PERFORMANCE MANAGEMENT AND CAPACITY PLANNING FOR MICROSOFT SQL SERVER

Metron Technology Limited

This paper describes the author's experiences in performance management and capacity planning for systems based on Microsoft's SQL Server database management product. The paper covers SQL Server database design, the resource consumption and performance data sources that are available, the way in which a monitoring regime was established, and the important areas that were addressed when tuning for optimal performance. The paper concludes with a description of the techniques employed to establish and verify a capacity plan using modeling techniques.

1. Introduction

When I started out with performance management for Microsoft SQL Server, I was told that all I needed was in Microsoft's 'Books Online'. Unfortunately this is rather like trying to learn a foreign language by reading a dictionary – all of the required material is present, but there is little indication of the things that are really important.

This paper is based on my experiences, gained over the past 12 months, working on performance management and capacity planning for systems involving Microsoft's SQL Server RDBMS. It is intended as practical guide to what is important and what is not, which data sources and metrics to use, and how to use them for tuning, reporting and capacity planning.

Section 2 describes the significant impact that good database design (both logical and physical) can have on performance. Often the performance analyst only becomes involved when poor performance starts to impact the ability of end-users to perform their business functions - well after the logical and physical design decisions have been made. Even in this case, there are a number of things that can be done without reverting to the drawing board!

Section 3 is concerned with database queries and covers query analysis, optimization and the use of parallel queries in SQL Server. Section 4 describes the available data sources for resource consumption and performance data, and concludes with some recommendations on the establishment of a monitoring regime.

Section 5 describes the way in which the data can be used to build and validate models that can be used for performance prediction. Section 6 provides a summary – and some thoughts for the future.

2. DB Design

2.1 Logical Design

Some people consider the pursuit of 3rd normal form as the Holy Grail of database design, but this is only the first step. After normalization, a process of de-normalization (in specific areas) should begin. De-normalization can contribute significantly to overall performance, particularly in environments like Data Warehousing, where the data changes are relatively infrequent.

There are a number of de-normalizing techniques, the most useful of which are data redundancy, and the use of virtual columns. Data redundancy can be used to remove the need for table joins – which are usually expensive operations to perform. (See section 3.1.2 below for details on how to determine the contribution of joins to the resources used by an SQL batch.) Data redundancy should be avoided for highly volatile data, where a significant amount of work could be required to maintain data integrity.

Virtual columns are calculated from data columns when a query is executed (they are not actually stored within the database) and can be used to reduce the load on the application.

2.2 Indexes

I/O is an inherently time-consuming process and so SQL Server uses indexes to reduce the physical I/O required to retrieve a required row. Candidate table values for indexes should be chosen with care. Whilst indexes speed up access to data, they must be updated when data is inserted or modified. The cost of maintaining a large number of indexes on data that is highly volatile can easily outweigh their benefits for data retrieval.

2.2.1 Index Selectivity

The *selectivity ratio* (Sr) of a potential key value in a table is a good guideline to its usefulness as an index. If the number of rows which are uniquely identified by the key is Rk, and the total number of rows on the Table is Rt, then

Sr = 100*(Rk/Rt)

The lower the selectivity ratio, the more useful the key would be as an index to the table. SQL Server stores this ratio for each index. If the selectivity ratio of a key is less than 15%, the optimizer will chose a full table scan

2.2.2 Index Design

SQL Server allows just one clustered index for each table. The actual data pages for the table are stored at the bottom of the clustered index b-tree, which means that the data is physically stored on the clustered key. Clustered indexes are therefore particularly efficient where data is selected by ranges of that key value (since all the data for a given range will be stored together).

In addition to a clustered index, SQL Server allows up to 250 non-clustered indexes for each table. The leaf nodes of a non-clustered index b-tree contain pointers to the physical location of the data.

2.2.3 Index Tuning Wizard

We found the Index Tuning wizard (which ships as standard with SQL Server 7.0) to be an extremely valuable tool. The wizard can be used to analyse a trace file collected by the SQL Server Query Profiler (see section 4.4 below for details) and recommends the addition of indexes that would improve performance. The wizard will even implement its recommendations if you so wish!

2.2.4 Maintenance

After some time in use, an index can easily become fragmented as data is added and modified in the table. This fragmentation can increase the number of I/Os required to retrieve the required row. The DBA can specify a 'fill factor' for an index, which specifies the amount of free space required on each index page. The fill factor is not automatically maintained and should be 're-applied' as regularly as is required to reduce the level of index fragmentation. (Section 4.2 gives details of the monitors available to detect excessive index fragmentation.)

