASQueryGenerator\a.xml" StartingNumber="1" NumOfFiles="25" /> | Location of XML output files containing MDX queries. The setting will generate 25 files from “a1.xml” to a25.xml” |

Each query in the <Query> element of the XML input file will correspond to one query entry in XML output file. All tokens in the XML file will be replaced by their Tuple values in XML output file. The same Token can be used in multiple queries and multiple Tokens can be used in a query. To run the Query Generation tool from the command line, type in

“ASQueryGenerator.exe xml\_template\_file\_name”.

(“ASQueryGenerator.exe template.xml” in default configuration)

XML output files with MDX queries will be generated in the target directory specified in the template file. They contain the actual queries to be executed by the Load Simulator tool and will have the following format:

<Queries>

<Query>/\* 1 : 1 \*/SELECT { DESCENDANTS( [Time].[Calendar]…

</Query>

<Query>/\* 1 : 2 \*/SELECT { DESCENDANTS( [Time].[Calendar]…

</Query>

</Queries>

After you generated the XML output files, we recommend verifying some of the queries by copying them into Management Studio and executing against the target Analysis Services database. If the query structure is correct, any query from the XML file executed from the Management Studio should produce a correct result set.

Configuring Load Simulator

Load Simulator tool can be launched either from Visual Studio 2005 or from command line using MSTest.exe utility. Visual Studio for Testers offers rich interactive way of controlling test parameters and monitoring the server performance during the tests. Once the desired test configurations are known, the load simulation tests can be run in command line mode to make the testing more effective. Setting up the environment for stress tests is a two step process

* configuring the target component (Load Simulator – ASWorkloadtool.dll)
* configuring Visual Studio by means of Load Tests to actually run the tests

Before configuring Visual Studio on the client machines you need to create a SQL Server database which holds the results of the test run. We recommend using a separate server for collecting this data. Run the “loadtestresultrepository.sql” script on the target SQL Server designated for collection of testing information. This script creates a “LoadTest” database. Next, review the Load Simulator configuration file configuration.xml in the “ASWorkloadTool” directory and modify it according to your environment.

Configuration file for the Analysis Services Load Simulator is described below:

| **XML Element and Value** | **Description** |
| --- | --- |
| <Server>TARGETSSASServer</Server> | Target Analysis Services Server |
| <Database>AdventureWorksAS</Database | Target Analysis Services Database |
| <ClientFileName>  C:\ASQueryGenerator\a.xml  </ClientFileName> | Location of XML files with MDX queries for load testing |
| <ClientFileStartingNumber> 1  </ClientFileStartingNumber> | Starting number of the XML file |
| <NumOfClientFiles>25</NumOfClientFiles> | Total number of XML files being used. |
| <ThinkTimeMin>15</ThinkTimeMin> | After executing a query, the Load Simulator pauses for at least this number of seconds before executing the next one. |
| <ThinkTimeMax>30</ThinkTimeMax> | Maximum Number of seconds for the pause between query executions. |
| <DataAccessMethod>Sequential</DataAccessMethod> | Drives the order of query execution. Two access methods are supported Sequential – all queries are run in sequence as determined by XML file Random – queries are randomly selected. In both cases every query is run only once. |
| <LogToServer>SQLLogSrv</LogToServer> | Name of the SQL Server for test results |
| <LogToDB>LoadTest</LogToDB> | Name of the database containing the test results data. |

We successfully configured the Load Simulator.

Configuring Visual Studio for Load Test Run

After configuring the Load Simulator, we need to configure the Visual Studio to run Load Tests using the Load Simulator dll. Open the “ASWorkloadTool.sln” solution and then the file LoaderPlugin.cs (in ASWorkloadTool project). Make sure that the variable m\_configFileName points to the correct name and location of the configuration file. Now set the project to “Release” mode and build it to produce ASWorkloadTool.dll. After we updated the assembly, we are ready to configure launching Load Simulator from the Visual Studio.

Before we create a Load Test, we need to specify the Test Results Store. To do that, select “Test > Administer Test Controller” from the top level menu of the Visual Studio. In the “Load Test Results Store” property, click the ellipsis button and connect to the SQL Server database on the remote server which has the “LoadTest” database mentioned in the previous section. The same server and database are used in the configuration.xml file mentioned in the section above.

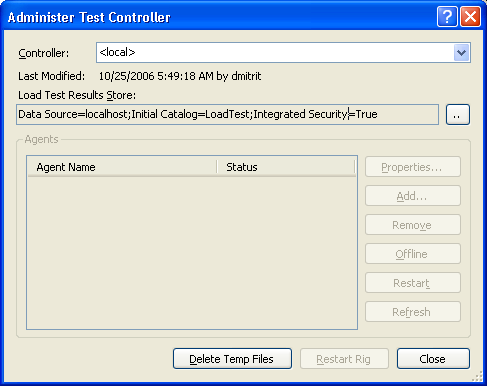


Figure 6. VS.NET 2005 Test Controller

Now we are ready to proceed with configuration of Visual Studio Load Tests. The “ASWorkloadTool.sln” solution comes with two load test configurations LoadTest1.loadtest and LoadTest2.loadtest. You can modify these load tests or create a new one. In order to create a new load test, first right-click on ASWorkloadTool project and select “Add > Load test”. This opens up a New Load Test Wizard.

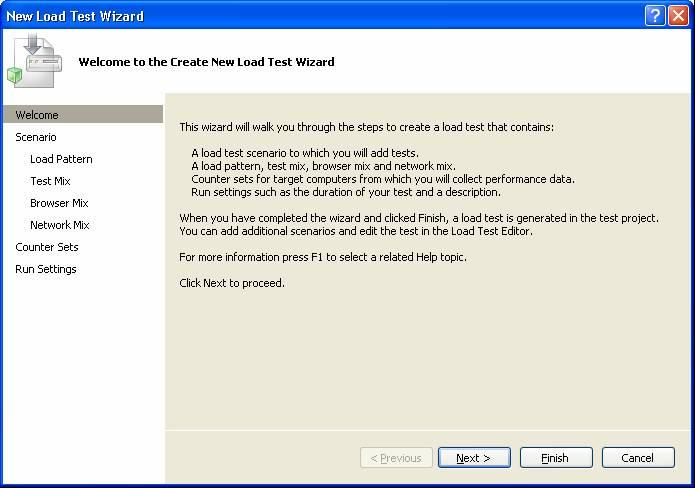


Figure 7. Load Test Wizard

Click Next and enter a name for the load test scenario, say “Scenario1”. Ignore “Think time between test iterations” field and click “Next”. In the tab “Load Pattern” enter the number of users which correspond to the name of XML files you generated. For example, if you enter 100 make sure you have at least 100 XML files pre-generated. In the tab “Test Mix” click “Add” Select “LoadStart” from the left “Available tests” window and add it to the right “Selected tests” window. Click “Next”

