Monitor an Enterprise of SQL Servers - Automating Management by Exception with Perl

Linchi Shea
Merrill Lynch & Co.
linchi_shea@ml.com

Abstract

Monitoring a large number of SQL Servers in an enterprise is a difficult task. The SQL Server administrators have to deal with a large amount of very dynamic and diverse information, and many other complicating factors. We found that the age-old principle of management by exception provides an effective framework in organizing our monitoring efforts. This paper describes our experience in using Perl to help automate the management-by-exception approach to monitoring SQL Servers. Perl is used, in most cases together with SQL, to collect and process SQL Server-related system information, to identify important exceptions, and to report them in a highly summarized fashion. The exception reports and a suite of Perl scripts developed to produce the reports are discussed. We conclude the paper by sharing some lessons learned in our struggle to tame SQL Server monitoring with Perl.

1. Introduction

Monitoring a large number of SQL Servers in a large-scale enterprise environment is a difficult task. When the number of SQL Servers reaches 50+, 100+, and still grows, the nature of monitoring these servers becomes significantly different from monitoring a dozen or two SQL Servers. To begin with, the SQL Server administrator (DBA) must monitor events that are recorded in a multiplicity of sources, and must cope with an overwhelming amount of information. They may have to retrieve relevant system information that does not typically fall in their own realm of responsibilities. In addition, the DBAs are often asked to accommodate a fast growing number of SQL Servers with not-so-fast growing resources.

This paper describes how we use Perl to help deal with monitoring a large number of SQL Servers. The next section identifies several prominent factors that have shaped our overall approach. Section 3 gives a brief summary of a simple three-layered strategy that frames our monitoring efforts. The first layer of the strategy deals with prompt responses to highly critical errors. The second layer provides concise management by exception reporting. The third layer consists of a DBA repository for storing system information from each monitored SQL Server. We use Perl primarily at the second layer to automate the collection, processing, and presentation of SQL Server-related system information for administration purposes. Notice that Perl is not the only language used to enable this strategy. Transact-SQL is also used extensively. But Perl is the focus of this paper. Section 4 covers the Perl scripts used in SQL Server monitoring.

Using this approach, we have been able to accommodate the rapid growth in the number of SQL Servers we have to monitor.

2. Motivation

The last several years have seen a rapid growth in the number of SQL Servers deployed in our environment. It is expected that this number will continue to grow rapidly for the next few years.

In search of a way to monitor these servers, we analyzed potential factors that may influence the effectiveness of any monitoring approach we might eventually adopt. We identified the following to be particularly prominent in our environment:

Information Overload

The amount of information that must be processed to effectively monitor the servers grows much faster than the number of the servers. It is also quite easy to run into the over-monitoring syndrome: monitoring too many things results in too little being effectively monitored.

Information Diversity

Comprehensive monitoring requires system information from diverse sources. They include SQL Server error logs, job log files, job schedules, NT registries, NT event logs, directories, NT service configurations, SQL Server system tables, SQL trace or profiler output, performance monitor counters, SQL Server cluster configurations and state, and so on.

Moving Targets

Monitoring requirements are in a constant state of flux, and the DBAs are in an endless learning mode. The monitoring strategy has to be completely open to accommodate changing requirements, and new experiences and insights.

Distributed Roles and Responsibilities

Though the responsibilities of SQL Server administration is assumed by a dedicated DBA group, it is often not easy to effectively manage SQL Server monitoring-related issues. Communication of these issues can be a challenge. For instance, NT system changes that are managed by separate organizational groups, but may have significant impact on SQL Server, are not always promptly communicated to the DBA group, or not communicated at all. Even within the DBA group, one cannot take communication of SQL Server monitoring for granted.

Enterprise Monitoring Infrastructure

The SQL Server DBAs work in an environment that has a well-equipped, but rapidly evolving, enterprise infrastructure for system monitoring. This infrastructure dictates how critical production errors are monitored. But it is not provided as a solution to meet all the platform-specific or local group-specific monitoring requirements. The SQL Server monitoring strategy must complement this enterprise infrastructure rather than implement a substitute.

