Investigation of factors that determine the ability of computer information systems to be self-healing.

By

Sean Ryan, BSc (Honours)

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

A Thesis Submitted in Fulfilment of the Requirements for the Degree of Master of Science

Research Supervisor	***Research Supervisor***
Dr. Noreen Quinn-Whelton	Michael McCarthy

Submitted to Waterford Institute of Technology
September 2013

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Declaration:

I declare that the writing of this thesis and the research contained within is my own work. Any assistance received has been acknowledged where appropriate.

Signed: ________________________________

Sean Ryan

Date: ________________________________

Abstract

Self-healing features within future system designs could potentially help reduce computer system operational costs and allow for a reduction in complexity. When this research was conceived, reduction of operational costs within the IT field was a challenge. With today's shrinking IT budgets, it has become a necessity. Automation and reduction of human interaction in system administration through the use of self-healing designs is one such method that can help reduce the overall cost, while improving utilisation can reduce the potential impact in relation to system downtime and performance.

This thesis is concerned with the design, evaluation and analysis of a self-healing mechanism and its effects on a real-world computer data system's availability and performance. The results from the analysis demonstrate the effects and benefits of using such a system in a real-world environment. It discusses what worked and what failed within the design and looks forward into what features and design could improve a self-healing system's abilities.

Table of Contents

List of Figures

List of Tables

List of Graphs

Acknowledgements

I would like to thank my wife and our four daughters for their patience and support during the course of this research. I would also like to thank Bausch and Lomb for their support and for giving me the opportunity to undertake this research project.

I would like to thank David Lee for his continued support, technical assistance and guidance with all things relating to the software engine.

I would also like to thank the following list of professionals, both inside and outside WIT, who helped during the various stages of this project, through their guidance, feedback and support:

Waterford Institute of Technology staff who helped with this research project:

Dr. Mícheál Ó hÉigeartaigh.
Michael McCarthy.
Dr. Noreen Quinn-Whelton.

Staff consulted external to Waterford Institute of Technology:
Dr. Gabriel J. Byrne (University College Dublin).
Dr. Kevin Ryan (University College Galway).

Experts consulted during construction of this research:
Paul M. Horn (IBM Senior Vice-President & Director of Research).
Dr. Jeff Kephart (Mgr., Agents and Emergent Phenomena, IBM Research).
Dr. Klaus Herrmann (Berlin University of Technology).
Rean Griffith (Columbia University).
Dr. Yixin Diao (IBM Research Staff).
Rob Pike (Principal Engineer: Google Labs).
Kevin Schofield (General Manager, Microsoft Research, Microsoft Corporation).
Dr. Ian Bond (Senior Lecturer in Aerospace Materials, University of Bristol: Department of Aerospace Engineering)

Other Acknowledgements:

Mandy Regenass (Corporate Account Manager, Diskeeper Corporation Europe)

Jennifer Massaro (SupportSoft Inc).

Mark Rein (Epic Games Inc)

Ruban Gilbert (Market Development Specialist, Akamai Technologies, Inc.)

Johan G. van de Groenendaal (Chief Manageability Architect, Intel Corporation**).**

Vijay Tewari (Senior Architect, Intel Corporation).

Mazin Yousif (Principal Engineer, Intel Corporation)

Jim Hobbs (Senior Architect in Strategy (retired), Intel Corporation)

Glossary

ACI	Autonomic Computing Initiative.
DBA	Database Administrator.
DMAIC	Define, Measure, Analyse, Improve, Control
HDD	Hard Disk Drive.
HP	Hewlett Packard.
IBM	International Business Machines.
I/O	Input/Outputs (within a computer system (read and writes))
IT	Information Technology.
.NET	Windows software framework.
SLA	Service Level Agreement.
SQL	Structure Query Language.
VB	Visual Basic - a third generation programming language.

Chapter 1: Introduction.

1.1 Introduction

The increasing complexity in maintaining information technology (IT) systems is adding to the cost of operations and ownership of these computer systems. In a world where system outages are becoming much more expensive and user expectations more demanding, IT managers are being challenged to improve system “up-time” while also reducing overall operational costs. This has introduced a new set of challenges: to remain operationally efficient without incurring wastage by operating with human or computer resources that are not fully utilised and thus adding to expenditure (Vilja, 1990). One possible method to assist in this challenge is to reduce the operational complexity through design and implementation of self-healing/self-management systems. This would effectively reduce the amount of time and effort needed for the human operator/administrator to interface and maintain the system in use. In this age of “credit crunch” or “recession”, whichever label we use, it means the same thing. Financial investment is now, more than ever, harder to justify. Companies that once strived for maximum “up-time” [1], no matter what the cost, are now talking about acceptable risk and investing less financially. This seems to suggest that perfect “up-time” in computer systems is now deemed less realistic because of the cost of obtaining it. Every layer of redundancy in a system has an underlying cost and cost in this modern age is harder to justify. This doesn’t mean that possible downtime as a result of these risks costs any less to the business.

The success of self-healing should help aid future systems to become more reliable, and also allow them to evolve with higher complexity, without making them increasingly difficult to interface with (IBM, 2006). IT components produced by high-tech companies over the past few decades are so complex that IT professionals are challenged to effectively operate a stable IT infrastructure (IBM, 2006). With the introduction of self-managed and self-healing computer systems, it is envisaged that computer operators and administrators will be required less to maintain a systems operation, freeing up their skills and time to use systems, rather than maintain them.

[1] Up-time: A reference to the amount of time a system is online and operational or uninterrupted system availability.

Presently, the main obstacle which is delaying the design and introduction of self-healing systems is complexity. Dealing with it is the single most important challenge facing the IT industry (IBM, 2001).

Operational complexity for the end-user/administrator, therefore, is helping to drive up IT costs as well as reducing how effectively systems operate, but to evolve systems into self-healing systems' variants will increase design complexity.The amount of human intervention required to keep computer information systems operational varies with the number of, as well as the complexity of, systems in use. The reaction and repair time can be increased by the complexity in diagnosing problems, as previous experience and knowledge are essential to repairing such systems. The motivation for this research pursuit is to explore if manual intervention, and thus, monitoring of systems, can be reduced by introducing elements of self-healing (Ryan, et al., 2008) and to build and analyse the effectiveness of an engine that can aid a computer system to be more self-sufficient and rely less on human intervention.

1.2 Background and objectives

The present evolution of modern systems is towards more self-sustained systems that react to situations and attempt to continue operations after experiencing a fault. This evolution towards Autonomic Computing[2] is a move that allows computers to grow in complexity by reducing the complexity presented to the human element, either in usage or in management.

If these systems were to become completely self-healing, they would need to react by following certain defined rules and options, thus drawing from knowledge and experience. A truly autonomic system needs detailed knowledge of its components, current status, ultimate capacity, and all connections with other systems to govern itself (BMC software, 2006). It cannot simply attempt to react to a fault, unless it is fully "aware" of the system environment that it is attempting to heal. If the human expert reacted as such, they could potentially cause more problems than they fix. This could be compared to a Doctor diagnosing a patient, without all the facts and patient

[2] Autonomic Computing: An initiative started by IBM in 2001, of which the ultimate aim is to develop computer systems capable of self-management.

history to hand. The Doctor is thus ill equipped and there is an increased risk of error or misdiagnosis as a result.

There are obvious advantages to the use of self-healing systems by businesses and managers. The first of these is that the systems are easier to manage than systems which simply alert faults. As a result of this, system administrators are required less and hence companies can save money. With every advantage, however, there is also a potential disadvantage – masking complexity can be hazardous, as features hidden from the administrator/manager means they will have little or no exposure to it and hence may not have the expertise to diagnose and repair it, if and when the needs arise. The IT manager/system owner could risk becoming a “slave” to the level of expertise that the system vendor/contractor has acquired over time and exposure. This type of support can potentially be more expensive and hazardous, mainly because the (contract) support personnel do not have the same vested interest in another company’s success. From a personal experience with computer systems support, if one does not have sufficient in-house expertise, one is at risk of experiencing expensive delays and ineffective support.

The objective of this thesis is to determine whether a computer system can better self-manage its own operations and if a system can be given the ability to heal itself, which was achieved by examining the routine operations of a data system that has been monitored and altered by a custom-built self-healing engine (Self-Healing Autonomic Database Engine: S.H.A.D.E.). It also aims to determine if the design and introduction of a self-healing agent can improve a system utilisation, by making the system more automated and thus reducing human interaction. Bausch and Lomb, like many companies, is constantly challenging its workforce to improve their productivity, through initiatives such as automation, as well as to research and implement such initiatives, where possible, and to aid in the reduction of human intervention, without effecting the system’s operational quality.

1.3 Outline of the thesis

This thesis presents an experiment to detect and heal faults on an Oracle database system running on the Windows server platform. Initially, a literature review was conducted (outlined in Chapter 2) whereby other approaches to self-healing systems were investigated. During this exercise, a moving trend towards self-managing systems was discovered, not just in commercial products such as Oracle and SQL server, but also in initiatives like IBM’s autonomic computing.

An experiment was conducted with a software engine to both detect and react to faults within the system. The software engine detects faults and logs results for two Oracle-based systems. The healing operations were only conducted for one of the test systems, allowing for comparisons of results between real-world systems with and without the ability to self-heal faults. However, this thesis was not conceived to focus on factors of ‘what if’, but to focus on what could enhance an information systems’ ability to self-heal and help to improve its operational status, thus enhancing its ability to remain in operation. This thesis investigates industrial databases through a literature review, by designing a self-healing engine, of which its primary function is to heal a database system.

Chapter 2: Literature Review.

In this chapter, the author outlines the relevant background theory/material in relation to self-healing systems. The chapter examines the current trend towards self-healing systems, by examining what is already in the market as well as what is being developed and tested in the area of self-healing computers. Existing computer systems that are moving towards and including self-healing options will be examined and compared.

Chapter 3: Materials and methods used to design and build the healing engine.

This chapter outlines all the proposed software elements and relevant platforms for the project. The methodologies that are used are described and explained in this chapter. The various stages of the design of the self-healing agent will also be described.

Chapter 4: Results and evaluation

This chapter outlines results gathered from two separate systems and compares both sets of operations and results. It contains a discussion that outlines both the beneficial and non-beneficial results of the system running under the self-healing engine routines.

Chapter 5: Discussion.

An explanation and description of both the beneficial and non-beneficial results of the system operating with the self-healing engine are given. The author's own views on the results will be discussed in this chapter.

Chapter 6: Conclusions and Future Work.

The conclusion chapter will discuss various topics such as problems in system design/implementation and lessons learned from such. It also looks at possible future designs in relation to self-healing systems of the future.

Chapter 7: Appendices.

This chapter outlines papers referenced through this document, as well a bibliography of books referenced during the engine design. Finally, third party products referenced and used during the course of this research are also listed.

Chapter 2: Literature Review.

2.1 Introduction

The field of self-stabilising computer systems is not a new topic for computer designers. The benefits associated with and the abilities required to create a system that can better “self-manage” have been well documented but as yet little has moved from the page into real-world systems. Several new systems such as Oracle and SQL server list “features” that have self-healing elements and thus benefits, but overall the systems still require substantial amounts of human time to ensure they remain operational and optimal. Self-managed is often confused with easier management, where user interfaces are changed and made easier for the user or features are hidden, making systems easier to interface with (so long as they are working). But self-healing is more concerned with easy use and masking complexity, than making systems better able to operate without the human element(s).

During the course of this literature review, it became clear that some of the biggest names in the Computer industry such as Oracle, IBM and Microsoft have shown interest in self-healing systems or at least have made some strides towards making such options a reality. IBM has made some of the biggest strides, but alas only on paper, as it attempts to introduce standards in the form of its autonomic computing initiative. Self-healing is one of the four key properties of autonomic computer systems. Its ability can enable large-scale software systems to deliver services on a 24 x 7 basis and to meet its goals without requiring any human intervention (Czap et al., 2005).

2.2 Information system evolution

The one factor of computer system design that remains constant is that they are constantly in a state of change. As systems evolve, they become more advanced and thus more complex. This evolution is not just in relation to systems components; hardware, software, firm wave. But also in how the system operates in relation to the data it handles. The data changes (unless it is a data warehouse system) and thus what is read/written changes along with how it is read/written. A system that performs optimally today may not tomorrow or next week, without making any physical changes or experiencing any fault. It may be allowed to evolve into an unhealthy system by simply not doing anything at all. Its own operations have allowed it to change so much that it simply cannot perform as well as it was once able. "In terms of total cost of ownership [3](TCO), however, the hardware cost is only a small fraction. The management overhead of dealing with the complexity of the system rises quickly with the system scale." (Zhang, et al., 2005).