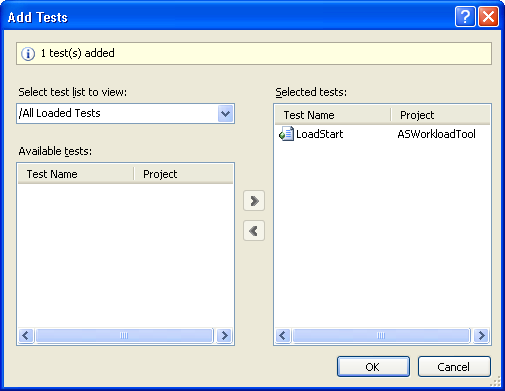


Figure 8. Adding Tests to the Load Simulation

In the tab “Counter Sets” add the desired counters you want to monitor and specify on which machine. Click “Next >”.

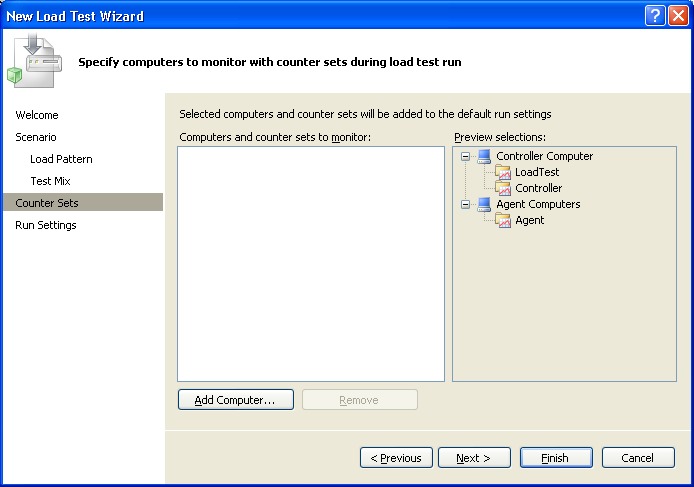


Figure 9. Load Test Counters

In the tab “Run Settings” specify the run duration and warm-up duration as appropriate. Keep in mind that if the all the MDX queries from XML files may get executed prior to the warm-up duration plus run duration. Make sure you have enough queries to use in your test cases. In addition, if you make the warmup period too long, very few queries will be left over for the actual test run and the test results may be skewed. Enter some information into the “Description” field to help the interpretation of the results. Complete the Wizard by clicking “Finish”. Now you created a brand new Load Test (say LoadTest3.loadtest). Behind the scenes you just generated an XML file (LoadTest3.loadtest) which can be edited by either Visual Studio in visual mode, or as a text file using any other authoring tool.

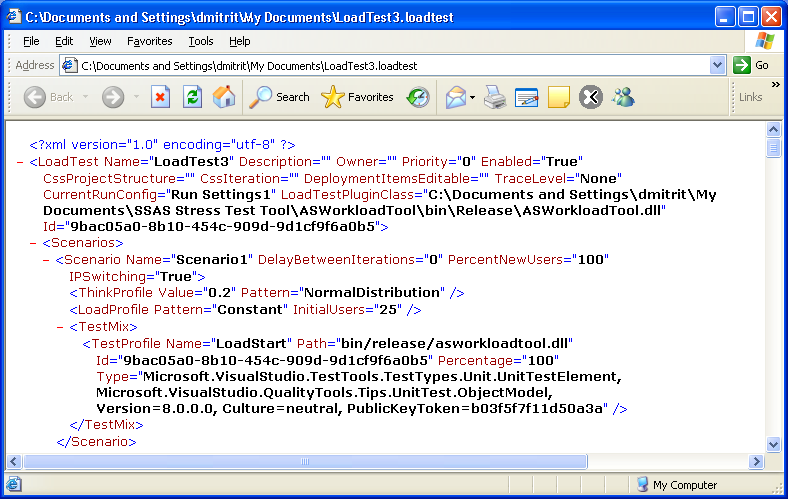


Figure 10. Load Test XML File

It is important to make sure that the ASWorkloadTool.dll is being used as Load Test Plug-In. In Load Test Editor, select the root node in the load test tree. Open the “Properties” window, find the “Load Test Plug-In” property and enter the string value for the path to ASWorkloadTool.dll.

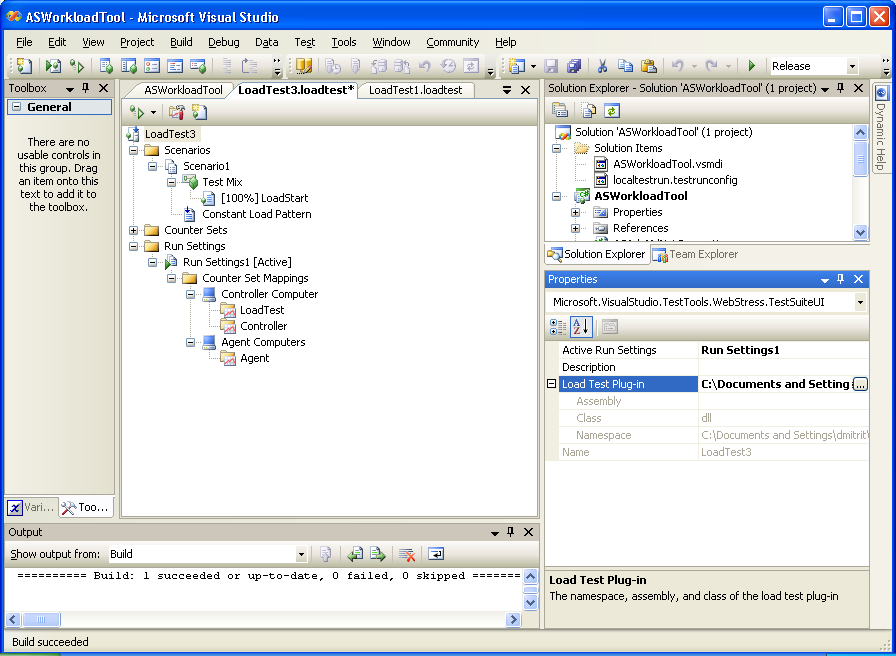


Figure 11. Specifing Location for ASWorkloadTool.dll

In Load Test editor, select the active Run Settings. Open its Properties window. Change property “Timing Details Storage” to “AllIndividualDetails”. This will provide detailed information about query execution to the reporting SQL Server database.