Varying Degrees of Urgency

Not all events are of equal criticality. Some events are so urgent that we need to respond right away, even in the wee hours. Responding to other events can wait until the morning, or we simply want to be informed on a daily basis. For events that are neither immediate nor temporarily deferred, we are content with a summary.

If we hope to effectively monitor a large enterprise of SQL Servers and keep up with its rapid growth, we need a strategy to frame our monitoring efforts. In the next section, I outline such a monitoring strategy.

3. Overview of the Monitoring Strategy

In a nutshell, our monitoring strategy is based on the principle of management by exception.

Whatever the strategy may be, the key to its success in this large enterprise environment is automation. We need tools to enable the automation, which adequately accommodate the factors identified above. The tools we use are a combination of a central DBA repository and a collection of SQL and Perl scripts. The repository is made up of (1) a SQL Server 7.0 database, (2) an NT registry hive, and (3) an NT directory tree. The repository stores comprehensive system information for each monitored SQL Server. Perl scripts are used to help extract system information from heterogeneous sources on remote servers, to store the information in the central repository, and to summarize the information by highlighting changes and errors, and identifying patterns.

3.1. Layers of Monitoring

We divide the focus of the monitoring strategy into three layers:

1. Critical production problems
2. Exception reporting
3. DBA Repository

The first layer deals with critical SQL Server production errors that must be addressed promptly. We leverage the existing enterprise monitoring infrastructure to notify the DBAs, and rely on the SQL Server alert mechanism and other enterprise monitoring agents for error detection at this layer. SNMP traps are sent to the enterprise data center, which then notify via pager or telephone the DBAs on an escalation list. SQL Server alerts have been heavily customized to minimize the number of nuisance notifications. The primary objective at this layer is to receive *prompt* notifications on all critical production problems, and on these critical ones only. A full discussion of how this layer of monitoring is implemented is, however, beyond the scope of this paper.

At the second layer, all system exceptions, including the production problems monitored at the first layer, are captured. While at the first layer

we'd like to be minimalist in containing the number of notifications, at the second layer we want to be comprehensive in capturing the exceptions that are deemed important. However, we want to report on these exceptions in a highly summarized fashion to reduce information overload on the DBAs.

The third layer of the monitoring strategy is the SQL Server DBA repository itself. At this layer, the objective is to be all-encompassing, i.e. to capture all the system information that may be useful even though it may not be presently used in generating any exception report. This repository also serves other useful purposes beyond reporting on exceptions. For instance, it provides a historical view of system configurations.

The focus of this paper is on the second layer.

3.2. Exceptions

We are interested in the exceptions originated from the following sources:

1. *Operational exceptions.* These are SQL Server-related error conditions. For instance, a table has become corrupted.
2. *Configuration exceptions.* These are significant changes (intended and unintended) in system configurations that affect SQL Server behavior. For instance, an NT administrator has unintentionally changed permissions on the SQL Server data files. Or the SQL Server is changed to no longer listen on an IPC method.
3. *Exceptions to the DBA standards.* These are serious deviations from the established local DBA standards, including standards on configurations, security, best practices, and SQL coding if applicable.

3.3. Exception Reports

The main categories of exception reports that we generate include:

Monitoring System Configurations

We retrieve SQL Server related system information from each monitored SQL Server and store it in the central DBA repository. Retrieved system information includes: (1) the SQL Server registry hives, (2) the SQL Server system tables, (3) generated SQL scripts, and (4) SQL Server-related NT and hardware configurations. We then compare two consecutive daily snapshots of this system information to report any significant changes.

Monitoring Database Schema

Changes in database schema are tracked by comparing two consecutive daily snapshots of the same database schema. Significant differences are highlighted.

Monitoring Security

SQL Server security data is retrieved from each server into the repository on a daily basis. We check the security data in the DBA repository in two areas: (1) we report on any significant changes in SQL Server access and database permissions, and (2) we check and report on the compliance with the established SQL Server security lockdown policies.