If the users of your SQL Server system report that performance is gradually getting worse, the level of index fragmentation is one of the first things to check!

2.3 Row Updates

From version 7.0 onwards, SQL server will update a value in a row directly, without having to delete and re-insert the row. The exception to this is when the value being updated is part of a clustered index. If this is the case then SQL Server will perform a 'deferred update' which consists of a delete and insert (perhaps even on a different page) together with the required transaction log and index page changes. This process can be quite expensive – the moral of this story is to avoid the inclusion of frequently updated columns in clustered indexes.

2.4 Partitioning, Filegroups & RAID

In general terms, the larger a table is, the slower it will be to access. There are two table partitioning techniques which you can use to reduce the amount of work that SQL Server has to do – vertical partitioning and horizontal partitioning.

Vertical partitioning is where some of the columns are stored in one table, and the remainder are stored on one or more additional tables. This can be useful if you can locate the most frequently accessed columns in the 'primary' table

Horizontal (or row-wise) partitioning results in multiple tables, each of which has the same structure. One example of this would be in time-stamped data where each week or month of data has its own table.

Further performance gains can be achieved by allocating partitioned tables to specific *filegroups*. The filegroup (a new concept for SQL server 7.0) defines the actual operating system data files that will be used by SQL server. The ability to map tables and indexes to separate file groups can therefore be used to control the physical location of the data.

Where RAID disk technology is being used, multiple filegroups can be set up to span different stripe sets. The objective here is to spread the I/O as evenly as possible across the physical disk devices.

2.5 Configuration Parameters

System Configuration parameters can be updated directly by using a stored procedure. The syntax of the command is

Exec sp_configure parameter_name, parameter_value

If you use this commend without any arguments, it returns the current settings for all of the parameters. Generally speaking we found that the default values were acceptable in most installations, the one important exception being the 'Max Async IO' parameter. This parameter defines the maximum number of asynchronous I/O threads and should be increased from its default value of 32 for larger I/O subsystems.

3. DB Queries

3.1 Query Analysis

3.1.1 Query Analyser

A brand-new tool called **Query Analyser** became available for SQL Server at release 7.0, replacing the previous (and much more limited) tool **ISQL/w**. As its name suggests, this tool enables you to track in detail the steps that SQL server takes to satisfy a SQL query, including the resources consumed at each step. One thing that has not changed is that the query optimizer is still *cost-based*.

3.1.2 Graphical Execution Plans

Perhaps the most exciting facility provided by Query Analyser is the graphical execution plan. I don't know why the Microsoft people don't make more of a noise about this because it is a fabulous analysis tool, and streets ahead of anything that Oracle has to offer!

Figure 1 below shows a sample graphical execution plan.

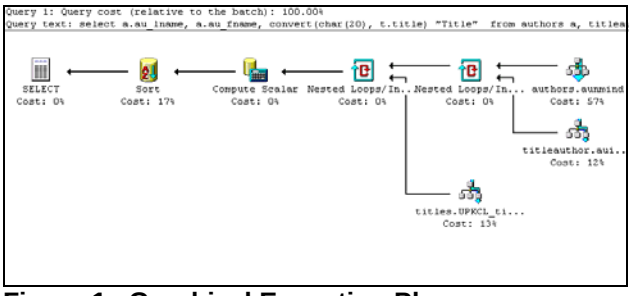

Figure 1. Graphical Execution Plan

Note that the major and minor data paths are indicated by the width of the lines. Moving the mouse over any of the icons in the plan will cause a 'ToolTip' to be displayed. A sample ToolTip is shown in figure 2 below.

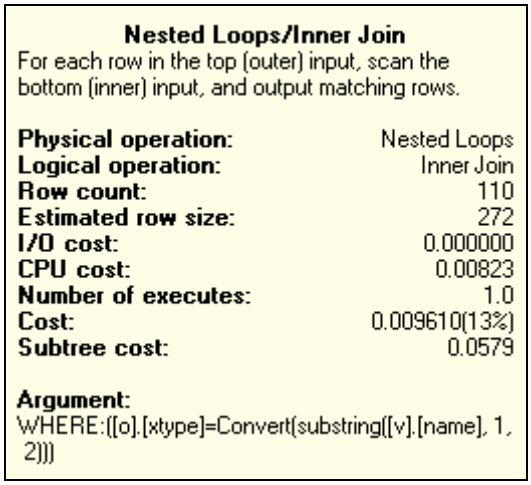

Figure 2. Execution Plan ToolTip

From a capacity management point of view, the most use of these are the CPU and I/O costs. Unfortunately these are not expressed in terms of CPU seconds or logical I/Os, but in a consolidated relative metric. Its main usefulness therefore is to identify the relative resource consumption of the steps in the processing of the query. Query Analyser calculates the relationship of the total cost of this query to other SQL queries in the batch and shows it in the header.