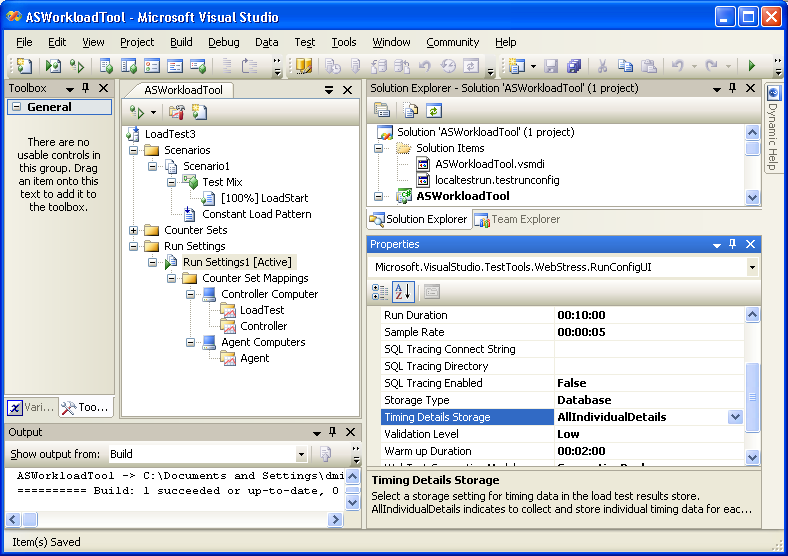


Figure 12. Timing Details Storage Setting

In the “Build” menu of the Visual Studio click “Build Solution”. Now we are ready to click the green arrow in the load test editor to launch the test run.

Instead of generating a new Load Test, we can modify the settings of the existing one. Click on existing “Loadtest1.loadtest” and then on “Constant Load Pattern” If you want to simulate 100 concurrent users, make sure you have 100 XML files with queries prepared and set “Constant User Count” property to 100.

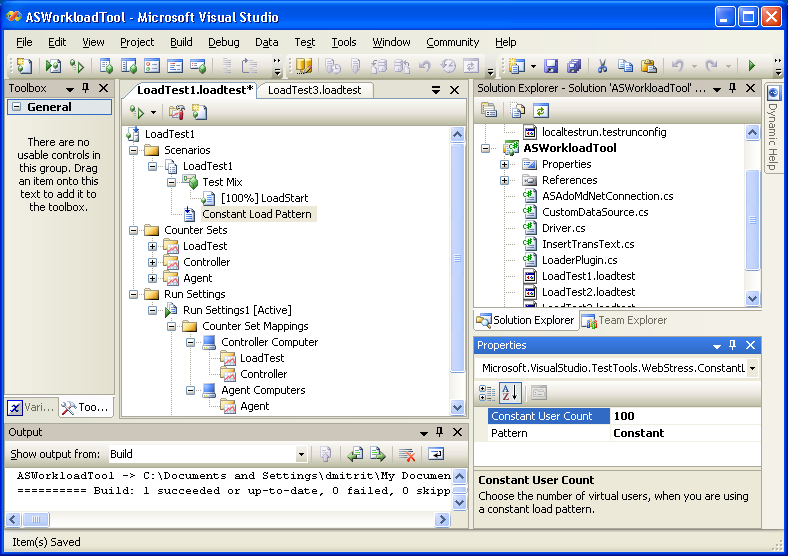


Figure 13. Constant User Count Property Setting

Next, click on “Run Settings 1 [Active]” and in Properties window update the property “Run Duration” to 00:10:00 which means 10 minutes. Also consider updating the “Warm up duration” property if you want to simulate the warm-up process.

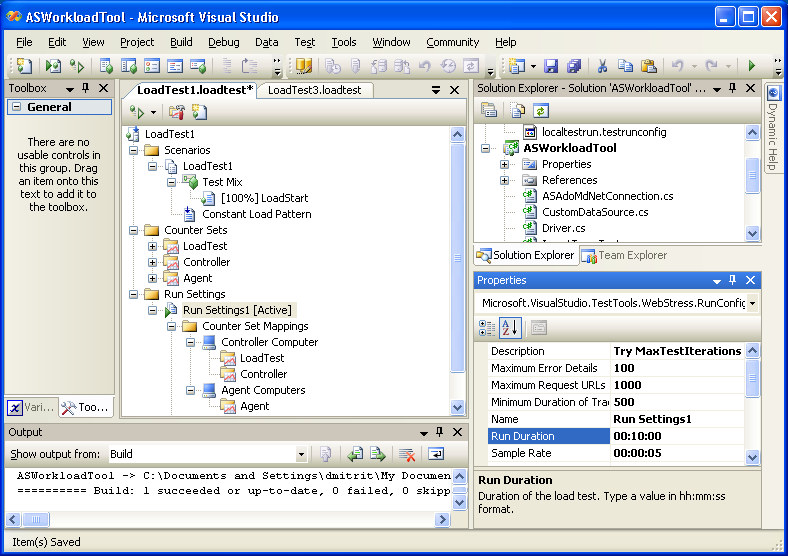


Figure 14. Run Duration Property Setting

Running Stress Tests

When you click on the green triangle to run the load test, Visual Studio will load the test parameters and the target assembly (ASWorkloadTool.dll) which in turn read the queries from XML files and send the requests to the target Analysis Services Server. You will be able to monitor the rest progress through Visual Studio.

*Note that there are various places from which you can run a Test from e.g. Test Manager. If you are not familiar with Visual Studio 2005 for Testers, you may end up incorrectly starting tests from the normal Visual Studio menu, which could result in issues with your test scenario.*

In some situations, you may want to run through client queries just once. In order to do that, follow the steps listed below:

• Right click the “Active Run Settings” node in your load test

• Select “Add Context Parameter” – you will see the property sheet

• Type “MaxTestIterations” as Parameter Name

• Type an integer (larger than 0) as its Value

In addition to using Visual Studio, you can run some of the tests from command line using MSTest.exe utility. First, you need to add MSTest.exe location (“%ProgramFiles%\Microsoft Visual Studio 8\Common7\IDE”) to PATH environmental variable. The syntax for running MSTest is following:

start mstest /runconfig:localtestrun.testrunconfig /testmetadata:ASWorkloadTool.vsmdi /testlist:"Load Runs"

* localtestrun.testrunconfig is the test run configuration file,
* ASWorkloadTool.vsmdi is the test metadata file
* “Load Runs” is the name of the test list to be executed

To create a test list (collection of tests) double click on “ASWorkloadTool.vsmdi” metadata file. Then right click on “Lists of Tests” and click “New Test List…” to create a test list. Then drag the load tests to the newly created test list. You can add more than one load test to the test list.