The computer game industry has become one of the largest and fastest growing. It spans across several hardware platforms and runs on numerous custom-enhanced engines, most of which is largely C++. These range from fairly simple smaller portable games right up to highly complex physics and AI engines running thousands of routines and textures on the screen at any given time. Each release and new system means more complex coding requirements and techniques. One example is "Gears of War" released in 2006. An internal presentation shows the game engine running with "250,000 lines C++, script code and 250,000 lines C++ code for the main engine" (Sweeny, 2006), along with components for various hardware Libraries. It was designed with 10 programmers, 20 artists, 24-month development cycle and a budget of 10million" (Sweeny, 2006), which translates into a software house with large resources for a piece of (complex) entertainment. Can a system so vast in lines of code and developers that worked on it be possibly released without bugs or issues? Perhaps not and it wasn't, but it was later patched automatically. So one could argue it was packaged, shipped, installed (if needed) and played, but the system Microsoft put in place patched and "healed" the software before it became or was reported as a major issue. Certain bugs remain undetected for some time, because modern software is more complex and feature-driven, that the element with a problem is simply not found until it is in the field. This costs money for various reasons. Testing is expensive in itself. It requires people, programs, policies and time. Testing and Quality need to capture as many issues before they get into the field. Once they get into the field, they cost money to fix. The cost the user money in lost time and dealing with problems, reporting

[3] Total Cost of Ownership: A financial estimate which helps determine direct and indirect costs of a product or system.

problems and waiting for fixes (Vanden Eynden, 2007), so each issue costs revenue. Obviously, it is far cheaper to find and fix before release, but due to complexity, the software still get released with issues. Hence modern system designers are required to recognise the disadvantages for companies when they need to run complex, hard-to-manage systems that require expensive expertise. “BitVault’s main objectives are self-managing, strong scale-out capability and very high reliability” (Zhang, et al., 2005). “Windows 2000 is not alone in shipping with bugs; most commercial applications have flaws that can affect the correctness of their results, or indeed whether they deliver results at all.” (Shaw, 2000).

This issue is not limited to software packages. Even sophisticated hardware such as Cisco switches and HP disk controllers are fixed with firmware revisions, which are essentially software patches. As users and consumers, we expect bugs and issues when computers are involved. We wouldn’t accept a kettle that only boiled water every now and again. But we reboot our computers on a regular basis, patch, and install new driver but why? Because it has become an accepted factor in computer usage. But perhaps instead of introducing more features and complexity, wouldn’t the experience bring us more value and save us precious time if the systems managed most of these themselves? In the real world, however, there is not enough time or enough testers to test every combination of every variable. Not all bugs will be found, making quality assurance a risk management discipline (Vanden Eynden, 2007). Almost with some irony, no software in the history of Microsoft development has ever been through the incredible, rigorous internal and external testing that Windows 2000 went through.” (Foley, 2000). Yet the same version of Windows shipped with 63,000 'defects (Foley, 2000).   Every IT organisation is unique. Each organisation has its own change management processes, its own security requirements, its own technology platforms and standards. As a result of this uniqueness, no single automation solution could possibly automate every unique process or every unique environment characteristic out-of-the-box (Opsware, 2006).

The model mentioned whereby Microsoft fixed the software bugs (with a single user prompt) shows how the user experience can be made easier, because the console contained common standards in both the software and hardware components. The user doesn't even need to know what the patch fixes. You are just informed you need it to play. You accept the issue. Download the patch and start the game. The older model for this operation on a similar platform (PC games) would have required reading up and doing research of what patches were available; seeing if the listed issues affected you hardware platform or would even fix the issues you had; and attempting to fix with another software element (driver or whatever), maybe even an operating system patch (after identifying the patch you are on). If you found yourself attempting to tweak an issue from a performance point of view, the average user needed a large amount of hardware/software knowledge. How much software was purchased that could even be used at all or to the level of its design. Hence the steps, knowledge and work for the user were more complex and time consuming. Now you apply and play. What was the issue and what did it fix? As a user, it's not really relevant. You should be more concerned with using rather than fixing. So automation and standards make the operation from a designer and user easier and cheaper.

But if we were to embrace a process of transparent automatic patching, the manufacturing would need to guarantee that the patches themselves would never cause an issue. So the system would rely on them operating effectively when the existence of the patch proves they had already failed in that regard. Microsoft themselves have even been forced to patch patches because of "human error" (Fontana, 2008). Within the same "Epic Games" presentation, it is also highlighted by one of the developers that in the future, the amount of power and thus complexity will obviously grow for home entertainment systems. Trending on past advances, they see programming platforms using "> 1TeraFlop [4]" (Sweeny, 2006) of computing power, stressing there is a need for "the next mainstream programming language" (Sweeny, 2006) or new standards to cope with the amount of change and power. Hence the experts are planning for more complex systems, the complexity that should be masked from the end-user.

[4] TeraFlop: FLOPS (or flops or flop/s) is an acronym meaning Floating point Operations Per Second.

Does the end-user really need to know how much ram, the speed of their hard disk and what patch/driver revision they are on? We are expected to know every single working component of our computer machines, yet most have no clue how or why their far more simple-to-operate car works. “With automated back-up, enhanced troubleshooting, and zero-touch deployment capabilities of Windows Vista, we can reduce the amount of labour involved in re-building a desktop by at least 50 percent” (Microsoft, 2007). Industry today uses terms like lean to introduce initiatives to reduce cost, and analysing wastage through repeated steps for maintaining systems’ health. Investing in platforms that simplify operations and allow faster recovery or re-builds can greatly reduce costs by avoiding repeating operations or “fire fighting[5]” where it’s not needed.

Every company would want to reduce the amount of time its staff spend dealing with computer faults and repairing them. One company discovered that approximately 70 percent of its staff’s time was spent re-building software, and in order to minimise this time and free up staff for more strategic endeavours, it was necessary to implement a more automated re-building process (Microsoft, 2007). The goal of an autonomic computing architecture is to limit hands-on intervention to extraordinary situations (Sudhir, et al., 2004) and to reduce the amount of time the human element spends fixing and reacting to problems. Information systems are constantly evolving - if not in size, then in complexity. This complexity is increased by the number of components that make up an average information system. With each component acting to a list of tasks, each task could potentially experience an issue or fault and fail. One failure can affect all other components, and thus, the system as a whole. “Using self-healing technology—a combination of hardware, software and firmware—IT infrastructures can be more easily managed” (Hunt, et al., 2003). “With over 27 million servers, over 50 million network devices, and over 2 million terabytes of storage installed across IT organisations worldwide, an increasing number of organisations are turning to Data Centre Automation solutions to automate the management of their IT environments.” (Opsware, 2006).

[5] Fire Fighting: What system administrators must do to correct sudden operational problems.

In visualising the average information system, one may think of software such as a database sitting on some form of computer hardware. This model is simple in definition while being relatively simple to maintain and interface with. “Some operating systems may constitute over 30 million lines of code, which may be created by over 4,000 software engineers.”(IBM, 2001). Combining this with the complexity of a modern database, there are a lot of software elements/features for a human to interface with, and hence, issues arise when a fault is detected. But even at this relatively simple design, there are several elements that could cause a system fault. The database itself, made up of thousands of its own components, could fail through it owns executions. Lock up, fail to expand, requiring more memory, are just a selection. If one was to mix these faults with potential operation system faults, the list could potentially double. If hardware is also taken into consideration, then the number of faults almost triples.

At present, relatively simple systems present problems in a complex manner, because you need to both analyse the fault, along with the other problem, of “how do you fix”. However, these types of simple system designs are rare. The computer systems of today are largely distributed systems, with complex middleware and several hardware components outside of the server hardware. As a result, the complexity for fault diagnosis and repair now becomes very challenging and time consuming. Traditionally, it takes more than one human operator or administrator to manage these systems by working together in teams, each member with different knowledge and skills. In modern systems, the complexity and distributed structures now means that this model is becoming increasingly unsuitable. If the human operator was removed from the equation, the problem focus would shift to how they might be replaced with a computer solution. On one hand, there is a problem of finding a solution to the issue of complexity, by either helping or replacing the human operator. On the other hand, humans offer elements which present A.I. systems have no hope of matching; “humans can handle unknown situations and learn from their experiences” (Herrmann, 2005)

IBM has identified the problem. “The computer industry has spent decades creating systems of marvellous and ever-increasing complexity. But today, complexity itself is the problem.” (Ganek et al., 2003). The solution, on the other hand, is going to take some time to achieve. Yixin Diao has shown in his work that a database can be made to self-optimise. These research results have been applied to the adaptive self-tuning memory allocation feature in DB2 version 9 (IBM’s Database Management System Version 9) (Diao, 2006). Diao and his colleagues tried to tackle some of the challenges by building a Deployable Test Bed for Autonomic Computing (DTAC) which “intended to be a complete end-to-end system with pluggable components so as to facilitate research in various aspects of autonomic computing” (Diao, 2004) This will require computer systems that can adapt, learn, draw on knowledge, process issues and make changes while ensuring each element of itself interacts and communicates. Thus, what is required are computer systems that can adapt to be more like humans, and not just automatically reacting to failures.

Blindly automating responses, if not correctly implemented, can also cause problems. One such example of automation responses and abilities within a system that failed in this respect (and in the lack of standards within its design) was a fatal test flight during the flight of an automated aircraft in 1995. A combination of factors led to a crash, killing the crew and destroying the aircraft. Numerous factors where blamed, ranging from the conditions to the design of the system itself. “A more generic contributing factor in this accident was the behaviour of the automation which was highly complex, inconsistent and difficult to understand. These characteristics made it hard for the crew to anticipate the outcome of the manoeuvre.” (Sarter, 1997).

Situations such as these prompt for changes in the design of automated systems, ensuring that they are less complex to operate and work as “co-operative agents”. Automated systems need to adapt as well as be aware, similar to humans. We cannot expect computers to heal or repair unless we give them the correct skills and tools for the task(s).

2.2.1 Initial developments in self-healing:

In 1974, Edsger Dijkstra defined a self-stabilising system as, “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps” (Dijkstra, 1974). The notion of self-stabilising systems in the 1970s was new and Dijkstra’s article on the subject was ignored. However, the idea of self-stabilising re-emerged again in 1984, when Leslie Lamport brought fresh attention to Dijkstra’s theories, as they added a new dimension to possible ways of dealing with errors and their recovery. Lamport himself has been quoted “I regard the resurrection of Dijkstra's brilliant work on self-stabilisation to be one of my greatest contributions to computer science” (Lamport, 1984). Hence, the theories and ideas are not new, but it is necessary to understand how systems have evolved since Dijkstra’s findings, in order to take the next steps required in building self-sufficient systems to support even more modern and complex environments. Elements of this approach are, in fact, an excellent foundation and are regularly referenced today. Dijkstra’s theories are even more prevalent today as systems continue to grow in complexity and size. Errors and faults are more numerous and complex to solve because they contain far more features and are considerably larger than their predecessors. Because of today’s distributed model of computing, downtime and faults may now affect users on a more global scale, increasing the potential cost to the business. With consolidation of data centres to reduce cost and complexity, the need for stable, reliable systems is needed now more than ever.

Autonomic computing is the next logical evolution of these past trends to address the increasingly complex and distributed computing environments (Ganek et al., 2003). As IBM states, autonomic systems must be “able” to anticipate needs and allow users to concentrate on what they want to accomplish (IBM, 2001). End-users would make better use of their time in front of computers if they did not require the knowledge of the computer system’s inner workings. The end-user could better use their time on the computer actually using them, rather than spend time working out why the system is not doing what it normally does, or as well as it normally does. Server farms and redundant hardware may aid in reducing the risk of performance degradation and system downtime. But hardware redundancy requires investment for hardware and real-estate. It also encourages wastage, as you are investing in hardware that will depreciate over time and you invest in it with the hope you will never need to use it. To be 100% effective and not simply reactionary, the system would need to have the ability to record and adapt by providing the best possible solution, much the same as humans learning from what they are exposed to and adapting to effective procedures based on previous experience.