3.1.3 Statistics

Details on actual CPU consumption, elapsed time and logical/physical I/O can be obtained by using the *set statistics* options. These options can be defined for a user session in Query Analyser.

The option *set statistics time on* gives the CPU time and elapsed time for each parse/compile and execute step in milliseconds. An example of the output is shown in figure 3 below

```
set statistics time on
select a.au_lname, a.au_fname, convert(char(20), t.title) "Title"
from authors a, titleauthor ta, titles t
where a.au_id = ta.au_id
and ta.title_id = t.title_id
order by Title

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 7 ms.

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 10 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

Figure 3. time statistics

The option set statistics io on gives the scan count, and the number of logical, physical and read-ahead reads. An example of the output is shown in figure 4 below.

```
set statistics io on
select a.au_lname, a.au_fname, convert(char(20), t.title) "Title"
from authors a, titleauthor ta, titles t
where a.au_id = ta.au_id
and ta.title_id = t.title_id
order by Title

SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 20 ms, elapsed time = 21 ms.

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(25 row(s) affected)

Table 'titles'. Scan count 25, logical reads 63,
      physical reads 0, read-ahead reads 0.
Table 'titleauthor'. Scan count 23, logical reads 35,
      physical reads 0, read-ahead reads 0.
Table 'authors'. Scan count 1, logical reads 1,
       physical reads 0, read-ahead reads 0.

SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 50 ms.

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 53 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

Figure 4. IO Statistics

The relationship between the logical and physical reads can be used to determine the cache-hit ratio, which is a fundamental indicator of the effectiveness of the data cache.

3.2 Query Optimization

SQL server uses a highly sophisticated cost-based optimizer. The optimizer derives an estimate of the CPU and I/O resources that would be required for all of the possible execution plans for a query, and then chooses the least expensive. In order to make this decision, the optimizer uses a set of statistics that provide details of the distribution of the data within each column. Prior to SQL Server 7.0, these statistics had to be updated manually, and performance would degrade if the statistics were not kept up to date. From version 7.0 onwards, these statistics are maintained automatically.

For particularly volatile tables it is sometimes desirable to force a statistics update. You can do this by using the UPDATE STATISTICS {table} command. It is also possible to turn off automatic statistics updating, but this is definitely not recommended!

Another really useful tool (which is available from version 7.0) is the Query Governor. This tool makes use of the fact that the optimizer will estimate the cost of execution of a query, before actually executing it. Using the Query Governor you can set a cost threshold beyond which a query will not be executed. We found this to be particularly useful in preventing long running queries being executed in circumstances (e.g. the use of English Query) where the DBA could not control user's access.

3.3 Parallel Queries

Before SQL server actually executes a query, it will check to see whether the system is capable of supporting a parallel execution plan. If so, it will calculate the most efficient number of threads (the 'degree of parallelism') and process the query accordingly.

The behavior of parallel queries can be controlled using two parameters – maximum degree of parallelism, and cost threshold for parallelism. The first controls the maximum number of parallel threads. The second specifies a cost threshold above which the optimizer will consider the use of parallel threads. The SQL server Profiler (see section 4.4) can be used to monitor the degree of parallelism that is being used for a query.

4. Monitoring

4.1 NT Perfmon

4.1.1 NT Counters

Most NT users will be familiar with PerfMon – the NT Performance Monitor. This provides a number of useful counters for monitoring activity at the operating system level including CPU, Memory, and disk I/O statistics. Note that the command

diskperf –y

must be used to enable the logging of disk data.

4.1.2 SQL Server Counters

When SQL Server has been installed, a number of new objects become available to PerfMon:

- **SQL Server: Access Methods**

Searches through and measures allocation of SQL Server database objects (for example, the number of index searches or number of pages that are allocated to indexes and data).

- **SQL Server: Backup Device**

backup devices used by backup and restore operations, such as the throughput of the backup device.