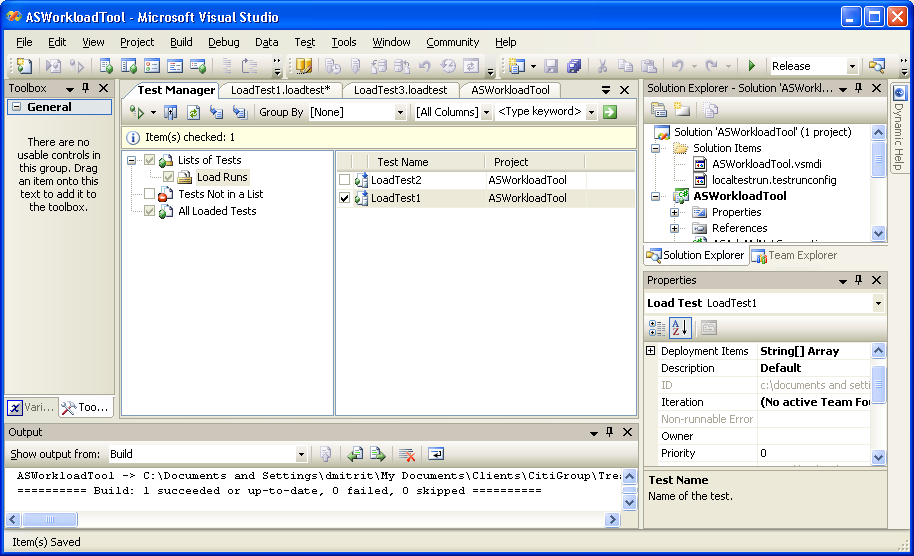


Figure 15. Executing Load Test

Reporting Performance Test Results

Once you have completed your stress test, the next step will be to see the results of those test runs. All of the results of the test, as well as the counters captured during the test have been recorded in the stress test database (default name of LoadTest). Analysis of the results and counters can be achieved using either direct query against the database, or by using the included SQL Server Reporting Services reports.

Direct Query of the Database

Perhaps the most direct method of retrieving stress test results is simply to run queries against the SQL Server database used by Analysis Services Load Simulator (using SQL Server Management Studio). The difficultly in running direct queries against the database is that you need an understanding of the underlying database tables and their relationships. Therefore, the table and diagram below are to assist when creating your queries. You should also refer to the table/column definitions of the LoadTest database in the appendix of this document.

| **LoadTest Table Names** |
| --- |
| 1. LoadTestPerformanceCounterCategory 2. LoadTestPerformanceCounterInstance 3. LoadTestPerformanceCounterSample 4. LoadTestRunAgent 5. LoadTestRunInterval 6. LoadTestScenario 7. WebLoadTestErrorDetail 8. WebLoadTestRequestMap 9. WebLoadTestTransaction 10. LoadTestSqlTrace 11. LoadTestBrowsers 12. LoadTestNetworks 13. LoadTestTestDetail 14. LoadTestPageDetail 15. LoadTestTransactionDetail 16. LoadTestTestSummaryData 17. LoadTestTransactionSummaryData 18. LoadTestPageSummaryData 19. LoadTestPageSummaryByNetwork 20. WebLoadTestTransactionContent 21. LoadTestRun 22. LoadTestCase 23. LoadTestMessage 24. LoadTestThresholdMessage 25. LoadTestPerformanceCounter |

| Figure 16. LoadTest Database Model Constructed from Visio 2003 |
| --- |

As an example of how to use the data within the LoadTest database, the following are a few example queries to leverage after connecting to the database through Management Studio. The main thing to consider when quering the database is that the “loadtestrunid” is critical to returning the correct test run data.

| **Query Reason** | **Query Syntax** |
| --- | --- |
| Get last load test run identifier. | select max(loadtestrunid) from LoadTestCase |
| Shows the performance counters logged in the load test run number twenty (20). | select \* from loadtestperformancecountercategory where loadtestrunid = 20 |
| List the average, maximum, minimum, percentile90, percentile95 value for each counter during the load test run number twenty (20). | select  lttrans.loadtestrunid, lttrans.transactionid, wltrans.transactionname,  lttrans.transactioncount, lttrans.average, lttrans.minimum, lttrans.maximum, lttrans.percentile90, lttrans.percentile95  from loadtesttransactionsummarydata lttrans  inner join webloadtesttransaction wltrans  on lttrans.transactionId = wltrans.transactionId  where lttrans.loadtestrunid = 20 and wltrans.loadtestrunid = 20 |
| Show the details of each transaction in the test run twenty (20). | select lttrans.loadtestrunid, lttrans.transactionid, wltrans.transactionname,  lttransdetail.timestamp, lttransdetail.elapsedtime  from loadtesttransactionsummarydata lttrans  inner join webloadtesttransaction wltrans  on lttrans.transactionid = wltrans.transactionid  left outer join loadtesttransactiondetail lttransdetail  on lttrans.transactionid = lttransdetail.transactionid  where lttrans.loadtestrunid = 20 and wltrans.loadtestrunid = 20 and lttransdetail.loadtestrunid = 20  order by lttransdetail.timestamp asc |

Leveraging Reporting Services

SQL Server Reporting Services is a comprehensive, server-based solution that enables the creation, management, and delivery of both traditional, paper-oriented reports and interactive, Web-based reports. An integrated part of the Microsoft business intelligence framework, Reporting Services combines the data management capabilities of SQL Server and Microsoft Windows Server with familiar and powerful Microsoft Office System applications to deliver real-time information to support daily operations and drive decisions.

Using Reporting Services against the LoadTest database, there are four included designed reports that summarize and detail the simulation test results. The following table lists the available reports. (Keep in mind that these are sample reports off the database, and modifications and/or new reports can easily be designed with knowledge of the underlying database.)

| **Report Description** | **Report Name** | |
| --- | --- | --- |
| Lists all load test runs that have been executed. | AllLoadTests.rdl | |
| Summarizes data captured in one load test run. It lists average, minimum, maximum, Percentile90, Percentile95 data about each transaction. | LoadTestSummary.rdl | |
| Lists information about execution of each transaction, For example: when did they execute, and their elapsed time. | TransactionDetail.rdl | |
| Shows the error information returned to each transaction execution. | ErrorDetail.rdl | |
|  | | |

Figure 17. Screen Capture of “Load Test Runs” Report

Considerations for Multiple Analysis Services Databases

Generation of MDX Client Files for Multi SSAS DB Environment

In the scenario described above, each Load Simulator tool was configured to send the MDX queries against a single Analysis Services Server, a specified database and cube. The target server name, database name and cube name are defined in the configuration file. If you have a server which contains multiple Analysis Services databases or multiple cubes in a single database which you would like to test simultaneously, you can still use the Load Simulator “as is”. The simplest approach is to create multiple instances Load Simulator and its entire environment (together with MDX client files) on the same workstation. Each instance of Load Simulator will be run independently, use its own configuration file and its own set of XML files with MDX queries. Here are the steps to configure the Load Simulator for a multi database load tests on a single workstation:

1. Generate several sets of XML files with MDX queries so each Load Simulator instance will use its own copy of XML files.
2. Copy the Load Simulator project multiple times on the same workstation. Each project will contain its own Load Simulator library and configuration file targeting it to a specific Server/Database/Cube
3. Rebuild the projects accordingly (make sure the output assembly uses its own configuration file – configuration file path and name is hard coded in the ASworkloadTool.dll in this version of the tool).
4. Create a script which will execute MSTest.exe for each Load Simulator instance.