What defines the health of the system will differ with end-user needs. The end-user, operator, manager, may each have different expectations and needs of the systems. In other words, what defines a healthy system will differ. If a system runs slowly but appears to be healthy to the manager because it is online, can it still be classed as operational? It works but not as well as it could because it is being pushed too hard, not optimised enough or maybe even requires an upgrade. Setting standards of how well the system should operate will better define how healthy it is. Following these standards will then indicate if the system is unhealthy and needs to be healed. The manager's needs are different to the end-user's needs. So the standards must benefit all users of the systems. This could be classed much the same as a patient visiting a doctor. Patients may class themselves healthy if they don't feel sick, but the doctor's definition of healthy would be substantially different, as his/her opinion would be based on experience as well as knowledge and insight. The argument is similar in how system users and system administrators define a healthy system, although the speed of system response times may reverse the argument somewhat. Essentially, what defines a suitable system response time? Also needed are standards for capturing relevant data from systems. "Existing standards can help, such as the Open Group's Application Response Measurement API11 or the Distributed Management Task Force's Common Information Model." (Hermann, et al., 2005).

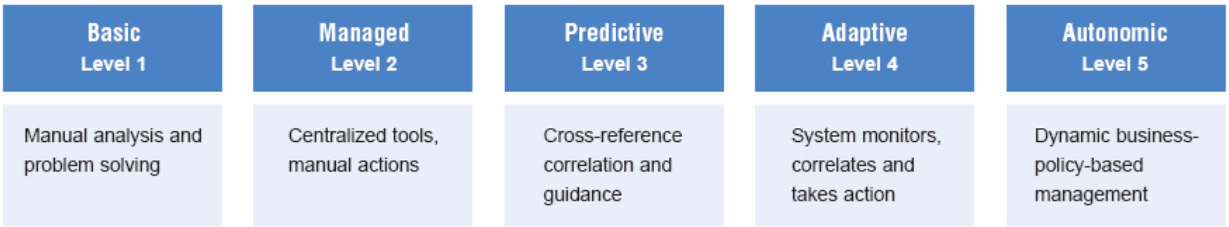
The other main issue is identifying a problem. Within smaller systems, this can be relatively easy, but in larger distributed systems where components are inter-dependent, it becomes much more of a problem. A fault can be detected but localising the component at fault can be a far larger task.

Thus problem localisation is a key ingredient of self-healing. In complex, highly interconnected systems, attributing the cause of a failure to a particular component can be difficult (Kephart, 2005). A misdiagnosis may introduce further change, making it harder to heal the system as it may disguise the root cause. Misdiagnosis not only increases the risk but also the impact (Borning, 1987). The longer the system is non-operational or not running optimally, the worse the situation becomes. Humans dealing in these situations may experience increased pressure and thus stress to fix the issue, which in turn can lead to mistakes. An automated computer system is not as exposed to these forms of external stimuli that can alter how the system reacts or how well they react.

Complexity makes management and administration harder, meaning higher skills are required, along with human operators to mend/deal with issues as they arise. The more complex a system, the larger scope for errors and the harder it may be to fix/improve. Applications and modern environments are made up of tens of millions of lines of code and, therefore, require skilled people to configure and maintain them effectively. (Tosi, 2004).

Delivering system-wide autonomic environments is an evolutionary process which relies on the implementation of technologies and supporting processes. As IBM states, the path to autonomic computing can be thought of in five levels. See Fig 1.0. which starts at basic and continues through managed, predictive, adaptive and autonomic" (IBM, 2001).

Fig 1.0: The five levels (IBM, 2001).

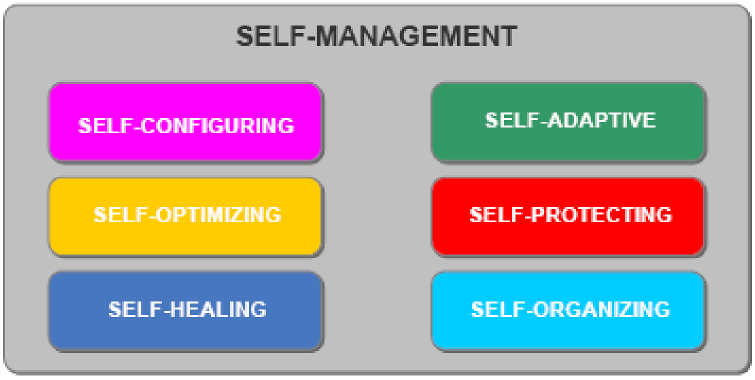


Basic Level 1	Managed Level 2	Predictive Level 3	Adaptive Level 4	Autonomic Level 5
Manual analysis and problem solving	Centralized tools, manual actions	Cross-reference correlation and guidance	System monitors, correlates and takes action	Dynamic business-policy-based management

Through the autonomic computing concept, IBM establishes the term self-management to re-group four main topic areas. These four areas are: self-configuring, self-healing, self-optimising, and self-protecting. The main fault in this classification provided by IBM is in its limitation to explain the interactions and overlap (similarities) between them (Tosi, 2004). Hence IBM's five levels are, at this time, not sufficiently defined or detailed enough to use as a baseline.

Microsoft Research (Asia) has a product called BitVault (a research prototype at time of writing) "a content-addressable retention platform for large volumes of reference data" (Zhang, et al., 2005). It makes use of standard peer-to-peer technology to distribute the main operations/tasks of storing and managing data. It has central control but it is designed to self-manage. The key features of the system are to "self-manage" and "self-heal", while its aim is to continue to function even in the face of failure. Essentially, the aim for the product is to continue to function even in the "face of failure" (Zhang, et al., 2005). The main objectives of any large-scale distributed storage system are its maintainability and availability (Zhang, et al., 2005). BitVault (research) moves closer towards a system that can manage itself, making it cheaper and easier to maintain and increase availability through self-healing.

Fig 2.0: SELF-MANAGEMENT (Tosi, 2004).



Self-healing itself is only one element of self-management as outlined in Fig 2.0 (Tosi, 2004). Self-management is defined as six individual elements which make up self-management as an overall ability. A self-healing system needs to be as autonomous as possible, otherwise there is an increased risk of impact to the system because of the healing operations (Williams et al., 2007). The self-healing must work similar to the human healing system, i.e. not consciously. The "heal" itself should work as another key part/element of the system as a whole/entire system, to be a working element of the system and not just a bolted-on option. Reliability needs to be the key element of the self-healing system. If changes are to be managed through healing, it has to work as designed without incurring further negative effects to the system itself. Achieving high reliability is essential if self-healing is to be considered for applications with performance systems (Williams et al., 2007).

Using any form of self-healing on a system will require physical changes and use of either hardware or software elements that would not normally be in use without one. This means the self-healing system itself increases the amount of change on the said system, but any change in itself introduces an element of risk, either large of small. Hence, any changes need to be monitored and applied through sufficient and reliable controls. A mature and well-designed self-healing system would have little or no penalty on performance and the additional mass of the self-healing system could be traded against a less conservatively-designed structure. Self-healing systems need to be as efficient in operations as any well-designed systems; any overhead could only negate the investment justification. (Williams et al., 2007).

In other words, the healing elements should not impact on the system (in a negative way) and thus have a minimal or even zero footprint [6], otherwise the healing element(s) can cause a decrease in system performance by being operational, thus negating their reason for operation in the first place.

Automating a response can be difficult, as it may be unclear what a program should do in response to an error or fault. A response system is forced to anticipate the intent of the programmer, even if that intent was not well-expressed or even well-formed. Hence the design of the S.H.A.D.E. system was such as to include limited responses to limited defaults, defining faults based on real operations and building responses that were gauged against real-world successes. Ideally, systems could recover from error(s) without any human intervention, but the reality is that most response mechanisms are external to the system they protect and are thus not “aware” enough of the system’s operations. Some simply restrict network connectivity or resource consumption. None (as yet) provide a fully acceptable response strategy (Locasto, 2005).

Continuous service “up-time” remains one of the most important goals of today’s businesses. No matter how well hardware and software are designed, faults inevitably occur. When things do go wrong, system administrators try to make sense of the problem by interpreting messages and system log files to help them fix or prevent a re-occurrence (Sun Microsystems, 2004). Basically, the system alerts that it can no longer do something but it is not always clear why it can’t or what the cause may be. It calls for help and relies on the human element to track down the cause and return the system to optimal health. A human user/expert determines possible faults through experience, adapting to changes and issues as they arise. They are able to draw on experience from previous events whenever possible. But when a system issue arises that they have no experience of or are unable to diagnose, they must seek external expertise and help to aid them in the

[6] Zero Footprint: Computer applications which do not require end-users to install any software.

tasks of accurate diagnose and repair. Hence, designing a system to anticipate each and every possible fault is a difficult task. The combination and type of faults cannot be effectively anticipated in on-going systems, as systems, especially data residing systems, are in a state of constant change. One added line of data can affect how code executes (Dolev et al., 2005). A single fault can be easier to diagnose and thus fix, but when a human or computer is attempting to fix multiple issues at once, the complexity of the problem and the possibility of misdiagnosis is increased. In order to be in "full control", the person or machine used to diagnose and fix the fault, must follow through and ensure the fault has receded and the change they have made is working without causing further issues.

The amount of fault is also a concern. In the real world, an administrator may remain constantly effective if they are dealing with single faults and a steady level of error: i.e. one system with one fault at a time. But in the case of various issues (especially within one system), the fault diagnosis and thus repair becomes increasingly more difficult to complete. Our multi-cellular organism is capable of self-repair: it can automatically replace one or more faulty cells and continue to function optimally, and can deal with multiple faults (Stauffer et al., 2001). Designing a healing system with the ability to detect and repair one fault at a time would indeed be simpler to construct but ineffective in practice. In the real world, we need to be ready to deal with multiple problems. In an effective self-healing system, multiple watches (or cells) must work together in maintaining the system's health to remain in an optimal state of health. One possible method to handle this problem is self-replication, which could allow the complete reconstruction of the original device in case of a major fault (Jonsson, 2006). It would be a much easier operation when working with the software in the system, rather than the hardware.

Health monitoring considers multiple aspects of production applications, including performance, security, connectivity, and application failure (Mushkatin, 2006). This is an interesting statement. It outlines the system's health as being a combination of various elements in operation, not just a faulted system. If it was designed to do all, when it can't achieve these operations, it could be regarded as not healthy. As discussed earlier in this thesis, interpretation of what defines health may be affected by one’s knowledge or ability to diagnose. There can be great divides between how a patient classes themselves as healthy in comparison to how a doctor would view the health of a patient. You may class yourself as fit and underweight but you are judging those facts on how you feel and your knowledge of the fitness and health. A doctor’s knowledge will be based on different experiences and facts. This real-world and practical example is not too different from the problem that divides the end-user’s understanding of a computer system’s health in comparison to how the administrator views the health of the system.

One distinction for health may be a system running optimally. In the modern world, where energy wastage and carbon footprints are as much of a concern as simply saving money, an optimally-running system has great potential in savings not just in capital costing. But how is a system gauged as healthy or optimal? The logical step is checking basic I/O (levels) for CPU, disk and memory. A system has the potential to use less power if it can take more information from memory than the moving parts/disks. So more ram and faster disks may cost money, but using them can reduce cost, operational times and make considerable savings. But poorly-tuned applications can cause the same wastage because they are simply not monitored or regularly optimised. The end-user complaining of slow system response is often the only system health-check that some companies use, after which considerable time will be spent optimising and fixing until the next complaint arises. How much time is wasted “staying behind” problems rather than ahead of them. Financial institutions could save billions of dollars (“as much as $50 billion”) in unnecessary computer upgrades by simply ensuring that their computer systems undergo a regular defragmentation check (Middlemiss, 2000). So health-checks and repair operations can save money. If that is the case then why don’t all companies carry out such tasks and avoid the ill-health situations in the first place. The answer is the lack of standards in defining health. Not all systems are the same, or are made up of the same components. A fragmentation of a drive in a stand-alone, single disk system will have a much greater negative effect on health than it would on a SAN with redundancy, multiple drives and load-balancing operations. They can both suffer from the same problem, but one will suffer from greater ill-health than the other. Why, because one will operate slowly and the drop in health will be more obvious. Standards in defining acceptable response times (much the same as the SMART technology defined earlier) would alert the problem (and even repair it), as the Diskeeper product does. This means a defined problem can still exists, but you can invest in technology that makes it a problem that can be avoided. Narrowly defining what constitutes a failure is a difficult task in such a large operation. Manufacturers and end-users often see different statistics when computing failures since they use different definitions for what a failure actually is. An end user may define a failure based on the speed of interface with the system. If an operation performs slowly, they may report it as an error but the administrator will only deem it a failure if the system is down or inaccessible. Hence, this is why agreement and standards need to be established and agreed upon for defining when a system or an element of a system is, in fact, broken.