Monitoring Performance and Space

Currently, the only performance data collected and reported on is table fragmentation and distribution statistics. Space usage information is extracted from the SQL Server system tables that are already stored in the DBA repository. An exception report is produced when any of the space thresholds is crossed.

Monitoring Logs

In addition to the SQL Servers' own error logs, a large number of log files are generated daily from various scheduled jobs for the DBAs to review. The NT event log may contain additional error messages. All these log files and the NT event logs are scanned to generate a consolidated concise summary.

Monitoring Scheduled Tasks

Depending on the robustness of error trapping, a scheduled job may fail without leaving any trace in its log. The scheduled jobs on all monitored SQL Servers are scanned, and a report on any failed jobs is produced. This also provides a means to monitor the monitoring system since the setup of monitoring uses scheduled jobs heavily.

Monitoring DBA Standards and Best Practices

Not all the DBA standards can be codified in scripts, but a significant portion can be. These scripts highlight whether there is any serious departure from the SQL Server DBA standards or best practices.

4. Perl Scripts

To implement the monitoring strategy of management by exception outlined above, a suite of scripts were written. Most of them were in Perl, some in Transact-SQL, and others in Perl with embedded SQL scripts. Since this paper is about using Perl to facilitate monitoring SQL Servers, I will focus the discussions on the workings of the Perl scripts. All the Perl scripts mentioned in the paper are listed in the appendix. Some sample reports from these scripts are also shown in the appendix.

4.1. DBA Repository

The DBA repository is made up of three collections of information on a dedicated SQL Server: (1*) a SQL Server 7.0 database*, (2) *a registry hive*, and (3*) a tree of NT directories*.

The SQL Server 7.0 database contains tables mirroring all the significant SQL Server system tables. The data is updated daily and kept for 30 days (configurable). Examples of the system tables include sysprocesses, syslocks, sysdatabases, sysdevices, and syslogins at the server level as well as sysobjects, sysprotects, and sysindexes at the database level. In addition, several tables record NT level information such as space on each drive, CPU count, and memory, as well as summary information such as database space consumption.

The registry hive contains the last five versions of SQL Server registry keys and values. The following registry hives from each monitored SQL Server are copied to the registry of the DBA repository server[1]:

- HKLM\Software\Microsoft\MSSQLServer,
- HKLM\System\CurrentControlSet\Services\MSSQLServer,
- HKLM\System\CurrentControlSet\Services\SQLExecutive

They are copied under the key

HKLM\Software\SQLServerAdmin\MonitoredServers\<Server Name>\<Date String>

Registry is used as a part of the DBA repository because some important SQL Server configurations are stored in the registry. There is no point of converting these SQL Server registry keys and values into any other storage format.

The NT directories contain generated SQL scripts for re-creating database devices, backup devices, server configurations, database schemas, and SQL Server scheduled tasks. The directory tree also contains exported system tables in plain text files. In addition, a SQL Server setup initialization file is generated daily. This is mainly for convenience because the same information is already captured in the registry hive and the system tables. Furthermore, complete database access information is dumped out daily to a text file. These files are kept for 30 days.

The system table information is conveniently retrieved into the DBA repository using SQL Server remote stored procedure calls or linked servers. The generated SQL scripts and the exported system tables are produced by the GenerateSQLScripts.pl and ExportSystemTables.pl scripts. The SQL Server setup initialization file is generated by script GenerateSQLSetupIniFile.pl. File extension pl indicates that it is a Perl script.

4.2. System Configuration Changes

Every day after the SQL Server registry keys and values are copied, script CompareSQLRegKey.pl is run to compare the most recent two versions of the registry keys and values for each monitored server. A summary report is produced if there is any significant change.

The CompareSQLConfig.pl script is run daily to identify significant differences between two consecutive copies of the generated SQL scripts for setting the server and database configurations. This helps identify, for instance, whether the SQL Server security mode is changed, a new data device is added, a new task is scheduled, a database option is changed, or even a new database is created.