- **SQL Server: Buffer Manager**

memory buffers used by SQL Server, such as free memory and buffer cache hit ratio.

- **SQL Server: Cache Manager**

cache used to store objects such as stored procedures, triggers, and query plans.

- **SQL Server: Databases**

Database information such as the amount of free log space available or the number of active transactions in the database.

- **SQL Server: General Statistics**

general server-wide activity, such as the number of users that are currently connected to SQL Server.

- **SQL Server: Latches**

latches on internal resources (such as database pages) that are used by SQL Server.

- **SQL Server: Locks**

individual lock requests made by SQL Server, such as lock time-outs and deadlocks. There can be multiple instances of this object.

- **SQL Server: Memory Manager**

SQL Server memory usage, such as the total number of lock structures currently allocated.

- **SQL Server: Replication Agents**

SQL Server replication agents currently running.

- **SQL Server: Replication Dist.**

Measures the number of commands and transactions read from the distribution database and delivered to the Subscriber databases by the Distribution Agent.

- **SQL Server: Replication Logreader**

Measures the number of commands and transactions read from the published databases and delivered to the distribution database by the Log Reader Agent.

- **SQL Server: Replication Merge**

Provides information about SQL Server merge replication, such as errors generated or the number of replicated rows that are merged from the Subscriber to the Publisher.

- **SQL Server: Replication Snapshot**

snapshot replication data such as the number of rows that are bulk copied from the publishing database.

- **SQL Server: SQL Statistics**

data about aspects of SQL queries, such as the number of batches of Transact-SQL statements received by SQL Server.

- **SQL Server: User Settable**

Performs custom monitoring. Each counter can be a custom stored procedure or any Transact-SQL statement that returns a value to be monitored.

Whist this looks, at first sight, to be a useful set of data, it is rather disappointing from a performance management point of view. In particular, there is no information at all on the resource consumption of individual user sessions. Fortunately the SQL Server Profiler tool (see section 4.4) can be used to remedy this!

4.2 DBCC

DBCC (Data Base Consistency Checker) commands are provided as part of T-SQL. These can be used to perform a variety of physical and logical consistency checks and can be used to fix errors. The most useful from a performance management viewpoint are:

- **DBCC PERFMON**

Network, I/O and page cache data

- **DBCC MEMUSAGE**

usage of page and procedure cache memory

- **DBCC SQLPERF**

Transaction and log space usage by database

4.3 Enterprise Manager

SQL Server Enterprise Manager is the primary DBA administration tool. It also allows you to monitor:

- Current user connections and locks.
- Process number, status, locks, and commands that active users are running.
- Objects that are locked, and the kinds of locks that are present.

The process information section includes data on CPU usage, physical I/O, and memory occupancy for each active process. The Locks/Process section provides details of the locks held by each process including the name of the object that has been locked. The Locks/Object section lists all the locked objects together with the name of the locking process.

4.4 SQL Server Profiler

In the old version of SQL Server (version 6.5) there was a tool called SQL Trace which enabled the selective tracking of specified events. This has been greatly extended and improved in the current release (version 7.0) and has been re-named as SQL Server Profiler. Through the use of a set of Wizards and GUIs, the Profiler enables you to define a trace, and log it to a file or to a table. This trace can then be manually replayed and analyzed further using, for example the 'Explain Plan' options in the Query Analyser.

In addition to the GUI interface, the SQL Server Profiler functionality is also available using a set of extended stored procedures, all of whose names start with *xp_trace*. These procedures can be invoked from your own application code and can be used to control and manage the data collection process (see section 4.5 – Establishing a Monitoring Regime).

Using these stored procedures you can select precisely which events you want to monitor, and the data that you want to be recorded for each event. By executing the appropriate SQL to initiate a trace for the data that we require (see the SQL in Appendix A) we are asking SQL server to log information for us. This information is time-stamped and can be written to a standard SQL Server table in the database.

The three sections below indicate the metrics which are traced by default, metrics which vary according to the event class, and a list of the event classes which are important from a performance management viewpoint.