You can waste hardware, resources and investment by not checking or knowing what a “healthy” and optimal system actually is. Granted optimal and healthy to one person may not be sufficient for another, but guidelines and standards may at least help to minimise waste. Real systems, however, are much more complex. They are utilitarian, focusing on fitness for purpose even when the problem is not completely understood and the requirements change unpredictably over time. They are built with under-specified components for use in domains that are only biddable (Beeler, et al., 2002). This goes back to the fact that a self-managing system requires the ability to be both adaptive as well as aware of its environments, as it lacks predictable causality and subject to uncontrolled external influences on the computation or the system in which it is embedded (Shaw, 2002). Whatever can influence the health of the system, either internal or external stimuli, the system needs to be made aware of or at least be able to react to it.

In order to investigate the healing process, the term “health” must be defined. Human health is clearly defined through medical checks backed up by hundreds of years of medical research. Often a person can tell a medical professional how they feel. The professional can then draw on experience and attempt to help that person to heal or suggest a solution to the problem. Investigating this communication, knowledge and standards help move towards the healing process. The benefits of defragmentation extend beyond performance improvements to lowering the total cost of ownership for an enterprise, according to the IDC report. By using a defragmentation facility, it is possible to achieve performance gains that meet or exceed many hardware upgrades (Middlemiss, 2000).This is one potential "health" change that could not only save operational time, but potentially expensive upgrades that simply "mask" the problem. It could make an argument into analysing how affecting some systems are being used because of this type of problem, or how much return the investors of the system are getting from their investment.

Maintaining the health of practical systems is correspondingly more complex. First, preservation of health depends on knowing what health is. Since the designer's understanding of both the properties of the system and the users' requirements will be incomplete and dynamic, "health" itself will be imprecisely understood (Shaw, 2002). But how is this activity achieved within computers? System health-checks and benchmarks are not new concepts. But are these standard checks or just a selection of potential issues based on past experiences of what may go wrong? In order to heal, it is necessary to know what to fix. So, as Mary Shaw states, maintaining system health requires knowing what "health" is and recognising when the system needs to be healed. The first problem is establishing the criterion for health, which depends on the way the user is depending on the system (Shaw, 2002). This is an excellent statement, which clearly defines why standards are needed. The health of the system will differ from end-user, to operator, to manager, as each has different expectations of systems. Even today, when a system runs slowly but appears to be healthy to the manager, can it still be classed as operational? It works but not as well as it could because it is being pushed too hard, not optimised enough or maybe even requires an upgrade. Setting standards of how well the system should operate will better define how healthy it is. Following these standards will then indicate if the system is unhealthy and needs to be healed. The manager's needs are different to the end-user's needs. So the standards must benefit all users of the systems. The only way to so is to standardise what makes a healthy system. A self-healing system automatically detects, diagnoses and repairs localised hardware and software problems (Kephart et al., 2003). Maintaining system health requires knowing what "health" is and recognising when the system needs to be healed. The first problem is establishing the criterion for health, which depends on the way the user is depending on the system. This criterion varies from one user to another and from one situation to another. The second problem is recognising the difference between "healthy" and "unhealthy" conditions (Shaw, 2002).

The other issue is identifying a problem. Within smaller systems, this can be relatively easy, but in larger distributed systems where components are inter-dependent, it becomes much more of a problem. A fault can be detected but localising the component at fault can be a far larger task, so problem localisation is a key ingredient of self-healing. In complex, highly interconnected systems, attributing the cause of a failure to a particular component can be tricky (Kephart, 2005). But without knowing the cause, the manager or management system runs the risk of getting stuck in a state of continuous repair. Self-healing has the ability to yield the greatest returns with the field of autonomic computing. Computers that could one day "self-sustain" themselves have the ability to deliver more reliable experiences in computer use, reduce the costs of running them and free up the time of humans who today spend too much time firefighting problems and reacting to failures and faults.

Of course, there is a starting "initial" cost of development and deployment with every system design and administration. Human administrators would not be eliminated from the equation, as there is still no substitute for the human "expert" now or within the foreseeable future. Instead "self-healing" and autonomic computer would improve their working environments, reduce complexity and thus cost. Is the administrator better utilised fixing issues or looking for ways to make improvements? According to a cost comparison study by IDC, defragmenting 1,000 workstations and 10 servers manually would take 52,520 hours and cost $2.1 million, based on $40 per hour for IT staff time. Using a network-deployed defragmentation system, the job would require only 24 hours of staff time at a cost of $960 (Middlemiss, 2000). With lean incentives being a modern challenge for most companies, common terms like "so more with less" are used as much in industry as terms to obtaining high quality. The modern business needs to deliver the same or better quality but do so by spending less money where possible. "The ability to adapt is critical for self-healing systems (Kephart et al., 2003). However, not every system is designed or constructed with all the adaptation mechanisms it will ever need. As a result, there needs to be some way to enable existing applications to introduce and employ new self-healing mechanisms (Griffith, et al., 2006). Otherwise, there will be a continued risk that self-healing systems will need to be platform-dependent and custom-designed for each and every system in operation. What works on one may not work effectively on the other, unless the system is designed to adapt from initial introduction.

Homeostasis is the propensity of a system to automatically resist change from its normal, or desired, or equilibrium state when the external environment exerts forces to drive it from that state (Shaw, 2002). One could argue that a percentage of database issues could be handled by simply restarting the process. This is true but is severe over-kill and there is always a risk of the process failing to restart without human intervention (Griffith et al., 2007). This is a modern "fix" for a lot of system hardware and software, the reboot method. But this would require expensive recoding of objects to make use of Oracle objects and effectively caching transactions at the client end until the system becomes available again. But this is also risky if the client machine crashes and data is lost. A lot of work is a database is running on low memory and simple autonomic-style "tweak" could fix it, unknown to the clients at all. (don't understand this sentence!!)

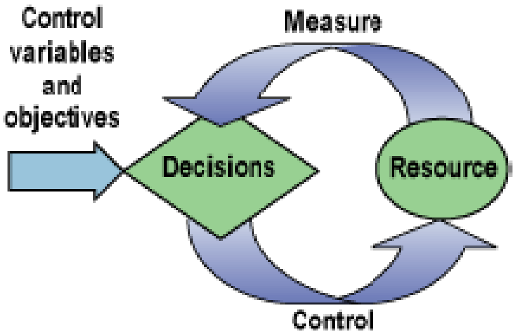
Although it has become quite an unfortunate practice with modern window systems to reboot and hence stop the system as part of the first attempt to return the system to a state of health, it is one option that cannot be in place for a self-management system. It could easily be a first step option, but would be a waste of end-user time, along with a risk that the system will fail to come back online. It would be regarded a much better practice to restart elements of the operating system/database (if needed) rather than flush the entire system as a whole. "Periodic refreshing of data-structures, components and sub-systems done using micro-reboots, which could be performed at a fine granularity e.g., restarting individual components or sub-systems, or at a coarse granularity e.g., restarting entire processes periodically." (Griffith, et al., 2006).

Unfortunately, most monitoring systems tend to show only the symptoms rather than the underlying problem. They act as alerting systems that pass on possible faults, with no understanding of how to deal with or identify the root cause of the problem. The system actually provides sufficient data to predict that a hardware component is about to fail; however, few systems are capable of recognising symptoms or taking proactive actions. And with more companies consolidating applications onto fewer servers to reduce costs, these applications become susceptible to hard-to-diagnose and non-recoverable errors in an increasingly complex set of system hardware and software components (Sun Microsystems, 2004). If placed in human terms, it would be an administrator who knows of a fault, but is without the sufficient training or skills to either diagnose the cause or be able to fix it. They simply pass on the news of a possible error and walk away to find the next problem, leaving another member of the team to fix it. Ideally the person should be able to do all tasks.

The operator is tightly integrated in this management process, and his or her tasks range from defining high-level policies to executing low-level system commands for immediate problem resolution. Although this form of having the human operator “in-the-loop” style management was appropriate in the past, it has become increasingly unsuitable for modern networked computing systems as they have increased in size, complexity as well as geographical locations (Hermann, et al., 2005).

An autonomic element consists of a “closed control loop” (Figure 3.0). Theoretically, closed control loops can control a system without external intervention and can keep it in a specified target state. This concept is can be vital in the design of self-healing systems, because it introduces the desired autonomy (Hermann, et al., 2005).

Fig 3.0: An autonomic element's closed control loop" (Hermann, et al., 2005).



Closed control loops are considered to be the most important concept of self-management. The basic idea of control loops is well known from a wide variety of technical applications - a thermostat (consisting of a temperature sensor and a coupled flow control valve) and a car's anti-lock breaking system are just two examples. Closed control loops can control a system parameter on the basis of some pre-defined "set point" and the constant observation of the parameter's current value. In relation to a thermostat, the human manually defines the set point or desired temperature and the thermostat measures the temperature and reacts by controlling the flow of heat, using the valve (Hermann, et al., 2005). The valve is adjusted based on the feedback from the closed loop. This principle works much the same as computer-controlled loops, whereby the computer adjusts a code parameter variance with the resulting feedback.

A simple example of a control loop would be an operation or set of operations that change the database by increasing the size of the in-memory buffer for caching data, if necessary. Assume the operating system runs a control loop for optimising memory usage. This control loop monitors the amount of free memory, and if it detects a shortage, it uses an interface to tell relevant applications to reduce their memory usage. This system, consisting of the database, operating system, response-time control loop, and memory control loop can produce an undesirable oscillation. The two optimisation criteria directly conflict with each other: Caching large amounts of data in memory achieves a good response time, while keeping memory usage low results in higher response times. One action triggers the other, so they will repeatedly increase and decrease the size of the in-memory database cache. This thrashing will likely decrease the overall system performance considerably and might increase the response time. So, the response-time control loop triggers an even greater increase of the cache size to react to this condition. This feedback loop will eventually lead to a new emergent behaviour of the system: complete failure (Hermann, et al., 2005). This example outlines how ineffective a healing system can be if it is designed to fire and forget and never fail until the system itself faults. Basically, it will continue to attempt to fix the problem, even when it cannot.

The purpose of a self-healing system is to provide reliability and data integrity in the face of imperfect underlying software and hardware (Tesauro et al., 2004). It is fair to refer to modern technologically advanced and complex systems as imperfect because a perfect system would operate 365 days of the year without fault. Although the statement may not be practical, it is what end-users expect from systems, especially in industry, as less and less time is allocated for maintenance. In the modern age, systems are expected to stay up running back-ups in “hot mode”, running maintenance tasks and sometimes even upgrades while users are still being serviced, even when said operations in fact introduce high elements of risk to the overall condition of the system.

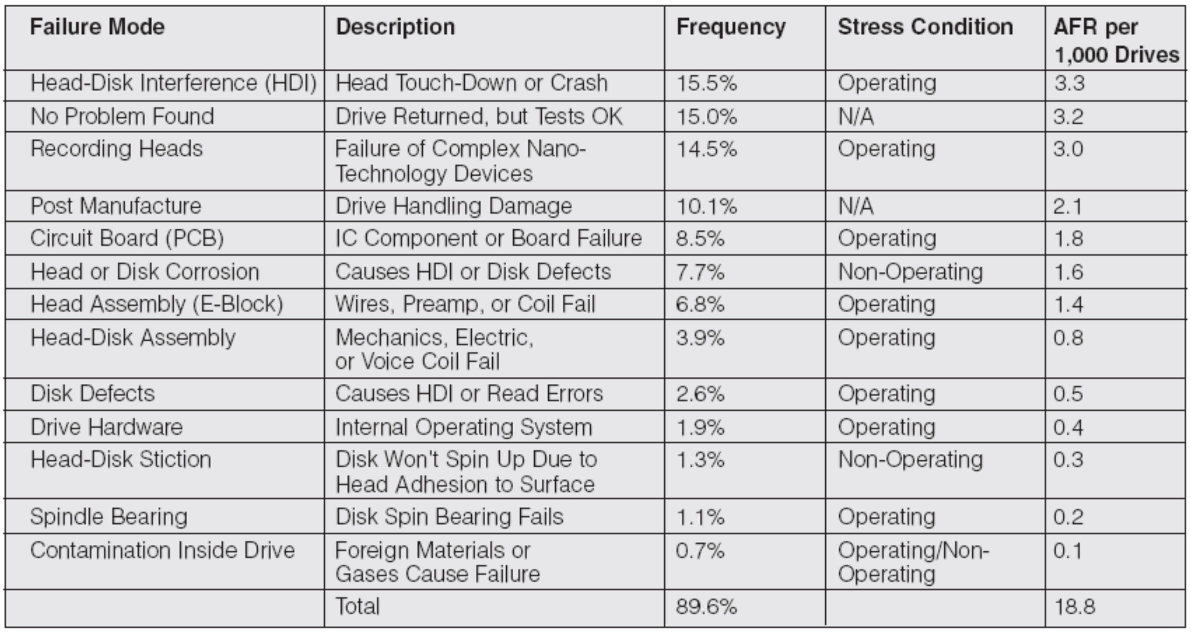
It is fair to suggest that the amount of investment in a system has a direct effect on the possible reliability of a system. Cheap low-cost investment results in less reliability and redundancy options. One could risk minimal investment and avoid the added cost of hardware redundancy or data replication. The cost of system downtime should have a direct bearing on how much should be invested in the system hardware itself. It would be wasteful to invest in expensive software solutions for the data storage and monitoring if the overall foundation, the hardware of the system, didn’t have the capacity to at least guarantee minimal reliability and redundancy if parts fail. But modern day reduced IT budgets have forced managers to look to cheaper and thus more cost-effective options. Time will tell how comparably effective some of these solutions may be in practice against the more expensive variants. One potential is to use cheaper types of drives such as SATA drives which may be “cheaper” alternatives and thus fail more than traditional IDE/SCSI drives [7], but show that parts fail. Using a method such as SMART [8], early “potential” failures can be detected and thus avoid data loss or downtime. As with MAID (Massive Array of Idle Disks) (Burniece, 2005), using cheaper cost-effective components in a reliable configuration (using traditional raid with custom “Aerobic” software) to delivers a more robust system with “future” monitoring for failures.