4.3. Database Schema Changes

There is often a need to compare the schemas of two databases to highlight any differences or to compare the schemas of the same database over time to identify any unauthorized changes. There are many existing approaches with varying degrees of success. Most of these approaches compare system tables through SQL queries. We found that, with the help of Perl it is much easier and versatile

[1] Most examples in this paper assume that SQL Server 6.5 is monitored.

to compare two copies of properly generated database schema scripts.

The CompareDBSchema.pl script executes in two steps. In the first step, it talks to SQL Server through the SQL Server distributed management objects (SQLDMO) to generate SQL scripts for the database objects. We control how SQL scripts are generated for which objects, depending on what we want to compare. The second step does a quick parse of the scripts and stores the result in a Perl nested hash record structure. For our purposes, there is no need to write a sophisticated parser to parse these SQL scripts. Because we control how they are produced by SQLDMO in a predictable format, Perl can parse them very conveniently. The Perl script compares the two hash record structures according to some comparison rules. These rules can be easily added to or removed from the Perl script.

Notice that for planned schema changes, this would provide additional validation that the changes have indeed taken place. At the risk of stating the obvious, the facility is not meant as a substitute for rigorous change control management.

4.4. Performance and Space Problems

Since the information on both database and disk space for each monitored SQL Server is already collected in the DBA repository, producing exception reports when the space consumption has crossed a threshold is done easily with SQL scripts alone.

To check table fragmentation, we run script AlertTableFrag.pl that executes an embedded SQL script in each user database. The SQL script produces, in a temporary file, the result of checking fragmentation against every user table. The Perl script then scans through the temporary file, reporting on large tables with fragmentation (i.e. scan density ratio) that exceeds a threshold.

Similarly, script AlertStaleStats.pl runs SQL Server distribution statistics utility against every user table index. It then scans the result for large table indexes whose statistics have not been updated for a period of time that exceeds a threshold.

We are also working on a Perl script that would automate the processing of SQL Profiler/Trace files and highlight significant performance variances for the same queries or stored procedures.

4.5. Log Files and NT EventLog

In our environment, the DBAs must review a large number of log files each day to identify problems or verify that the systems are in good health. If done manually, this process is boring and laborious, and may result in the log files not being effectively reviewed at all. We need a way to condense this vast array of log files while making sure that no significant errors would be overlooked. This is achieved with script CheckSQLErr.pl, which processes the log files as follows:

- The script summarizes the SQL Server startup process by identifying the SQL Server startup date/time, and reports any failure to listen on any configured IPC method.
- For each SQL Server error number, the script reports the total number of occurrences of the error and the last time it occurred as well as the error message text from the latest occurrence.
- For deadlock error messages, the script reports the total number of times deadlocks have been detected by SQL Server as well as the last time a deadlock occurred.
- The script takes advantage of the fact that every database backup is recorded in the errorlog. If a database has not been backed up within the last 24 hours (configurable), this information is noted in the summary.
- Other messages/errors (e.g. KILL messages, DB-Lib problems, and ODBC connection errors) are treated similarly. This list is easily extensible by adding new regular expression patterns, whenever we see fit.

Words/lines that are not needed in the summary are skipped, and the rest are stored into a Perl nested hash record structure. The script then prints the summary report from the record structure. The script can be easily modified to decide what should or should not be skipped or printed.

The NT EventLog is summarized similarly for SQL Server related errors and warnings using CheckNTEventlog.pl. SQL Server error messages that are written to both the SQL Server error log and the NT EventLog, and therefore already reported by CheckSQLErr.pl, are ignored.

4.6. Security and Access Exceptions

Script CheckSQLLockdown.pl checks each server for significant deviations from the established DBA security policies. For instance, it checks for the following (not an exhaustive list):

- Presence of SQL Server logins with null password
- Guest account
- Whether any SQL Server user can automatically get local NT administrator 's privileges through shell escape (i.e. xp_cmdshell)
- A list of NT accounts with sa access to the SQL Server through NT authentication
- Remote servers with sa access to this SQL Server via remote stored procedure calls

A second script, CheckSQLSecurityChanges.pl, reports on significant changes of security setup in the above listed areas, and database access changes in general. The report is produced again by comparing two consecutive copies of security setup and access information output by a SQL script.