4.4.1 Default Data Metrics

The xp_trace stored procedures always provide the following set of data metrics from which you can select the ones that you require:

- **Application Name** The name of the client application that created the connection to SQL Server.
- **Connection ID** The ID assigned by SQL Server to the connection that is established by the client application.
- **CPU** CPU time (in milliseconds) that is used by the event. This metric is recorded to the nearest 10 milliseconds.
- **Database ID** The ID of the database specified by the USE *database* statement, or the default database.
- **Duration** elapsed time (in milliseconds) taken by the event.
- **End Time** The time at which the event ended. This column is not populated for 'starting' event classes.
- **Event Class** the type of event class that is captured.
- **NT Domain Name**
- **NT User Name**
- **Reads** Number of logical disk reads that are performed by the server on behalf of the event.
- **Server Name** Name of the SQL Server that is traced.
- **SPID** Server Process ID assigned by SQL Server to the process associated with the client.
- **SQL User Name** SQL Server username of the client.
- **Start Time** Time at which the event started, when available.
- **Writes** Number of physical disk writes performed by the server on behalf of the event.

4.4.2 Variable Data Metrics

The following additional metrics are available, some of which are populated differently depending upon the event class.

- **Binary Data** Binary value dependent on the event class captured in the trace.
- **Event Sub Class** Type of event subclass. This data column is not populated for all event classes.
- **Host Name** The name of the computer on which the client is running.
- **Host Process ID** ID assigned by the host computer to the process in which the client application is running.

- **Index ID** ID for the index on the object that is affected by the event.

- **Integer Data** Integer value dependent on the event class that is captured in the trace.

- **Object ID** System-assigned ID of the object

- **Severity** Severity level of an exception.

- **Text** Text value dependent on the event class that is captured in the trace.

- **Transaction ID** System-assigned ID of the transaction.

4.4.3 Event Categories

The different event categories that you can trace are shown below, together with an indication of their usefulness for performance management and capacity planning.

- **Cursors** - Collection of event classes that are produced by cursor operations. Not immediately useful to us from a performance viewpoint, although by monitoring the **CursorOpen**, **CursorExecute**, and **CursorImplicitConversion** event classes, you could determine when a cursor is executed and what type of cursor is used. These event classes are useful to determine the actual cursor type used for an operation by SQL Server, rather than the cursor type specified by the application.

- **Error and Warning** Collection of event classes that are produced when a SQL Server error or warning occurs. By monitoring the **ErrorLog** and **EventLog** classes, we can trap the text of any messages sent to either the SQL Server Error Log or to the NT Application Event Log. The **Execution Warnings** event class can be monitored to determine if and how long queries had to wait for resources before proceeding. By monitoring the **Missing Column Statistics** event class, you can determine if there are statistics missing for a column used by a query. Missing statistics can cause the optimizer to choose a less-efficient query plan than otherwise expected.

- **Locks** Collection of database object locking event classes. The **Lock:Acquired** and **Lock:Released** event classes can be used to monitor when objects are being locked, the type of locks taken, and for how long the locks were retained. The **Lock:Deadlock**, **Lock:Deadlock Chain**, and **Lock:Timeout** event classes can be used to monitor when deadlocks and time-out conditions occur, and which objects are involved.

- **Misc.** Collection of miscellaneous event classes that do not fit into any of the other event categories. Not directly relevant for performance reporting.

- **Objects** Collection of event classes that are produced when database objects are created, opened, closed, dropped, or deleted. Not directly relevant for performance reporting.

- **Scans** Collection of database object scan event classes. Database objects that can be scanned include tables and indexes. Using the **Scan:Started** and **Scan:Stopped** event classes, we can monitor the type of scans being performed by a query on a specific object.

- **Sessions** a collection of event classes that are produced by clients connecting to and disconnecting from SQL Server. Using the **Disconnect** and **ExistingConnection** event classes, we can monitor the length of time each user connection was connected to SQL Server, and the amount of SQL Server processor time and I/O used by the session.

- **SQL Operators** a collection of event classes that are produced from the execution of SQL data manipulation language (DML) operators. Not immediately useful, although we could monitor here the degree of parallelism chosen by the optimizer for any Delete, Insert, Select, and Update SQL statements

- **Stored Procedures** a collection of event classes that are produced by the execution of stored procedures. By monitoring the **SP:CacheHit** and **SP:CacheMiss** event classes, we can determine how often stored procedures are found in the cache when executed.

- **Transactions** Collection of event classes that are produced by the execution of Microsoft Distributed Transaction Coordinator (MS DTC) or SQL transactions or by writing to the transaction log. The usefulness of this collection will depend upon the use that your applications make of the SQL transaction log.