All components, especially disk drives, have a measure failure rate. More expensive fibre/scsi drives have higher failure rates than the cheaper counterparts SATA. This means you will be replacing SATA drives because of increased failures, but they are cheaper to purchase. Hence, they may be better suited to lower cost data systems where downtime or repair time cost less to the business operations. But installing SATA at a low cost and configuring them in such a way as to expect them to last as long as standard disks would be risky. Table 1.0 below outlines various failures with these cheaper disks.

[7] IDE/SCSI/SATA physically connecting and transferring data between computers usually associated with hard disk and tape drives.

[8] Self-Monitoring, Analysis, and Reporting Technology, or S.M.A.R.T. is a monitoring system for hard disks to detect and report on indicators of reliability; they can help in anticipating failures.

Table 1.0: Results of USCD Failure Analysis of 4,000 SATA Drives, December 2004 (Burniece, 2005).



Failure Mode	Description	Frequency	Stress Condition	AFR per 1,000 Drives
Head-Disk Interference (HDI)	Head Touch-Down or Crash	15.5%	Operating	3.3
No Problem Found	Drive Returned, but Tests OK	15.0%	N/A	3.2
Recording Heads	Failure of Complex Nano-Technology Devices	14.5%	Operating	3.0
Post Manufacture	Drive Handling Damage	10.1%	N/A	2.1
Circuit Board (PCB)	IC Component or Board Failure	8.5%	Operating	1.8
Head or Disk Corrosion	Causes HDI or Disk Defects	7.7%	Non-Operating	1.6
Head Assembly (E-Block)	Wires, Preamp, or Coil Fail	6.8%	Operating	1.4
Head-Disk Assembly	Mechanics, Electric, or Voice Coil Fail	3.9%	Operating	0.8
Disk Defects	Causes HDI or Read Errors	2.6%	Operating	0.5
Drive Hardware	Internal Operating System	1.9%	Operating	0.4
Head-Disk Stiction	Disk Won't Spin Up Due to Head Adhesion to Surface	1.3%	Non-Operating	0.3
Spindle Bearing	Disk Spin Bearing Fails	1.1%	Operating	0.2
Contamination Inside Drive	Foreign Materials or Gases Cause Failure	0.7%	Operating/Non-Operating	0.1
	Total	89.6%		18.8

MAID, together with DISK AEROBICS, anticipates potential failures, instead of reacting to failures, like normal always-on RAID systems (Burniece, 2005), where disks are removed and rebuilt when failure is detected. MAID/SMART technologies are designed to highlight the potential future loss of a disk and to move away from said disk before it fails.

With a system looking to its own components and alerting potential failures, allowing for components to be replaced before an actual failure, the system is thus acting to better protect its own availability as well as its data. Although hardware components are harder to “self-heal”, they can be designed with sufficient redundancy to allow the move off suspected hardware that may or will fail. SMART and similar SCSI alerting and analysis are a step in the right direction towards detecting issues rather than reacting to events. But SMART is only a step in the right direction; a lot more systems and design changes are required in the future. Powerful predictive models need to make use of signals beyond those provided by SMART (Pinheiro, et al., 2007).

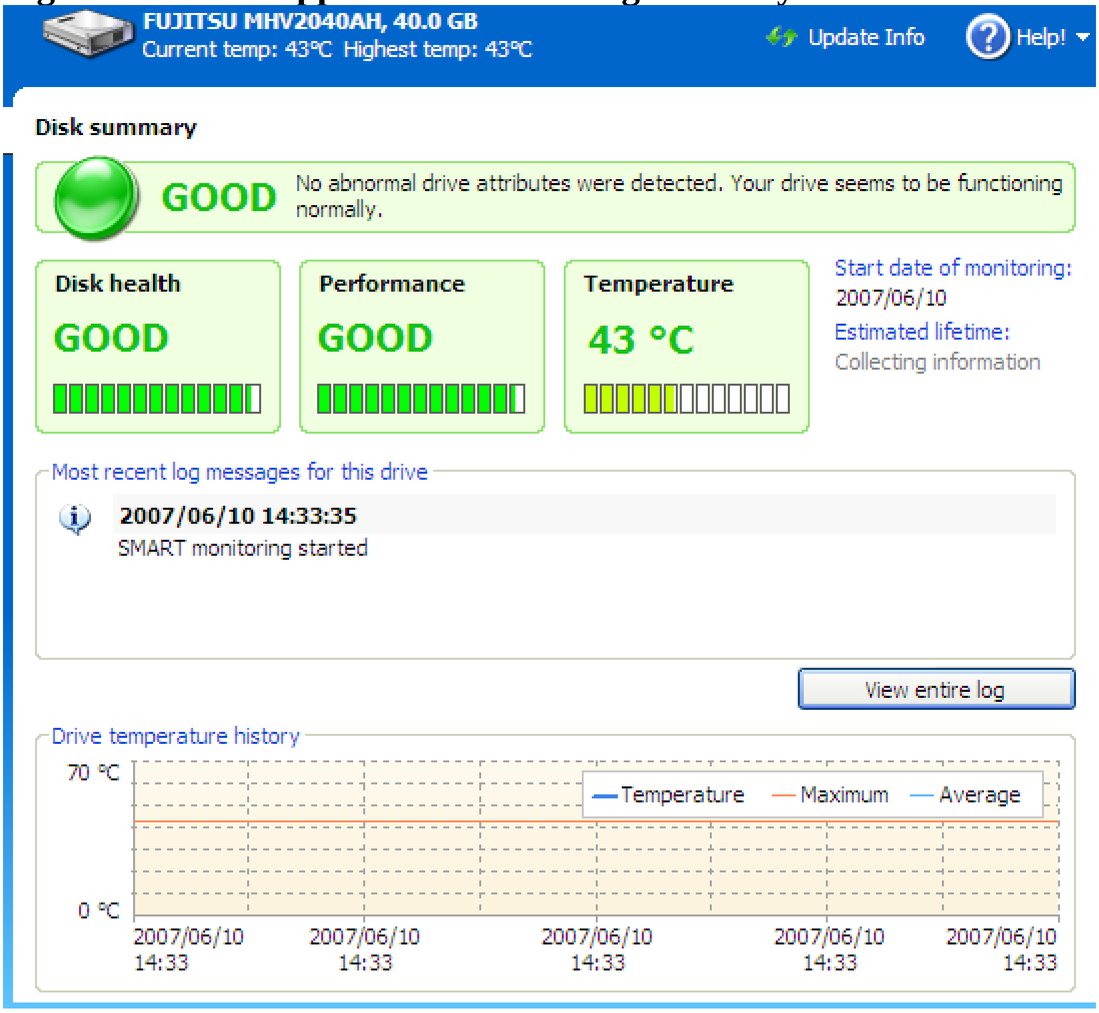
Over 90% of all new information produced in the world is being stored on magnetic media, most which is on hard disk drives (Pinheiro, et al., 2007). Taking this figure into account, one could only hope the said information is being stored on systems with adequate redundancy and sufficient disaster recovery procedures built into their design specifications and operational budgets. Self-healing denotes a system's ability to alert, diagnose and react to system malfunctions, if and when they arise. Self-healing components or applications must be able to observe system failures and apply appropriate corrections (where possible). In order to automatically discover system malfunctions or faults/failures, the system needs to know what "the expected system behaviour" is. Autonomic systems must have knowledge about their own behaviour then they must have knowledge in order to determine if the actual behaviour is consistent and expected in relation to the environment. More challenging is the need for the self-healing module to evolve with the environment (Tosi, 2004). If the system is not designed with change in mind, or given adequate knowledge of the key elements that change, it will be more prone to failure. This would be much the same as a system engineer would need to analyse what components have changed before determining a fault, if any. They would not be successful in their job if they worked with the presumption that the last repair would work again and the system was the same as before.

Corporations rely and expect system "up-time" to be as high as possible. They design and build expensive redundancy into their systems to allow them to operate if a selection of hardware of software components fails. If computer systems were perfect, they would never fail. But no systems are perfect and will fail, but what if they had the ability to react and heal themselves? Hence faults would still occur as regular or irregular as they did and do, but now the system would react rather than sit and wait to be fixed.

Sufficient fault finding should not concentrate on present system condition alone, but should also be looking for future faults. Similarly, the SMART (Self-Monitoring Analysis and Reporting Technology) allows users to detect HDD's defects in advance (Samsung, 2007), and thus removes the "infected" component before it actually fails and causes expensive downtime.

SMART technology resembles a jigsaw puzzle—it takes many pieces, put together in the right way, to make a pattern. Understanding failures is one piece of the puzzle. Another piece of the puzzle is the way attributes are determined. Attributes are reliability-prediction parameters, customised by the manufacturer for different types of drives (Seagate, 1999). Once a parameter response is detected as being high or higher than a defined parameter (parameters are set optimal ratings for individual components), then set within the expected tolerances, the “possibility” of disk failure increases, meaning the disk is flagged as a possible potential failure. One example of a stand-alone software application that uses SMART technology to monitor the “health” of a hard disk online, is inexpensive and very effective “active smart” (Ariolic Software, 2007), illustrated below.

Fig 4.0: SMART application illustrating a healthy disk



FUJITSU MHV2040AH, 40.0 GB
Current temp: 43°C Highest temp: 43°C

Disk summary

GOOD No abnormal drive attributes were detected. Your drive seems to be functioning normally.

Disk health: GOOD

Performance: GOOD

Temperature: 43 °C

Start date of monitoring: 2007/06/10

Estimated lifetime: Collecting information

Most recent log messages for this drive

2007/06/10 14:33:35
SMART monitoring started

This software monitors the hard disk for failures and potential degrading and prompt replacement before it fails (even alerting). When the hard disk state changes, the SMART system, integrated into the disk, notifies the user (Ariolic Software, 2007). However, this is not incorporated into the Windows operating system. Most hardware will detect disk changes, but only on reboot. So the end-user may not even be aware. This type of design is what S.H.A.D.E. plans to embellish. A simple user interface with effective operations and options is included for SMART abilities in its disk checks. There is no reason why features such as SMART cannot be built into operating systems like Microsoft's Windows, to enable your computer to alert or even make a proactive back-up at the first sign of disk failure. Your data would be much more secure and the computer more user-friendly if it "protected itself" and your critical data on its own, rather than waiting for you "the user" to detect a possible problem, and then make a back-up before the drive failed. If Windows detected, backed-up and then alerted the issue automatically, it could reduce repair time as well as potential loss of data.

Thresholds that help detect failure rates on one model of drive may not be sufficient for another. A comparison of car models can help explain this point. Certain model cars are equipped with four-wheel drive, but others are not. The architecture of the drives determines which attributes to measure and which thresholds are effective to use. Subsequent changes to attributes and thresholds will also occur as field experience allows improvements to the prediction technology, making the technology more intelligent over time and based on more data from the field (Seagate, 1999). This also depends on the fact that drives are made to standard design. In the world of technology where design and features can lead to increased market share, this could become increasingly difficult, as manufacturers will use technology changes and innovations to their own advantage rather than limiting their designs to standards. This alone is one argument towards why open source will always be more successful (not limited to financial gain) than commercial software, as the designers are more interested in making a better product than they are in making a better profit.

Although this technology can only be applied to desktop or cheap(er) server storage (such as MAID), it embellishes the same principles of similar solutions for its more expensive server cousins (SCSI) drives. SMART emerged for the ATA (IDE) [9]environment when SFF-8035 was placed in the public domain. SCSI drives incorporate a different industry specification, as defined in the ANSI-SCSI Informational Exception Control (IEC) document X3T10/94-190 (Seagate, 1999).