4.7. Exceptions to DBA Standards and Best Practices

When we establish DBA standards and best practices, we intend to have them followed, or broken for good reasons. If they are broken, we definitely want to be informed. The CheckSQLStdDev.pl script scans SQL Server to report on deviations from the DBA standards and best practices.

The following is a sample list of DBA standards and best practices that are codified in script CheckDBAStdDev.pl:

- *Hardware and NT configurations.* Is the time on SQL Servers synchronized? Are the data files placed on a uniform level within the directory tree? Is compress turned on for the drive that stores the database files? Are consistent file extensions used?
- *Server- and database-level configurations.* Is the amount of memory allocated to SQL Server within the reasonable range? Is the master device marked default? Are data files and log files for a database placed on separate disks? Is a disk device shared by databases? Do data and log share the same device for a database? Is there a device not used by any database? Is auto shrink turned on? Are there too many data files for a database? Is a database set to grow in small increments?
- *Database schemas.* Is there a new user object created in the master database? Are there any users with mismatched logins? Is there any index with very low selectivity? Are there any orphaned users? Are there any duplicate indexes? Are there any cluster-indexed columns that are updated frequently? Is there any very wide clustered index with multiple non-clustered indexes on the same table? Is there any procedure/trigger using 'SELECT * FROM' or doing INSERT INTO <table> without identifying column names?

It is worth pointing out that what we consider should be on the list of the DBA standards and best practices are constantly changing.

5. Lessons Learned

Our experience so far indicates that organizing Perl scripts under an articulated framework to accomplish specific SQL Server administration tasks is a powerful concept. We have been able to scale this approach in three directions. First, we started with monitoring a relatively small number of SQL Servers. Adding more servers results in little extra monitoring effort. Secondly, the SQL Server DBA group has grown in size. These exception reports prove to be an effective communication tool within the group. Thirdly, with the repository containing comprehensive system information, we have been able to create new exception reports easily to cover the areas we overlooked.

Perl is integral to the success of this strategy. The large number of modules that come with the ActiveState distribution or available from CPAN makes it easy to extract system information from such diverse sources as NT registries, NT event logs, SQL Server databases, and text files, and to obtain information on hard drives and filesystems. It is very convenient to wrap Perl around NT utilities such as *sc.exe* (a versatile NT service configuration and query tool from the NT resource kit) to get system information. Similarly, we take advantage of the text processing power in Perl to combine the results from multiple SQL scripts into a desirable presentation.

For the kind of monitoring discussed in this paper, we found that it is often simpler to first run a SQL

script through *isql.exe*, and then use Perl to process the well-formatted results. In many cases, this is much easier than going through a row-oriented database access module like Win32::ODBC.

Why not use third party tools? Using third party tools to execute the same tasks accomplished by the Perl scripts described in this paper would require an expensive potpourri of tools. We are not aware of any single commercial package that does this type of monitoring comprehensively, or that can be easily tailored to our specific requirements.

Why not use a scripting language like VB? The intent of this paper is to demonstrate the value of Perl in helping automate the SQL Server monitoring tasks rather than discount the value of other languages. It would be interesting to see how any other commonly used language or tools come close to the power of Perl in text processing and pattern matching, and the ease of Perl to glue together the results of wide-ranging tools.

Perl is not the most convenient tool for everything. This is obvious, yet easily overlooked when one is totally engrossed in Perl. In our case, for instance, some exception reports are much more easily produced with SQL queries alone.

Very gladly, we found that many of these Perl scripts are useful beyond monitoring SQL Servers. For instance, it is common for us to receive ownership of a SQL Server installation that has been running without administrative care. We are able to run some of our system monitoring and security monitoring scripts to give us a quick inventory of the system and to identify areas we should pay attention to bring the system in line with our standards.