- **TSQL** a collection of event classes that are produced by the execution of Transact-SQL passed to SQL Server from the client. This is perhaps the most useful collection of all! The TSQL event classes can be used to monitor the

execution and completion of a remote procedure call (RPC), a batch, and a Transact-SQL statement. The available classes are:-

RPC:Completed
RPC:Starting
SQL:BatchCompleted
SQL:BatchStarting
SQL:StmtCompleted
SQL:StmtStarting

All of these classes include the actual SQL statement(s) being processed, the 'Completed' classes giving the resources consumed in each case. You can even trap user-selected SQL and invoke the Query Analyser to provide an instant 'Explain Plan' showing the execution route chosen by the optimizer.

- **User** a collection of user-configurable event classes. You can generate your own SQL Server event classes and enable them to be trapped by the Microsoft extended stored procedures in exactly the same was as for any other event.

4.5 Establishing a Monitoring Regime

Our client organisation required the regular capture of performance data (from a distributed environment) and the centralisation of that data in a Performance Database (PDB). This PDB was required to be the central repository for all of the data required for short-term analysis, longer-term management reporting, and performance prediction. The two main sources of data that we used on the target systems were NT PerfMon (NT and SQL Server counters) and the SQL Server Profiler extended stored procedures.

Many of the NT performance metrics are cumulative, i.e. they must be read once, and read again after a known interval. The first readings can then be subtracted from the second in order to determine the activity that took place during the interval. A data capture agent (implemented as an NT service) was defined to snapshot the counters at the required interval (usually 15 minutes in this case) and store the required deltas locally. This data was then retrieved daily and stored in the central PDB.

We had to take a quite different approach for the Profiler data. The NT PerfMon data is continuously updated by NT and must be 'snapshotted' (as defined above). The Profiler data is, by contrast, concerned with *events*, an entry being written to the log location when each required event occurs.

Using the xp_trace extended stored procedures, we started an event queue for the required time period (usually covering the main on-line day) writing the trace data to a table on the local database. This data was copied to the central PDB on a regular basis (usually daily). We did not use SQL server Replication to achieve the central PDB update, although there is no reason why this would not have worked equally well.

In order to limit the inevitable overhead of capturing and storing the event data, we kept the number of event classes and data columns to the absolute minimum (further details in section 5.3 – Workload Definition).

5. Capacity Planning

The primary objective of Capacity Planning is the provision of consistent acceptable end-user service levels, at a known and controlled cost, both now and in the future. There are a number of techniques which can be used to predict the impact of changes in configuration and workload. These techniques employ a variety of methods in including trending, extrapolation and modelling, each of which have their place in the process.

Section 5.1 below compares and contrasts these techniques, the remainder of section 5 is devoted to the process of *performance* prediction using analytical modelling techniques.

5.1 Trending, Extrapolation & Modelling

As the central PDB is built up, it becomes a valuable source of information regarding the way in which varying workloads affect the utilization of the components of the system, and the delivered service levels. Trend analysis using standard statistical techniques (such as correlation analysis and least squares line fitting) can be used to identify patterns and trends in system behavior.

One of the most useful applications of trend analysis is to provide a starting point for future workload level prediction. A linear extrapolation of an observed trend can then be overlaid with commercial data relating to anticipated business changes, and their likely effect on system workload levels.

Whilst linear extrapolation techniques are useful for workload planning, they are not applicable to performance prediction. The delivered performance of a SQL server system is a non-linear function of the workload placed upon it, due to the effects of contention for logical and physical resources – this is where analytical modelling comes in.

Analytical modelling techniques based on multi-class queuing network theories have been used for many years now to predict the effect on system performance of changes in configuration and workload. We have recently extended these techniques to cover SQL Server environments.

The first step in the modelling process is to build and calibrate a baseline model. This is based on actual system measurements and can be used to predict current performance. This prediction is then compared to the measurements and the model is fine-tuned accordingly – a process usually referred to as 'calibration'.

A model consists of a configuration and workload pair as shown in Figure 5 below.

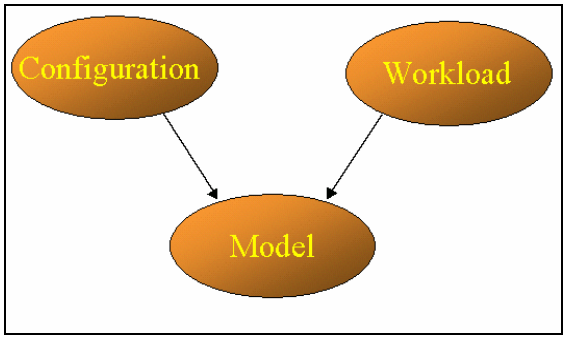

Figure 5. Model structure.