Before one can effectively detect a fault, one must define “What is a fault?”

Faults can be classified into three different groups: (Tosi, 2004).

• Design: made by hardware designers. In this case, faults are raised during the design process;

• Fabrication: manufacturing processes can introduce defects into the replicated device;

• Operational faults: they can be caused by device wear out or by environmental disturbance such as electro-magnetic interference or high temperature.

Errors and failures can be classified into four different groups which identify phases and steps for building software. In each step, typical errors or faults can be introduced. These groups are:

1. Requirement definition: made by software designers. In this case, application problems are not solved;

2. Design: inadequate problem analysis, inadequate knowledge of language design techniques, and incorrect implementation of algorithms can introduce incorrect method output, inconsistency, and unexpected system behaviour;

3. Implementation: inadequate programming knowledge and typographic errors can introduce incorrect output;

[9] ATA (IDE): interface standard for the connection of storage devices in computers

4. Runtime: inadequate knowledge of application can raise runtime exception or system crash (Tosi, 2004).

This reaction or self-healing is one of the goals of autonomic computing and is the primary goal of S.H.A.D.E. and of this thesis to explore the possibilities of self-healing systems and measure the effectiveness of a software engine in aiding a real system in being more self-sufficient. Since the first presentation of a self-stabilising algorithm, made by Dijkstra in 1974, progress made in this area has proved that it is one of most important and promising topics of fault tolerance (Brzezinski, 2000).

The concepts of self-management with the idea of dependable and fault-tolerant systems, as well as self-stabilising systems will be compared and contrasted. Leasing resources is a simple yet powerful and widely used mechanism for collection in systems. The system automatically frees a leased resource (i.e. a memory block) when the leaseholder does not renew within a defined time period (Hermann, et al., 2005).

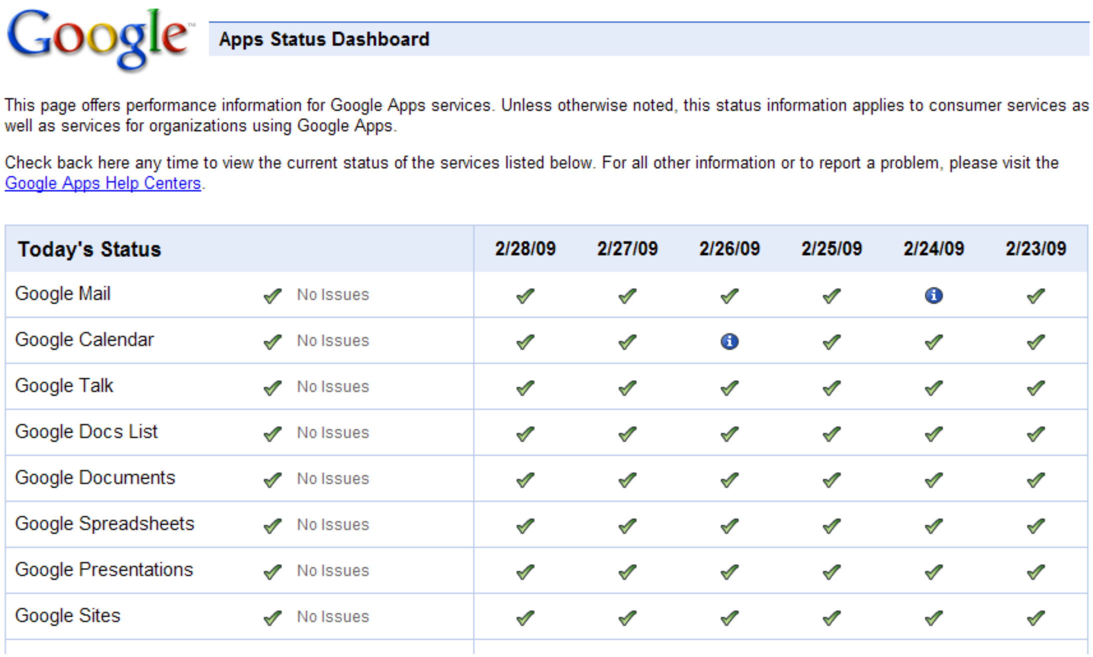
2.3 Self-Control and Self-Management:

Justification for self-management and self-healing is easy, as it will aid in reducing the need for human administrators. Less humans translates into less overheads in wages and other benefits. But the success of any self-controlling system resides in the quality of the software/system. Faults and bugs within this system will only help in potentially reducing the health of a system as well as potentially introducing or increasing downtime risks. A recent Google outage experienced with their Gmail[10] application was caused by what Google referred to as "routine maintenance". Basically, code was automatically fired with the sole purpose of improving the service/system by moving accounts to better geographical locations for the end-users. The code was adequately tested and took into account each and every factor; however, it overloaded the system, crashing it for 3 hours. In today's more volatile IT market, this was reported on a global stage. In the Google event, one data centre was overloaded and then it served to

[10] Gmail: Google's e-mail application.

take down others – one unhealthy system kicked in load balancing routines, which resulted in all centres being affected instead of quarantined. Unexpected side-effects of some new code that tries to keep data geographically close to its owner caused another data centre in Europe to become overloaded, and that caused cascading problems from one data centre to another (Cruz, 2009). Obviously, the code was not tested enough or all possible events were not factored into its design and implementation. The root cause of the problem was a software bug that caused an unexpected service disruption during the course of a routine maintenance event (Google, 2009). The issue was then further compounded by a phishing attack taking advantage of the outage and the efficiency of Google to get fully operational. Google charge for a professional flavour of their services, offering guaranteed “up-time”. Professional users are covered by a service level agreement that promises 99.9% “up-time” in any month. Google Apps includes a 99.9% uptime SLA (Service Level Agreement) (Google, 2009). As a result, Google became more proactive in announcing downtime to its users by building a downtime application into the dashboard of application suites. This application allows the end-users to read detailed reports on events as well as see a complete list of system outages and downtime, as illustrated in Fig 5.0.

Fig 5.0: Google Apps Status Dashboard: Illustrating the Gmail outage in February 2009



Apps Status Dashboard

This page offers performance information for Google Apps services. Unless otherwise noted, this status information applies to consumer services as well as services for organizations using Google Apps.

Check back here any time to view the current status of the services listed below. For all other information or to report a problem, please visit the Google Apps Help Centers.

Today's Status		2/28/09	2/27/09	2/26/09	2/25/09	2/24/09	2/23/09
Google Mail	✓ No Issues	✓	✓	✓	✓	ℹ	✓
Google Calendar	✓ No Issues	✓	✓	ℹ	✓	✓	✓
Google Talk	✓ No Issues	✓	✓	✓	✓	✓	✓
Google Docs List	✓ No Issues	✓	✓	✓	✓	✓	✓
Google Documents	✓ No Issues	✓	✓	✓	✓	✓	✓
Google Spreadsheets	✓ No Issues	✓	✓	✓	✓	✓	✓
Google Presentations	✓ No Issues	✓	✓	✓	✓	✓	✓
Google Sites	✓ No Issues	✓	✓	✓	✓	✓	✓

Technologies such as Cloud computing[11] could sufficiently reduce complexity for the end-user, because the user need only know what type or version of browser they are using (along with possible plug-in versions such as JAVA) rather than knowing hardware specifications, driver versions or patch set revisions. They would need to know less of how the system works, because those elements would be handled by the host. With such initiatives, self-managed and stable "back end" computer systems will be even more vital, hence self-healing and self-managing systems would become more prevalent.

2.4 Benefits of Self-healing.

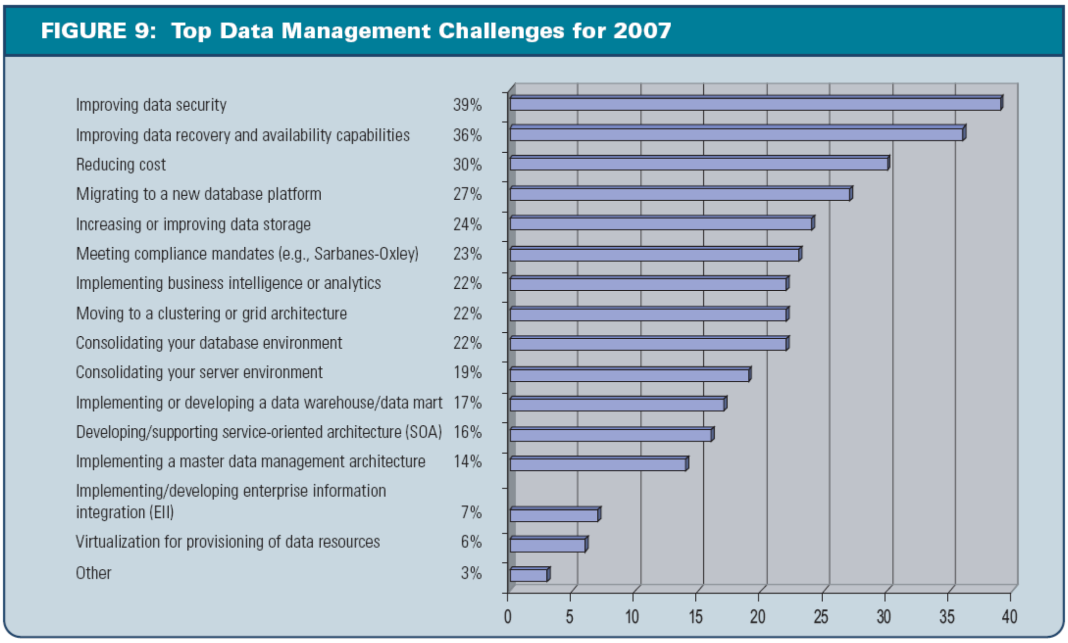
Self-healing is one of the four key properties of Autonomic Computer systems. Its ability can enable large-scale software systems to deliver services on a 24/7 basis and to meet its goals without requiring any human intervention (Czap, 2005). Self-healing shares similar needs, and thus, goals with self-stabilisation. This was conceived in 1974 by Edsger Dijkstra who defined a self-stabilising system as,

[11] Cloud Computing: A method of computing in which scalable and often virtualised resources are provided as a service over the Internet.

“regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps” (Dijkstra, 1974). The notion of self-stabilising systems in the 1970s was a very new problem. Dijkstra’s article on the subject was ignored but re-emerged again in 1984, when Leslie Lamport brought fresh attention to Dijkstra’s theories as they added a new dimension to possible ways of dealing with errors and their recovery. Lamport himself has been quoted “I regard the resurrection of Dijkstra's brilliant work on self-stabilisation to be one of my greatest contributions to computer science” (Lamport, 1984). Hence the theories and ideas are not new, but it is necessary to understand how systems have evolved since Dijkstra’s findings, in order to take the next steps required in building self-sufficient systems with even more modern and complex environments. It is possible to learn from elements of this approach; in fact, they are an excellent foundation and are regularly referenced today. Dijkstra’s theories are even more prevalent today as systems continue to grow in complexity and size. Errors and faults are more numerous and complex to solve, because today’s systems are bigger and far more complex.

Security, availability and cost containment are the top three challenges to be faced by Oracle managers over the coming year (2007). The roles of DBAs and other data professionals are clearly shifting to more strategic business priorities whereby professionals are spending less time on technical details and improving and spending more time on higher-value tasks for the business. This might include securing the database for greater compliance, or ensuring that the right people are able to access the right data. Data professionals also need to increasingly take an enterprise view, and manage hundreds of servers, rather than just one particular application (McKendrick, 2007). This translates into a bigger field of systems to watch, which means more manual operations and checks and further complexity and stress, all combining into possibly increased risks of downtime, performance degrading and mistakes through work overload.

Fig 6.0: Data management challenges 2007 (McKendrick, 2007).



They must understand that client charge-back and quality of service go hand in hand, so the definition and planning of data services in all their aspects is important to achieve this goal. This has been further challenged (especially from 2009 onwards) whereby IT budgets are increasingly shrinking. But the end-users and clients expect more value for their money, without effects to system "up-time" (McKendrick, 2007). This means providing minimal system disruptions with little or no financial investment. This challenge prompts the need for more "intelligent" system design. Self-healing IT environments can detect improper operations (either proactively through predictions or otherwise) and then initiate corrective action without disrupting system applications (IBM, 2001).