As a bonus, we also found that our monitoring strategy enables us to maintain what we call 'live documentation'. Since all the system information is in the repository, we can run a script against it to generate up-to-date documentation for any server we are monitoring.

Constantly, we are being reminded that there is no such thing as 'the production release' of the monitoring solution. We are forced to regularly modify our scripts to incorporate new experiences, to adapt to changes, to cover new requirements, and to correct false assumptions, in particular, assumptions on what is worth noting and what is simply a nuisance. The combination of Perl and SQL proves to be extremely convenient in this regard.

Finally, it should be stressed that the approach described in this paper should be applied only to monitor the production environments where changes are introduced in an orderly fashion and any uncontrolled changes are truly exceptions. Otherwise, the exception reporting can be overwhelming and defeats its very purpose.

6. Conclusions

Writing Perl scripts to automate large-scale system administration is not new at all. People have been doing this since the inception of Perl. Unfortunately, though, this still seems to be a rather foreign concept to most Microsoft SQL Server DBAs. Hopefully, this discussion of automating SQL Server monitoring with Perl has demonstrated the power of Perl in improving SQL Server administration.

7. Appendix

7.1. Perl Scripts

The following is a list of Perl scripts mentioned in this paper. They can be obtained from the author:

- GenerateSQLScripts.pl
- GenerateSQLSetupIniFile.pl
- ExportSystemTables.pl
- CompareSQLRegKey.pl
- CompareSQLConfig.pl
- CompareDBSchema.pl
- AlertTableFrag.pl
- AlertStaleStats.pl
- CheckSQLErr.pl
- CheckNTEventlog.pl
- CheckSQLLockdown.pl
- CheckSecurityChanges.pl
- CheckSQLStdDev.pl

7.2. Perl Modules

The following standard Perl modules are used in the scripts mentioned in this paper:

- Win32::Registry
- Win32::EventLog
- Win32::OLE
- Win32::Service

The following modules are also used, and are available from www.roth.net:

- Win32::AdminMisc
- Win32::Perms

In addition, I collected common Perl routines used in performing DBA work into the following module:

- SQLAdmin::DBA

All Perl scripts listed above in Section 7.1 use this module.

7.3. Sample Exception Reports

To illustrate the exception reports produced by the Perl scripts, we give some sample here. Notice that these reports have been reformatted and abridged to fit the required layout of this paper. Each sample report is followed by a brief explanation, and boxes are used to draw attention to certain salient items in the reports.

7.3.1. CompareSQLRegKey.pl

```
***Alert SQL Registry Changes

[ABCSQL01]
[Diff btw 2000-04-12 & 2000-04-13] =>{
  [Software] => {
     [Microsoft] => {
         [MSSQLServer] => {
              [Client] => {
                [ConnectTo] => {
                  <>value: ABCSQL02
                }
              }
              [MSSQLServer] => {
                [Parameters] => {
                  + value: SQLArg2
                }
              }
         }
     }
  }
}
```

This report shows that between April 12, 2000 and April 13, 2000 on SQL Server ABCSQL01, the client network configuration to server ABCSQL02 was changed and a new startup parameter (SQLArg2) was added.

7.3.2. CompareDBSchema.pl

```
***Alert SQL DB Schema Changes ...

[ABSQL01.CaseDB]
Diff btw 2000-04-12 and 2000-04-13

***Msg: DB objects in 2000-04-12 but
   not in 2000-04-13:

 Procedure     dbo.sp_TableLoad
 Procedure     dbo.sp_dbcc
 Table         dbo.AppConflict

Total # of DB objects in 2000-04-12
but not in 2000-04-13 = _3_

Total # of DB objects in 2000-04-12
but not in 2000-04-13 = _0_

***Msg: Scripts for these common DB
   objects are different:

 Procedure      sp_CheckSum
 Procedure      sp_BatchUpdate
 Table          tb_Employee
```

This report highlights the differences in database CaseDB schema between April 12, 2000 and April 13, 2000. For instance, procedures sp_TableLoad, sp_dbcc, and table dbo.AppConflict were dropped between April 12, 2000 and April 13, 2000. It also shows that no new objects were added. Moreover, it indicates that even though procedure sp_CheckSum and sp_BatchUpdate, and table tb_Employee are in the database on both April 12, 2000 and April 13, 2000, they are modified.