First, you need to run Query Generator tool several times to generate several sets of MDX query files for each individual SSAS Database and Cube. For example, if need to generate 10-user load on SSAS Server which has:

* SSAS Database: TargetA
* Cube: CubeA

you will generate XML files MDXSourceA1.xml, MDXSourceA2.xml ... MDXSourceA10.xml to prepare the queries for 10 users. These XML files can be deployed in their own directory to be used by the Load Simulator A.

Next, for example, you need to generate queries for the 20 user load on another database on the same server:

* SSAS Database: TargetB
* Cube: CubeB

The Query Generator will produce a set of 20 query files MDXSourceB1.xml, MDXSourceB2.xml … MDXSourceB20.xml. These XML files will be used by the Load Simulator B which is a separate instance.

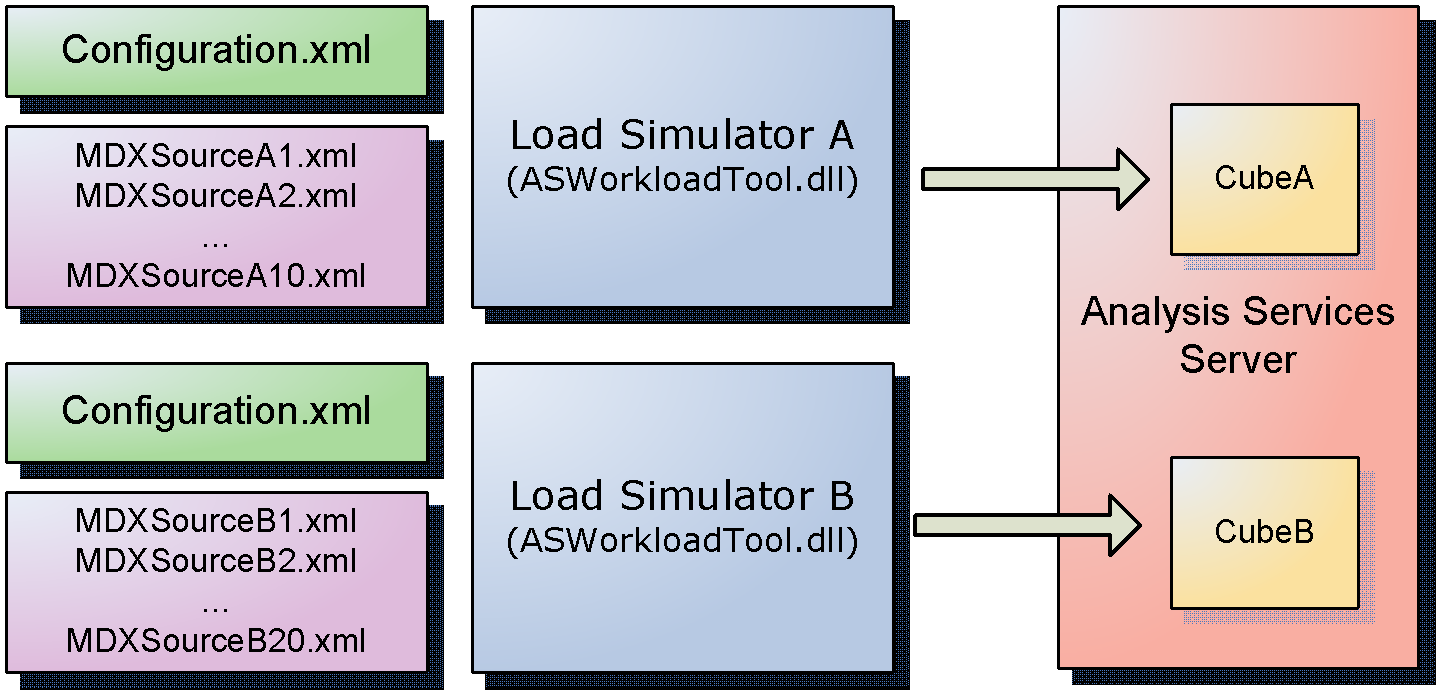


Figure 18. Configuration of multi-database environment

If you are testing multiple cubes with the same structure, the creation of MDX files for several instances of Load Simulator can be simplified. In this case, you can just re-use the same XML files by copying them in locations specified by each instance of Load Simulator configuration file. Every instance of Load Simulation tool needs its own set of XML files which are not shared with other Load Simulator instances. If load files are identical (for cubes with the same structure), make sure <DataAccessMethod> property value is set to “Random” to assure a realistic workload on multiple databases.

Executing Stress Tests for Multi SSAS DB Environment

Each workstation can accommodate several Load Simulator instances running multiple simultaneous users. However, the maximum number of users supported in multi-instance scenario is close to that of single-instance scenario and was approximately 300 in our tests. In order to automate the testing process, you can create script files which will launch multiple instances of MSTest.exe each running an instance of Load Simulator.

| cd ASWorkloadTool\1  start mstest /runconfig:localtestrun.testrunconfig /testmetadata:ASWorkloadTool.vsmdi /testlist:"List1-Baseline"  cd ..\2  start mstest /runconfig:localtestrun.testrunconfig /testmetadata:ASWorkloadTool.vsmdi /testlist:"List1-Baseline"  cd ..\3  start mstest /runconfig:localtestrun.testrunconfig /testmetadata:ASWorkloadTool.vsmdi /testlist:"List1-Baseline" |
| --- |

ASWorkloadTool is the directory containing subdirectories 1,2,3 for each instance of the Load Simulator tool. Keeping track and updating large numbers of XML configuration files can be done in many different ways, for example using SSIS or XSL transforms.