"Without effects to system "up-time"" is the key element of the quote above. This sounds relatively simple by definition but harder in practice if one was to implement it in a "real-world" system. This is where the self-healing system must be built with a certain amount of "self" awareness but it also needs a degree of overall awareness. Without such awareness, the design would run the risk of a "push-and-pull effect"; whereby components may attempt healing unaware of what effect their action could be taking on other components in the system. The push-and-pull effect is where the system falls into a "trap" of constant healing for actions that the healing system itself is causing. Apart from the fact that cells around the injury are able to adapt to a different function based on the new circumstances, it is their level of awareness that these cells possess that makes such healing possible (Mazzotta, 2000).

Self-awareness has been built into the S.H.A.D.E. engine not just for healing but for alerting of issues, working with the human administrator to solve certain key issues that the software has not been equipped with the knowledge to fix (much the same method as contacting technical support for assisted help). If these systems were to become completely self-healing, they would need to react by following certain defined rules and options, thus drawing from knowledge and experience. Since a "system" can exist at many levels, an autonomic system will need detailed knowledge of its components, current status, ultimate capacity, and all connections with other systems to govern itself (IBM, 2001).

A good example to simplify an action is flicking a light switch that blows a trip switch. As humans, we may repeat the action once or maybe twice before we know that we have to investigate and thus fix the cause, as we are now aware that there is a "problem" as the switch is not performing to our "expectations" on how we know it should work. If this was computerised and the self-healing system attempted and continued the simple action of flicking the switch; it would never fix the issue, just react to the problem, unaware of the result or effect of its actions. It would need to know when a failed action becomes a problem. i.e. it may be OK for an action to fail once, but not to continue to try and fail. A defined tolerance level of acceptable failure rate would allow the system to identify when it "may" be experiencing a failure and to carry out an action to resolve this failure. With this simple example, the systems would need to check for success and have a fault tolerance level set after attempted fixes. It must also be aware of results of its own actions as well as the success or failure of its actions. As IBM states in its Autonomic initiative: Initially, healing responses taken by an autonomic system will follow rules generated by human experts (IBM, 2001). From a design and implementation standpoint, the preferred way to enable repair in a self-healing system is to use an externalised repair/adaptation architecture rather than hardwiring adaptation logic inside the system where it is harder to analyse, re-use and extend (Griffith, et al., 2006).

Every day, computer users are wasting a significant amount of their own time and thus money maintaining their own computer systems. Modern systems are simply not robust enough and are too complex for an average user to fix. (Ryan, et al., 2008). The average American is wasting 12 hours per month - the equivalent of half a weekend - due to problems with their home computer (Kelton research, 2007). System designers have become too focused on implementing more features, rather than improving the quality of the end-user experience. Logically, there is no need to buy into the new version of a software product unless it does more or offers more features. The more features within a product means more lines of code and options that need both writing and testing. How many of these features does the average user need? These features also change the way in which the system is used, how easy it is to use, and how effectively the system (software or hardware) can be de-bugged before reaching the end-user. One could argue a lot of this problem is down to software companies release practices. They are rushing new versions of their software to market as quickly as possible to keep customers “wanting new versions”, even before the last release has been effectively patched or made free of errors. The global recession has changed this approach with major software vendors such as Microsoft deciding to reduce releases numbers for some of their software, planning to not develop or release office applications in 2009, but rather adopting to maximise return from their older version of the product, reducing the overhead of developing costs and focusing resources on other projects with greater return possibilities. But this could also be a result of market analysis whereby the vendor has discovered that they cannot release software with sufficient new features to warrant the customer investing in a new version of the product when the older one that they already own does enough to satisfy their needs.

The introduction of wide use of the internet has provided a perfect delivery system for software patches, thus reducing the impact on the customer as well as the cost to the manufacturer of making patches available and delivering to the customer.

Bugged software [12] and hardware is still a damaging and thus expensive business practice, both in relation to time and money. Modern software release practices' have made it acceptable to patch a product after it has been released to the market, because not all issues can be cost-effectively detected and fixed before the product is released to the end-user (Vanden Eynden, 2007). This practice only helps to further degrade the end-user experience by increasing the complexity of general usage. They need to be aware of what components make up the system as well as which versions of software are installed. In the future, this needs to be simplified and systems need to be able to patch and repair themselves without the user needing to know everything about the system. Currently, the user needs to know the version numbers, service pack and operating systems release they are working with (although this would also require user knowledge and consent to avoid potential legal ramifications). The need to know each and every component that makes up a system further erodes the end-user experience, as well as waste time that could be better spent using the system rather than fault finding (Kelton research, 2007). Does the owner of a car need to know every component of their vehicle to ensure it remains operational? The simple answer is no. But computer users are expected to be experts (with different levels of skill). Hence, modern systems are too complex by design and obsessed with complex functions (IBM, 2001; IBM, 2006). Attempts to automate operating system patches in the past have led to great concerns and complaints. By definition, if you change anything and the system fails to operate after the change, the change applied will be at fault (whether it actually is or not). If these types of system updates are to ever work, testing at the source would have to be considerable more effective to help reduce the need for further testing and bug fixes. In a perfect world, no bugs would be released to the public domain and the end-user would be left unaware of the actual updates, because the automatic implantation and installation would have no knock-on effects on their system. Today, companies require the ability to manually patch because it allows them time to test and apply patches/changes that they know would not impact their system (Dunne, 2007). Manual and thus controlled patching is also a requirement within industries that need change control procedures to be adhered to. These industries require testing and sign-off

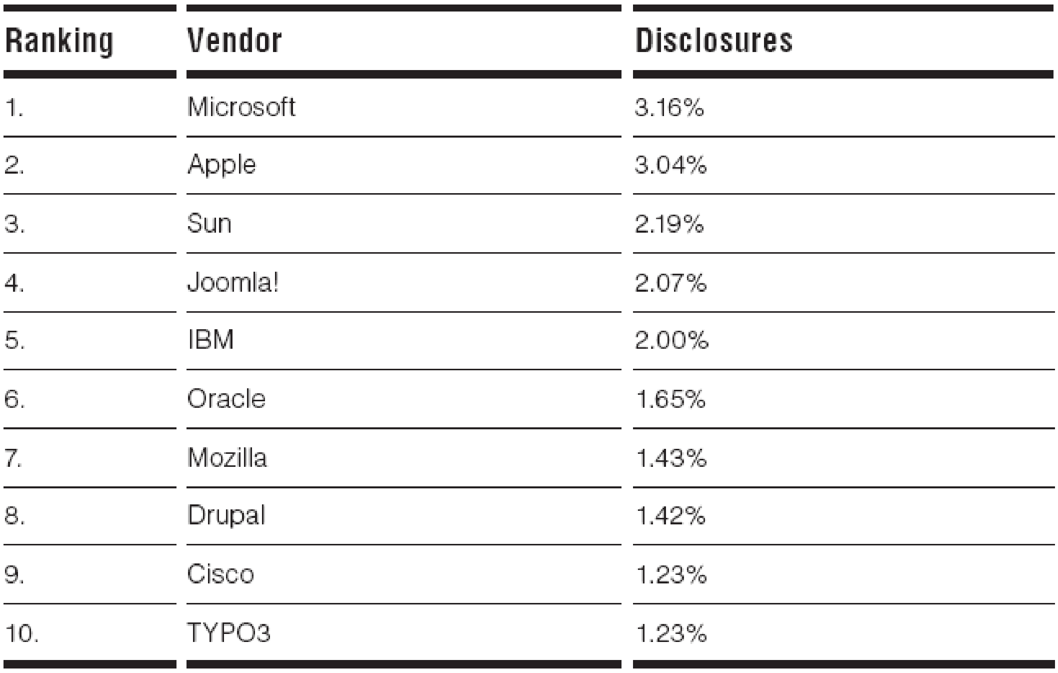
[12] Bugged Software or software bug: Software than contains an undocumented feature of fault that affects its ability to run as designed.

on all software changes before proceeding to live environment(s). The Food and Drug Administration (FDA) clearly outlines that software should be more tightly controlled because of its complexity (FDA, 2002). The need for automatic patching could also be determined by the type of software fault or "bug". An issue with a user interface may not be seen as a critical fix but a security hole in code may be classed as top priority, as the effects of exploitation could aid with security breaches or virus distribution.

If these high priority patches are released and left to the end-user to deploy or ignore, then there are huge risks to systems. In January 2009, 15 million Windows PCs were affected by a worm known as "Downadup" [13]. Although the patch had been released in October 2008, one in three users failed to install/apply the patch, which was further compounded by lack of adequate anti-virus software to deal with the outbreak (Randall, 2009). A more serious side to the argument of patching, either automatic or manual, is how effective modern patching/bug fixing actually is, either before or after a product is released. IBM reported that a high percentage of vulnerabilities go un-patched - "46 percent of vulnerabilities from 2006 and 44 percent from 2007 still had no patch by the end of 2008" (IBM, 2008). With such vulnerabilities, there is an even higher risk that a program is simply not doing what it should, but is introducing greater risks in virus, computer high-jacking, identity theft, phishing and other forms of data theft and intrusion. With the increasing global recession and thus increase in unemployment, especially for those with IT skill sets, vulnerabilities in computer systems that are left un-patched only increase the risk of using computer systems at all. This points to a strong argument that present day systems are not operating as well as they should. Even with constant updates and code changes, how effective would a computer be when teams of humans can address the issue?

[13] Downadup: a computer worm targeting the Microsoft Windows operating system that was first detected in November 2008.

Fig 7.0: Vendors with the most disclosures of vulnerabilities (IBM, 2008).



Ranking	Vendor	Disclosures
1.	Microsoft	3.16%
2.	Apple	3.04%
3.	Sun	2.19%
4.	Joomla!	2.07%
5.	IBM	2.00%
6.	Oracle	1.65%
7.	Mozilla	1.43%
8.	Drupal	1.42%
9.	Cisco	1.23%
10.	TYPO3	1.23%

As shown in Fig 7.0, the top ten vendors in the vulnerability listing are major players in the IT world. They have large pools of resources in software engineers and infrastructures and it would be expected that they would be more on top of the situation. The figures paint a different picture and strongly suggest that software is getting so complex that bugs and vulnerabilities are becoming a more common issue unless design practices change.

Even though it is well documented that complexity can affect the quality of the user experience, as well as increase the risks of bugs released in final versions of hardware and software, new systems still remain complex to use and faults are still found by the users and not the testers. The difference is (that) no software in the history of Microsoft development has ever been through the incredible, rigorous internal and external testing that Windows 2000 has been through (Foley, 2000). Yet it was reported to have been shipped with 63,000 documented 'defects' (Foley, 2000). Even with systems that are built on millions of lines of code, it is hard to believe that these levels of defects are acceptable in any product released into the public domain, especially one with such a large potential user base such as Windows. If software companies were to test and thus catch every issue, the time taken to develop these systems would increase, meaning greater expense before revenue is returned. Hence adding more features and thus complexity is reducing the overall quality of software, as well as tarnishing the end-user's experience (Vanden, Eynden, 2007). Thus modern systems are not yet self-managing, but are also not even working 100% when shipped (Ryan, et al., 2008). What elements of a system is the most important, quality or features? Features allow the company to sell a new version and make more money to develop the next, building a software development cycle with frequent returns. The end-user is, however, paying for something that is not as good as it could or should be. If development times increase, the end-user price would be in danger of increasing and people do not want that either.

One of the objectives listed with Microsoft's more recent operating system and Office packages (Vista and Office 2007) was to simplify the user interface (Ryan, et al., 2008);(Ballmer, 2007). This allowed the average user to concentrate less on how to get something done and more on the content they are working on (Ballmer, 2006). This is a complete change for the company, who once were more concerned with adding more and more options and now seem to be more concerned with making the experience easier for the user to interface with. Admitting that the typical user only knows (and thus uses/needs) 20 – 30% of their products, they are effectively admitting there is too much complexity and features in their own products (Ballmer, 2006). Effectively, these companies are coding features that most users do not need, adding to cost in development and resulting in inflated product costs. Modern systems such as Windows Vista (Stanek, 2006), SQL Server, Oracle (10g + 11g) (Wood et al., 2007), and SMART DB2 (Sterritt et al., 2005) are starting to list and implement self-healing options within their current and future software releases. These systems are shifting away from the trend of adding more features, and are moving towards enhancing the user experience through more simplified interfaces and overall easier management. This shows that the industry is beginning to accept that there is a problem with complexity and is starting to address the issue (Ryan, et al., 2008). With each additional feature within a system (or software package) there are potential security flaws that also need bug testing and patching. Unfortunately for the end-user, it is within the interest of software companies to get releases out as quickly as possible, to fix issues in the field. The sooner the product is launched, the sooner they can start to get a return on their investment. One more famous suggestion is the release of Microsoft's Xbox 360 console. It was stated as being listed with a hardware fault which caused the system to overheat and break. Microsoft "did right" by the consumer and offered a free repair for three years for each machine that failed. The cost of this gesture was one billion dollars. One could argue Microsoft can afford this money. It later emerged and was suggested that the company knew of the fault and made the conscious decision to get the machine to market anyway, beating its competitor and getting a stronger grip on a user base. What does this mean? One of the biggest companies in the world released a known defective product and took the gamble on paying for it later, through money and reputation, for one simple gain of getting their product into the market first. This could be viewed as a short-term loss and a long-term gain for Microsoft, but for the consumer maybe not.