7.3.3. AlertTableFrag.pl

```
***Alert Table Fragmentation

[ABCSQL01.CaseDB]
  dbo.Accounts              80.00%
  dbo.TranMaster            77.00%
```

This report shows that two tables, Accounts and TranMaster, in database CaseDB on server ABCSQL01 are considered large (e.g. > 100,000 rows) and are significantly fragmented (the main indicator scan density ratio < 85%).

7.3.4. CheckSQLErr.pl

```
[ABCSQL01]
Chking Errorlog D:\MSSQL\LOG\Errorlog
Restarted at: 2000/02/20 14:55:45
```

```
Err: 1608, Severity: 21, Total #: 36,
   Last Occurred: 2000/04/26 11:47:06,
   A network error was encountered
   while sending results to the front
   end.

Chking DBCC log D:\DBA\LOG\DBCC.LOG,
   Created 0 days ago
CaseDB, Msg: 2540, Severity: 16, Total
   #: 1, Allocation Discrepancy: Page
   is allocated but not linked;
tempdb, Msg: 2540, Severity: 16, Total
   #: 79, Alloc Discrepancy: Page is
   allocated but not linked;
tempdb, Msg: 2546, Severity: 16, Total
   #: 1, Table Corrupt: Extent id 256
   on alloc pg# 256 has objid –641,
   used bit on, but reference bit off.

[ABCSQL02]
Chking Errorlog D:\MSSQL\LOG\Errorlog
Restarted at: 00/02/10 08:36:35

Chking DBCC log E:\DBA\Log\DBCC.LOG,
Created 0 days ago

DB: tempdb, Msg: 2540, everity: 16,
   Total #: 1, Allocation Discrepancy:
   Page is allocated but not linked;
Warning:  dump tran with no_log issued
   by 'sa' invalidates log dumps in
   database 'Loans' taken after Apr 27
   2000 6:05PM, Total #: 1, Last
   Occurred: 2000/04/27 18:15:11
***Deadlock detected, Total #: 46,
   last occurred: 2000/04/30 17:24:34
```

Among other things, this report summarizes the following: SQL Server ABCSQL01 is last started on 2000/02/20, and so far, it has encountered 36 errors of error number 1608. It highlights when a log file is created. It also shows that the SQL Server has encountered a total of 46 deadlocks and the last one took place at 17:24:34 on 2000/04/30.

7.3.5. CheckSQLStdDev.pl

```
[ABCSQL02]
***Warning: 16MB out of 512MB
   allocated to SQL Server

***Warning: These data files are in
   the root directory:
        D:\data1.dat
        D:\data2.dat

***Warning: The following devices are
   not used by any database:
TestCaseDB_Dat01,
   D:\MSSQL\Data\TestCaseDB_Dat01.DAT
TestCaseDB_Log01,
   E:\MSSQL\Data\TestCaseDB_Log01.DAT

***Warning: Master device is default.

*** Warning: these devices are shared
   by different databases:
   DeviceName     DatabaseName
   -------------- -------------
   DBA_DAT01      DBA
   DBA_DAT01      ProjectDB
   DBA_LOG01      DBA
   DBA_LOG01      ProjectDB
   master         master
   master         model
   master         pubs
```

The report identifies the potential violations of the following DBA best practices:

- Giving 16 MB out of 512 MB of available RAM to SQL Server is unusually low.
- We do not advise placing data files in the root directory.
- If a device is not used by any database, it should be removed. Otherwise, it's just a waste of space.
- The master device should not be the default data device.
- No device should be shared by different user databases.