A model is usually built from measurements taken during a busy period, and several baseline models may be required – one for each relevant workload peak. The configuration definition can be derived directly from the information provided by PerfMon.

The workload is described as set of workload components – each of which represents an important classification of work. The degree of detail required here will define the data sources that are required. The process of workload component definition is described in sections 5.2 and 5.3 below.

5.2 'System-Only' Models

A model is, be definition, a *simplification* of reality, built for a specific purpose. Any system model should only be detailed enough to answer the 'what-if' questions posed by the business. For example, if the only thing that we know about the future workload on a system is that it will be twice the current level, then you only need one workload component! If, on the other had (and this is much more usual) different parts of the workload are expected to change at different rates, then we must build a workload component for each of those parts.

A model with a single workload component (often referred to as a 'System Only' model) can be built very simply from the PerfMon NT metrics alone. Where more detail is required, additional data sources must be used as described in the next section.

5.3 Workload Definition

The majority of the models that we built required a grater level of detail than that provided by PerfMon. We used the SQL Server Profiler xp_trace extended stored procedures to trap and store the SQL BatchCompleted Event class, and used the following data metrics from ther trace table:

- NTUserName
- ApplicationName
- TextData (SQL code listing)
- StartTime
- EndTime
- Reads
- Writes
- CPU

A combination of the first three was used to identify the workload component (usually based on application task) to which each SQL completion should be mapped. We were able to derive an I/O and CPU count for the workload from the other metrics for each SQL completion.

The totals thus derived were matched against the overall 'system' totals provided by PerfMon. An 'overhead' workload component (which usually amounted to around 6 – 8% of the total) was built to account for the inevitable differences between the low-level application view (provided by SQL Profiler) and the NT totals.

5.4 Performance Prediction

This section includes some of the results of a modelling study that we conducted. The system in question was required to support a workload which was expected to increase by around 15% per month. A baseline model was built for the mid-morning period (the period most likely to experience the increased user load). A scenario model was then built, using the baseline model as a starting point, and 'ramping-up' the workload contribution from each the SQL Server workloads in linear steps of 15%. Figure 6 shows the predicted impact on CPU utilization. Each vertical bar represents a further fifteen percent linear workload increase (the first is the baseline position). The y-axis is CPU utilization.

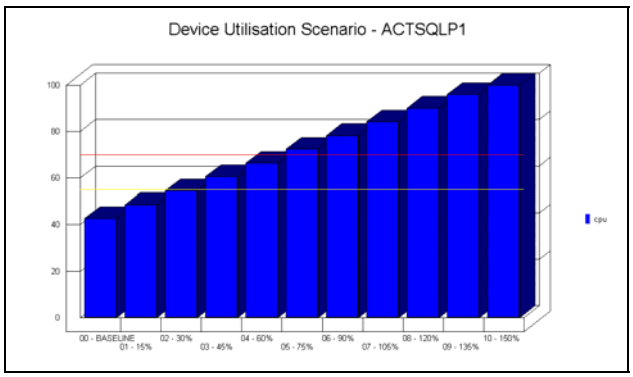

Figure 6. CPU Utilization Scenario

This shows the expected linear behavior and indicated that the system is capable of handling a 135% increase on the SQL Server workload before saturating the CPU. The response time scenario graph in figure 7 below shows the way that the realtive performance (y-axis)of one of the SQL server workloads will changes as the workload increases. (The x-axis is the same as for figure6.)

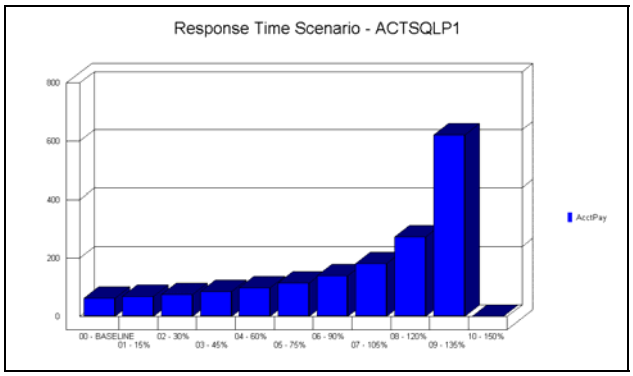

Figure 7. AccPay Response Scenario

Note the non-linear behavior of the response time due to (in this case) contention for the CPU.