Reporting Performance Test Results for Multi SSAS DB Environment

Each instance of Load Simulator can have its own reporting database as described above. However, it may be more practical to use one centralized repository for collecting all client load simulation information. This is because separate instances of Load Simulator can run at the same time as a part of the same test scenario. To provide effective reporting on the data, you can configure the “Description” property of the load test file to incorporate the Load Test instance information. This can be accomplished either via Visual Studio (as shown below) or by editing appropriate “.loadtest” files.

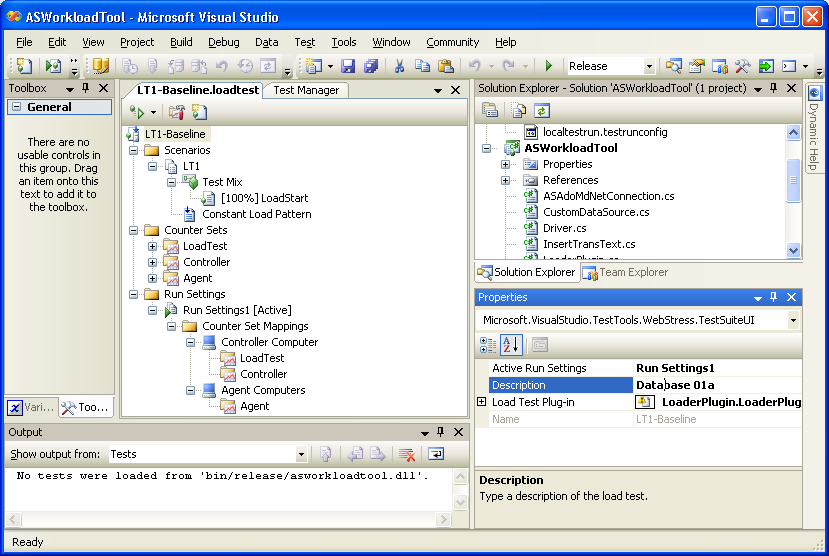


Figure 19. Configuration of Description for reporting purposes in multi-database environment.

Conclusion

Now that you have a basic understanding of how to use the Load Test tool with Analysis Services, you can build upon that knowledge by creating and running tests on configurations used in your organization.

In most cases, you will want to run a load test on different computer than the one used to run the report server. Additionally, to mimic actual user activity, configure multiple user sessions across multiple computers. This configuration is called controller-agent configuration and the computer that runs the load test is the controller. The computers that host the user sessions are created as agents. Controller-agent configurations are beyond the scope of this white paper, but if you want to learn more about controllers, agents, and Microsoft Visual Studio 2005 Team Edition for Software Testers, see the following links on MSDN:

For more information about how to set up controller and agent functionality, see [Setting Up the Controller and Agent Functionality](http://go.microsoft.com/fwlink/?LinkId=70268).

To learn more about controllers and agents, see [Working with Controllers, Agents, and Rigs](http://go.microsoft.com/fwlink/?LinkId=70267).

For more information about Microsoft Visual Studio 2005 Team Edition for Software Testers, see [Team Edition for Testers](http://go.microsoft.com/fwlink/?LinkId=70266).

For more information about Analysis Services server performance and how to report on load test results so that you can compare results over time and on different configurations, see [Planning for Scalability and Performance with Reporting Services](http://go.microsoft.com/fwlink/?LinkId=70650) and [Share the load: Report Visual Studio Team System Load Test Results via A Configurable Web Site](http://go.microsoft.com/fwlink/?LinkId=70265).

Appendix - What's in the Visual Studio Team System Load Test Results Store

Table: **LoadTestRun**: Contains one row for each load test that is executed.

Columns:

* **LoadTestRunId (int):** A unique ID for this load test run in this load test results store.   The value is 1 for the first load test and increments for each load test that is executed.   All the tables in the load test results store use the LoadTestRunId to identify the load test run to which the data belongs.
* **LoadTestName (nvarchar(255)):** The name of the load test.
* **RunId (char (36))**: A GUID that identifies the test run in which the load test was run.
* **Description (nvarchar (255)):** A description of the load test from the Analysis dialog in the Load Test Results Viewer.
* **StartTime (datetime):** The time at which the load test started.
* **EndTime (datetime):** The time at which the load test completed or was stopped by the user.  EndTime is NULL while the test is in progress.
* **RunDuration (int):** The load test run duration (in seconds) that was specified in the active Run Settings for the load test.
* **WarmupTime (int):** The warm up time (in seconds) that was specified in the active Run Settings for the load test.
* **RunSettingUsed (nvarchar (32)):** The name of the active Run Settings at the time the load test was run.
* **IsLocalRun (bit):** Indicates whether the load test was run locally or on a rig, that is controller & agent(s):. Values are as follows: 1 for local; 0 for rig.
* **ControllerName (nvarchar (255)):** Name of the controller computer (or local computer name for a local run).
* **Outcome (nvarchar (32)):** The outcome of the load test, which will be one of the following values: "InProgress", "Completed", "Aborted", or "Error".
* **LoadTest (image):** An encoded version of the LoadTest that was run.
* **Comment (ntext):** A potentially long comment about the load test from the Analysis dialog in the Load Test Results Viewer.

Table: **LoadTestCase**: Stores data on the inner tests that were included in the load test (one row per inner test).

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **ScenarioId (int)**: The unique ID of the Scenario that contains the test (from the LoadTestScenario table).
* **TestCaseId (int):** A unique ID for the test within this load test run.
* **TestCaseName (nvarchar (64)):** The name of the test.

Table: **LoadTestMessage**: Stores data for each error that occurs during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **AgentId (int):** The unique ID of the load test agent on which the error occurred (-1 for errors not specific to an agent).
* **MessageId (int):** A unique ID for the message on the agent during this load test run.
* **MessageType (tinyint):** An integer that encodes the type of error; the values used and their meanings. Values are as follows: 0: TestError; 1: Exception, 2; HttpError; 3: ValidationRuleError; 4: ExtractionRuleError; 5: Timeout.
* **MessageText nvarchar (2048):** The error message text.
* **SubType (nvarchar (64)):** The subtype of the error.  The values for this vary, depending on the MessageType.  For example, if MessageType is Exception, SubType is the name of the Exception class.
* **StackTrace (ntext):** A stack trace for the code location where the error occurred (used when MessageType is Exception or TestError).
* **MessageTimeStamp (datetime):** The time at which the error occurred.
* **TestCaseId (int):** The unique ID of the inner test (from the LoadTestCase table) in which the error occurred, or NULL for errors not associated with a specific inner test, such as performance counter collection errors.
* **RequestId (int):** The unique ID of the Web test request (from the WebLoadTestRequestMap table) on which the error occurred. For errors not associated with Web test request, the value is NULL.

Table: **LoadTestThresholdMessage:** Stores one row of data for each threshold rule violation.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run
* **TestRunIntervalId (int):** ID of the collection interval during which the threshold rule violation occurred.
* **CounterInstanceId (int):** ID of the performance counter instance that had the threshold rule violation.
* **MessageId (int):** Unique ID of the threshold rule violation within this load test run.
* **MessageText (nvarchar (2048)):** Text describing the threshold rule violation.

*Overview of performance counter tables*

        The metadata for all the performance counters collected during a load test is stored in three related tables: LoadTestPerformanceCounterCategory, LoadTestPerformanceCounter, and LoadTestPerformanceCounter.

        This metadata includes all four components of performance counter identifier: computer name, category name, counter name, and instance name.

       These three tables store only metadata about the performance counters; the sampled performance counter values are stored in LoadTestPerformanceCounterSample.

Table: **LoadTestPerformanceCounter**: Stores one row for each performance counter that is collected during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **CounterCategoryId (int):** The CounterCategoryId from the LoadTestPerformanceCounterCategory row for the category that contains this counter name.
* **CounterId (int):** The unique ID of this performance counter for this load test run.
* **CounterName (nvarchar (255)):** The name of the performance counter (for example "Available MBytes").

Table: **LoadTestPerformanceCounterCategory**: Stores one row for each performance counter category and computer on which the category is collected.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **CounterCategoryId (int):** The unique ID of this performance counter category for this load test run.
* **CategoryName (nvarchar (255)):** The name of the performance counter category (for example "Memory").
* **MachineName (nvarchar (255)):**  The name of the computer on which the category was collected.
* **StartTimeStamp100nSec (bigint):** A timestamp that indicates when the category was first collected during the load test.

Table: **LoadTestPerformanceCounterInstance**: Stores one for each perforce counter instance that is collected during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **CounterId (int):** The CounterId from the LoadTestPerformanceCounter table for the performance counter of which this is an instance.
* **InstanceId (int):** Unique ID of this performance counter instance in this load test run.
* **LoadTestItemId (int):** For performance counters in the "LoadTest:\*" categories, this
* **InstanceName (nvarchar (255)):** The name of the performance counter instance, for example "\_Total".
* **CumulativeValue (real):** The final calculated value for this performance counter instance over the entire load test run.
* **OverallThresholdRuleResult (tinyint):** Indicates the most severe threshold rule result for this performance counter instance during the load test. Values are 0 for Ok; 1 for Warning; 2 for Critical.

Table: **LoadTestPerformanceCounterSample**: Stores the performance counter samples.  There is sample for each performance counter instance collected for each sampling interval during the load test. Many of the columns in this table are fields of the .Net class System.Diagnostics.CounterSample, which is used internally by the load test implementation. Their values do not make much sense outside that context.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TestRunIntervalId (int):** The sequence number of the collection interval in which this sample was collected.
* **InstanceId (int):** The InstanceId from the LoadTestPerformanceCounterInstance table row that contains the performance counter instance of which this is a sample.
* **ComputedValue (real):** A value computed for the performance counter based on this sample and the previous sample. If there was no sample in the previous interval, the value is NULL.
* **RawValue (bigint):** The RawValue field of the System.Diagnostics.CounterSample value.
* **BaseValue (bigint):** The BaseValue field of the System.Diagnostics.CounterSample value.
* **CounterFrequency (bigint):** The CounterFrequency field of the System.Diagnostics.CounterSample value.
* **SystemFrequency (bigint):** The SystemFrequency field of the System.Diagnostics.CounterSample value.
* **SampleTimeStamp (bigint):** The SampleTimeStamp field of the System.Diagnostics.CounterSample value.
* **SampleTimeStamp100nSec (bigint):** The SampleTimeStamp100nSec field of the System.Diagnostics.CounterSample value.
* **CounterType (int):** The PerformanceCounterType field of the System.Diagnostics.CounterSample value.
* **ThresholdRuleResult (tinyint):** The result of any threshold rule applied to this sample: 0 for Ok, 1 for Warning, 2 for Critical.
* **ThresholdRuleMessageId (int):** The MessageId from the LoadTestThresholdMessage row with more details on any threshold violation thrown for this sample. NULL If there was no threshold violation, the value is NULL.

Table: **LoadTestRunAgent**: Stores the names of the agent machines used for a load test run.

Columns:

* **LoadTestRunId** (int): The unique ID of the load test run.
* **AgentId (int):** A unique ID of this agent during this load test run.
* **AgentName (nvarchar (255)):** The computer name of the agent.

Table: **LoadTestRunInterval**: Stores one row for each performance counter collection interval during the load test run.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TestRunIntervalId (int):** The unique Id of this interval in this load test run.
* **IntervalStartTime (datetime):** The start time of the collection interval .
* **IntervalEndTime (datetime):** The end time of the collection interval.

Table: **LoadTestScenario:** Stores the names of the Scenarios defined for a load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **ScenarioId (int)**: The unique ID of the Scenario within this load test.
* **ScenarioName (nvarchar (64)):** The name of the Scenario.

Table: **WebLoadTestErrorDetail**: Stores detailed data about a Web test request when there is an error associated with a Web test request.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **AgentId (int):** The ID of the agent on which the error occurred.
* **MessageId (int):** The MessageId from the row in the LoadTestMessage table for the error with which this error detail is associated.
* **WebTestRequestResult (image):** A binary encoded version of the object that stores all the data captured regarding the result of a Web test request.

Table: **WebLoadTestRequestMap**: Stores one row for each unique Uri to which a Web test request is sent during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **RequestId (int):** The unique ID of this request in this load test run.
* **TestCaseId (int):** The Id of the Web test that generated this request.
* **RequestUri (nvarchar (2048)):** The Uri for the request.   This excludes the query string parameters except for those query string parameters with the property "

Table: **WebLoadTestTransaction**: Store the names of all transactions defined in the load test. This includes transactions defined in Web tests, and transaction defined in unit tests using the BeginTimer()/EndTimer() methods.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TransactionId (int):** The unique ID of the transaction in this load test run.
* **TestCaseId (int):** The TestCaseid from the LoadTestCase table for the test that defined the transaction.
* **TransactionName (nvarchar (64)):** The name of the transaction.

Table: **LoadTestSqlTrace**: Stores collected when performing SQL tracing during a load test; the field corresponds to SQL trace fields.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TextData (ntext):** The text of the SQL operation.
* **Duration (bigint):** The duration of the SQL operation.
* **StartTime (datetime):** The start time of the SQL operation.
* **EndTime (datetime):** The end time of the SQL operation.
* **Reads (bigint):** The number of reads that occurred while the SQL operation was performed.
* **Writes (bigint):** The number of writes that occurred while the SQL operation was performed.
* **CPU (int):** The CPU time required to perform the SQL operation.
* **EventClass (int):** An integer identifying the type of SQL operation traced.