The model was then modified to include a CPU upgrade (to a higher clock speed) timed to take effect just before the response times start to degrade rapidly. The CPU utilization scenario from the new model is shown in Figure 8.

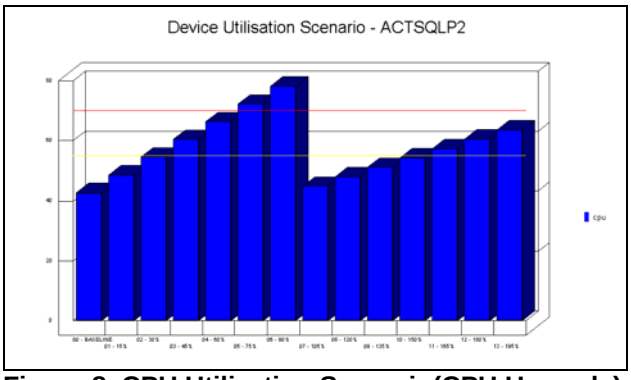

Figure 8. CPU Utilization Scenario(CPU Upgrade)

Note the effect of the CPU upgrade, and the fact that, after the upgrade, the CPU utilization increases in smaller steps due to the increased power of the new CPU.

The most important question was, however, "How will SQL Server response times be affected?" Figure 9 shows the relative effect on AccPay response times of the workload changes and CPU upgrade already described.

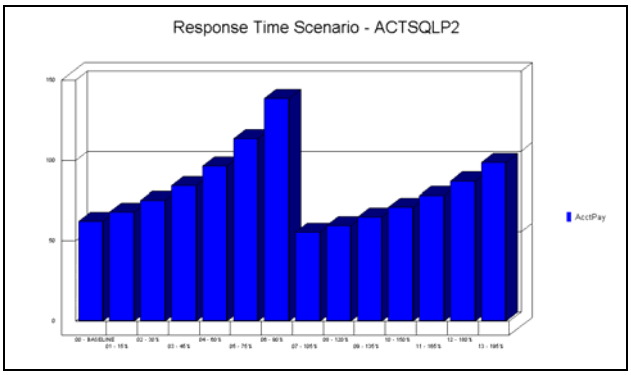

Figure 9. AccPay Response Scenario (CPU Upgrade)

Due the relatively light I/O loading, it was the CPU that, once again, caused the response times to degrade at an increasing rate as the workload grew.

6. Conclusions

Microsoft has improved the power and scalability of SQL server very rapidly over the past 2 – 3 years. This fact, together with the availability of very powerful clustered systems running Windows NT/2000, means that large user populations, with a variety of business workloads, are being supported on SQL server databases.

Whilst traditional methods of simple linear sizing were adequate for smaller, single application boxes, the larger and more varied systems require a more

sophisticated approach – based on modelling technology.

A formal capacity management regime, where investment can be matched against an anticipated future workload, can be established - using traditional analytical modelling techniques, and the data sources provided by SQL server and Windows NT/2000.

Bibliography

Microsoft SQL Server 7.0 Unleashed, Sharon Bjeletich, Greg Mable, et al.

Microsoft SQL Server 7.0 Performance Tuning Technical Reference, DeLuca, Garcia, Reding & Whalen

Appendix A

****

SQL to trace SQL Batch Completion (including resource consumption) and log to a trace file. The trace will autostart with SQL Server

****

```
--Declare the variables.
DECLARE @queue_handle int , @column_value int
--Set the column mask for the data columns to capture
--as documented on Books Online
SET @column_value = 0 -- all columns for now
--Create a queue.
EXEC xp_trace_addnewqueue 1000,
5,
95,
90,
@column_value,
@queue_handle OUTPUT
--Specify the event classes to trace.
--For now, just get the SQL Batch completion class.
EXEC xp_trace_seteventclassrequired @queue_handle,
12, 1 --SQL:BatchCompleted
--Set a filter. (Don't trace the Profiler).
EXEC xp_trace_setappfilter @queue_handle,
NULL,
'SQL Server Profiler%'
--Configure the queue to write to a file.
EXEC xp_trace_setqueuedestination @queue_handle,
2,
1,
NULL,
'c:\trace\demo.trc'
--Start the consumer that actually writes to a file.
EXEC xp_trace_startconsumer @queue_handle
--Save the queue definition as TimQueue1.
EXEC xp_trace_savequeuedefinition @queue_handle,
'TimQueue1',
1
--Mark it for autostart
EXEC xp_trace_setqueueautostart 'TimQueue1', 1
```