Table: **LoadTestBrowsers**: Stores the names of all browser types used during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **BrowserId (int):** The unique ID of this browser.
* **BrowserName (nvarchar (255)):** The name of the browser.

Table: **LoadTestNetworks**: Stores the names of all network types used during the load test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **NetworkId (int):** The unique ID of the network type.
* **NetworkName (nvarchar (255)):** The name of the network type.

*Tables used to store Timing Details*

All the following tables only have data stored in them when the "Timing Details Storage" option on the Load Test Run Settings is set to a value other than "None."

Table: **LoadTestTestDetail:** Stores the results of individual inner tests; there is one row for each test completed during the load test.

Columns:

* **LoadTestRunId (int)**:The unique ID of the load test run.
* **TestDetailId (int):** A unique ID for this row in the load test run.
* **TimeStamp (datetime):** The time that the test completed.
* **TestCaseId (int):** The TestCaseId from the LoadTestCase table for the row that identifies the test.
* **ElapsedTime (float):** The elapsed time (in seconds) to run the test.
* **AgentId (int):** The AgentId from the LoadTestAgent table row that identifies the agent on which the test was run.
* **BrowserId (int):** The BrowserId from the LoadTestBrowser table that identifies the browser type used for a Web test.
* **NetworkId (int):** The NetworkId from the LoadTestNetwork table that identifies the network type used for a Web test.

Table: **LoadTestPageDetail**: Stores timing results of individual Web test pages; there is one row for each Web test page completed during the load test.

Columns:

* **LoadTestRunId (int)**: The unique ID of the load test run.
* **PageDetailId (int)**: A unique ID for this row in the load test run.
* **TestDetailId (int):** The TestDetailId from LoadTestTestDetail row with the result for the test that contains this page.
* **TimeStamp (datetime)**: The time at which the page completed.
* **PageId (int):** The RequestId from the WebLoadRequestMap table (join on this to identify the Uri for the page).
* **ResponseTime (float):** The response time (in seconds) for the page.
* **ResponseTimeGoal (float):** The response time goal (in seconds) for the page.
* **GoalExceeded (bit):** Boolean value that indicates whether or not the response time goal was exceeded by the response time.

Table: **LoadTestTransactionDetail**

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TransactionDetailId (int):** A unique ID for this row in the load test run.
* **TestDetailId (int):** The TestDetailId from LoadTestTestDetail row with the result for the test that contains this transaction.
* **TimeStamp (datetime):** The time at which the transaction completed.
* **TransactionId (int):** The TransactionId from the WebLoadTestTransaction table (join on this to identify the transaction name.)
* **ElapsedTime (float):** The elapsed time (in seconds) to complete the transaction.

Table: **LoadTestTestSummaryData**: Summarizes the detailed Test result data that is stored in LoadTestTestDetail; there is one row for each test in the load test run which rolls up the results for that test.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **TestCaseId (int):** The TestCaseId from the LoadTestCase table for the row that identifies the test.
* **TestsRun (int):** The number of tests completed.
* **Average (float):** The average time (in seconds) to run the test.
* **Minimum (float):** The minimum time (in seconds) to run the test.
* **Maximum (float):** The maximum time (in seconds) to run the test.
* **Percentile90 (float):** The 90% percentile test time (90% of the tests completed in less than this time).
* **Percentile95 (float):** The 95% percentile test time (95% of the tests completed in less than this time).

Table: **LoadTestTransactionSummaryData**: Summarizes the detailed transaction data that is stored in LoadTestTransactionDetail; there is one row for each transaction name used during the load test run which rolls up the results for that transaction.

Columns:

* **LoadTestRunId (int)**: The unique ID of the load test run.
* **TransactionId (int):** The TransactionId from the WebLoadTestTransaction table (join on this to identify the transaction name) .
* **TransactionCount (int):** The number of tests completed.
* **Average (float):** The average time (in seconds) to run the transaction.
* **Minimum (float):** The minimum time (in seconds) to run the transaction.
* **Maximum (float):** The maximum time (in seconds) to run the transaction
* **Percentile90 (float):** The 90% percentile transaction time (90% of the transactions completed in less than this time).
* **Percentile95 (float):** The 95% percentile transaction time (95% of the transactions completed in less than this time).

Table: **LoadTestPageSummaryData**: Summarizes the detailed page data that is stored in LoadTestPageDetail; there is one row for each page Uri invoked during the load test run which rolls up the results for that page.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **PageId (int):** The RequestId from the WebLoadRequestMap table (join on this to identify the Uri for the page).
* **PageCount (int):** The number of times this page was completed .
* **Average (float):** The average time (in seconds to run the page. )
* **Minimum (float):** The minimum time (in seconds to run the page.
* **Maximum (float):** The maximum time (in seconds) to run the page.
* **Percentile90 (float)**: The 90% percentile page time (90% of the pages completed in less than this time.
* **Percentile95 (float):** The 95% percentile page time (95% of the pages completed in less than this time).

Table: **LoadTestPageSummaryByNetwork**: Summarizes the detailed page data that is stored in LoadTestPageDetail by network type; there is one row for each page Uri for each network type invoked during the load test run, which rolls up the results for that page on each network type.

Columns:

* **LoadTestRunId (int):** The unique ID of the load test run.
* **PageId (int)**: The RequestId from the WebLoadRequestMap table (join on this to identify the Uri for the page.
* **NetworkId (int)**: The NetworkId from the LoadTestNetwork table that identifies the network type used for a Web test.
* **PageCount (int)**: The number of times this page was completed (on the network type identified by NetworkId).
* **Average (float)**: The average time to run the page (in seconds) on the network type identified by NetworkId.
* **Minimum (float)**: The minimum time to run the page (in seconds) on the network type identified by NetworkId.
* **Maximum (float)**: The maximum time (in seconds) to run the page on the network type identified by NetworkId.
* **Percentile90 (float)**: The 90% percentile page time on the network type identified by NetworkId.
* **Percentile95 (float)**: The 95% percentile page time (in seconds) on the network type identified by NetworkId.
* **Goal (float)**: The response time goal for the page.
* **PagesMeetingGoal (int):** The number of pages run on the network type identified by NetworkId with a response time less than the goal.

**For more information:**

<http://msdn2.microsoft.com/en-us/sql/aa336310.aspx>

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), [how would you rate this paper](mailto:sqlfback@microsoft.com?subject=White%20paper%20feedback:%20Using%20Visual%20Studio%202005%20to%20Perform%20Load%20Testing%20on%20a%20SQL%20Server%202005%20Analysis%20Services)?