This offers up another question. Are companies more interested in the return of investment than they are in producing and releasing superior products sch as more robust, secure, self-managing product that may in fact make our life's easier? The answer is that unless they work together through standards and sharing, they will remain more concerned with the performance of their own products. IBM took a big step in the right direction with its autonomic computing initiative (IBM, 2001). Autonomic computing design is based on the Autonomic (Human) Nervous System (ANS) (Sterritt et al., 2005) which handles some of the elements of the human body without conscious action from the human themselves. The computer system could manage and even repair itself without the need for constant intervention from the administrator. "To meet this autonomic selfware vision, systems should be designed with components that are allocated an autonomic manager" (Sterritt et al., 2005).

The human system works because of cross-component communication. Each element works with its own effective goals, but it is necessary that they all work together also. The Self-Healing Engine should, by design, consist of two parts: a problem determination component and a problem resolution component (Gao et al., 2004). This, in others terms, means; detect and repair. Most industrial monitoring systems do the first part well, with options to alert in many different ways, following custom and predefined tolerances that define what a fault is or could possibly be in the future. Their repair abilities however, mostly rely on the human element to receive the alert and act on it. Helpdesk applications attempt to improve the repair operation by assigning the human user to a potential repair and continuing to alert them until it is fixed, with options for escalation. All help in ensuring that someone will act on an alert but without the human element, faults will continue to be flagged but never fixed.

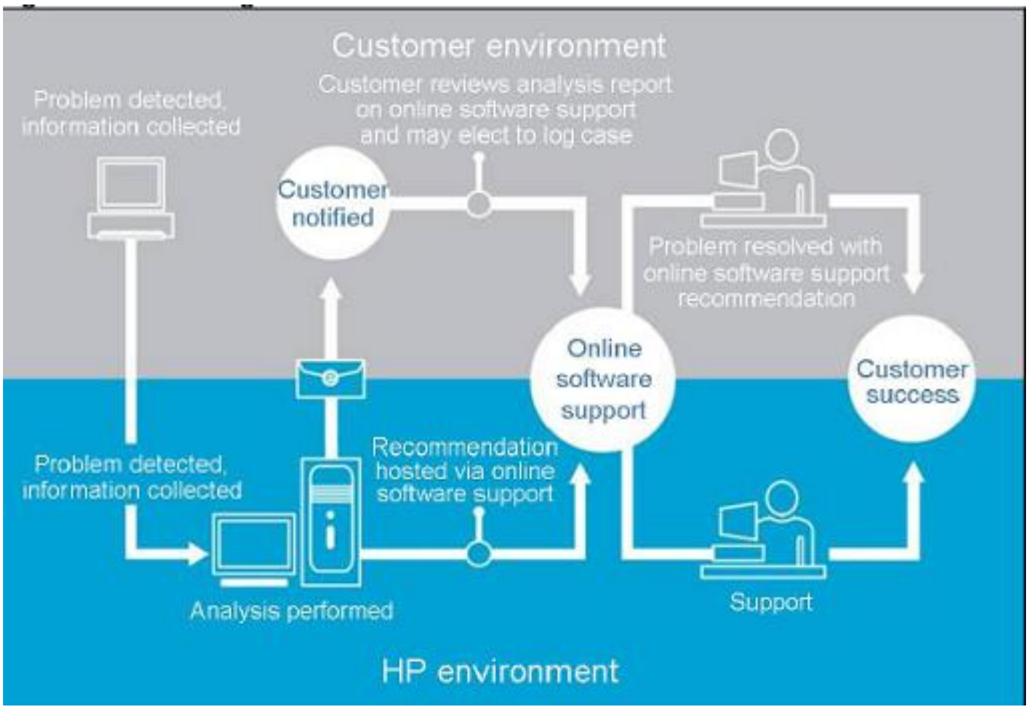
2.5 Future enhancements to self-healing.

In order to gain an understanding of the term “self-healing”, one must look to the field of Expert Systems. IBM’s ACI is definitely a step in the right direction, by setting standards and moving towards common goals aids in success. Autonomic computing is still in its infancy and most of today’s systems whicht contain autonomic options have only low levels of autonomic maturity (Brown, 2005). Standards will be very important in defining levels of health, as well as gathering requirements and expectations of what users and managers define as a healthy system. The study of the human as a system, as well as how humans interface, may provide the key elements in computer systems’ control and will help with future designs. However, until it is possible to completely replace the human operator with effective computer replacements, computer systems will remain dependent upon humans for the foreseeable future. A short-term solution would be people working on less complex systems due to autonomic elements that reduce tasks for healing and management.

The need for self-healing software to respond with a reactive, proactive or preventative action as a result of changes in its environment has added another requirement into the required list of capabilities that a self-managing system “must” have, that is, the system must be able to adapt to internal and external changes, to anything that can alter the running state of the system and also changes within the system itself. “The adaptations we are concerned with assist with problem detection, diagnosis and remediation” (Griffith, 2006). Many systems do not include such adaptation mechanisms, as a result these systems either need to be re-designed to include them or there needs to be a mechanism for retro-fitting these mechanisms. The purpose of the adaptation abilities is to ease the burden and thus decrease the human intervention in the management of software systems (Griffith, 2006).

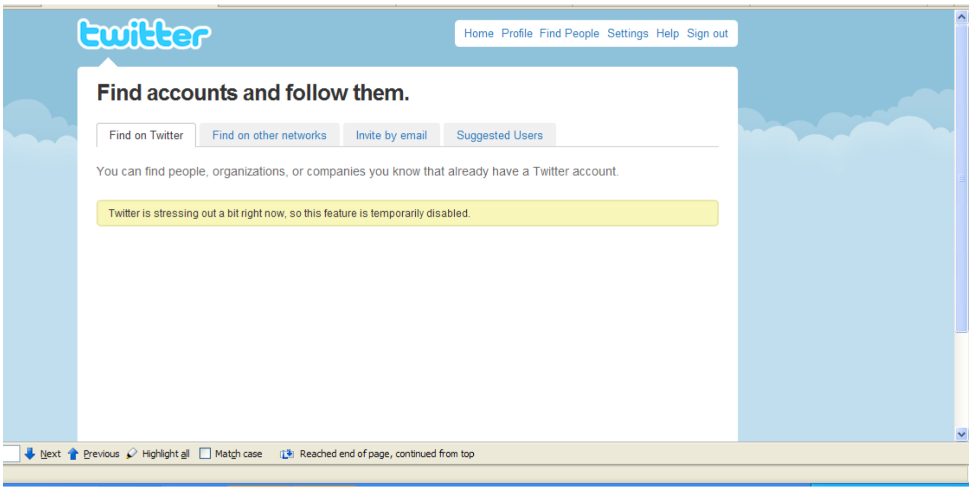
As mentioned, Oracle and other leading companies like IBM are moving towards and even including self-healing elements in their newer system releases. These are being used as sales features more than following IBM's vision to reduce complexity. Oracle has introduced features to reduce complexity but what lies under the hood is far more complex than previous releases. One service that is "jumping on the bandwagon" is HP's "Openview" Self-Healing services. This service basically attempts to simplify and automate some of the steps that need to be taken when a fault occurs, by pulling together information and proactively alerting the support centre. Is this a self-healing service? Absolutely not, the design could be used as part of a baseline structure but unless there is an intelligent computerised layer, the service still relies on humans at either end to both effectively report as well as diagnose/fix.

Fig 8.0: The structure of the service: (Hewlett-Packard, 2005).


Customer environment
Problem detected, information collected
Customer reviews analysis report on online software support and may elect to log case
Customer notified
Problem resolved with online software support recommendation
Online software support
Customer success
Problem detected, information collected
Recommendation hosted via online software support
Analysis performed
Support
HP environment

As the Fig 8.0 shows, it still takes human intervention at both end of the operation, hence it is still manual. From the time the problem starts, HP “Openview” Self-Healing services alerts you with a description of the problem, as well as relevant data collected about the fault and a proposed solution based on the knowledge documents from Software Support Online (Hewlett-Packard, 2005). Sun’s Solaris 10 operating system boosts self-healing capabilities, with abilities that diagnose problems and the results can be used to trigger automated reactions such as dynamically taking a CPU, regions of memory, and I/O devices offline before these components can cause a system failure. Solaris Fault Manager isolates and disables faulty components before they can halt/fail the system. In doing so, the manager helps ensure continuous service even before (human) administrators know there is a problem. (Sun Microsystems, 2004). One modern method that attempts to “cheat” “up-time” statistics can be observed in the way Twitter operates its social networking application/site. When the system as a whole is overloaded, Twitter dynamically disables features to reduce the load on the system. Why this is a cheating method is because the system remains operational and “up”, but not 100% with all features enabled. Hence, rather than having access resources that could be used to load balance a high load (i.e. more server storage or CPU power within its server farm), the application instead “decides” (manually or automatically) to disable features and thus reduce I/O until user operations and thus overall system load decreases.

Fig 9.0: The Twitter interface displaying disabled features because of system load



Find accounts and follow them.

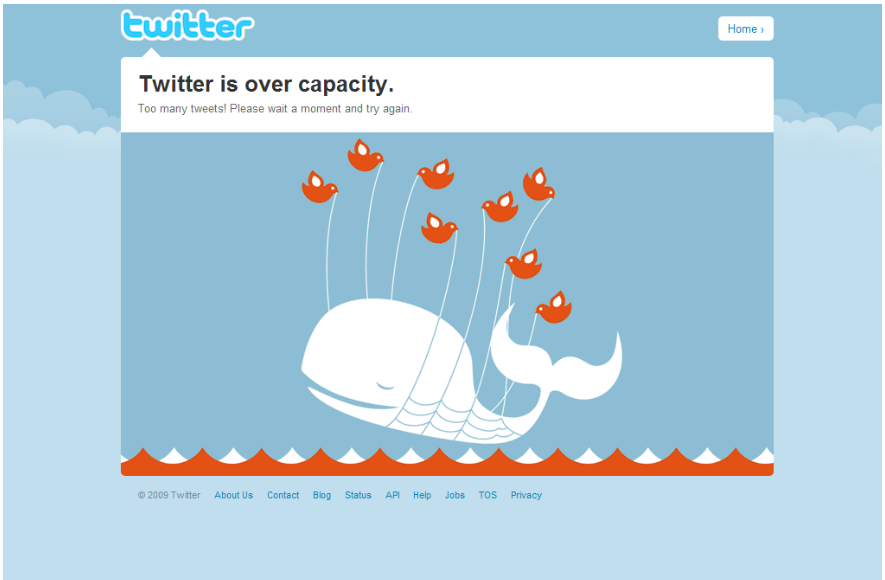
You can find people, organizations, or companies you know that already have a Twitter account.

Twitter is stressing out a bit right now, so this feature is temporarily disabled.

Reached end of page, continued from top

Twitter also (during severe load) disables all new transactions while the system is overloaded. While the system in fact is still online and up, it can't be used and thus should be deemed as being in a state of severe ill-health or down.

Fig 10.0: Twitter overloaded to the point that it cannot process any transactions.



Twitter is over capacity.

Too many tweets! Please wait a moment and try again.

Since users experience such "load reactions" continuously on Twitter, one could argue it simply isn't able to cope with its own user transactions and should be scaled upwards to support the spikes in I/O load. In a world where people demand quicker "always on" access, this level of service for a site/application of this sort could only serve to hurt its reputation and thus its user base.

The main reason Sun adopted the self-healing model was because they needed to improve "up-time". With businesses operating around the clock and demanding uninterrupted service, service availability is of paramount importance. Predictive self-healing delivers the next generation of available technology today, including features that keep systems and services running and simple for administrators. Over time, the rapidly-evolving ecosystem of self-healing components can help provide consistent, easy-to use, and always-available Sun systems (Sun Microsystems, 2004). With Oracle's potential acquisition of Sun in 2009, self-management is likely to become an even more prominent feature in future releases of the Oracle RDBMS family of products, especially within enterprise releases where improved system "up-time" is an integral part of data system operations.

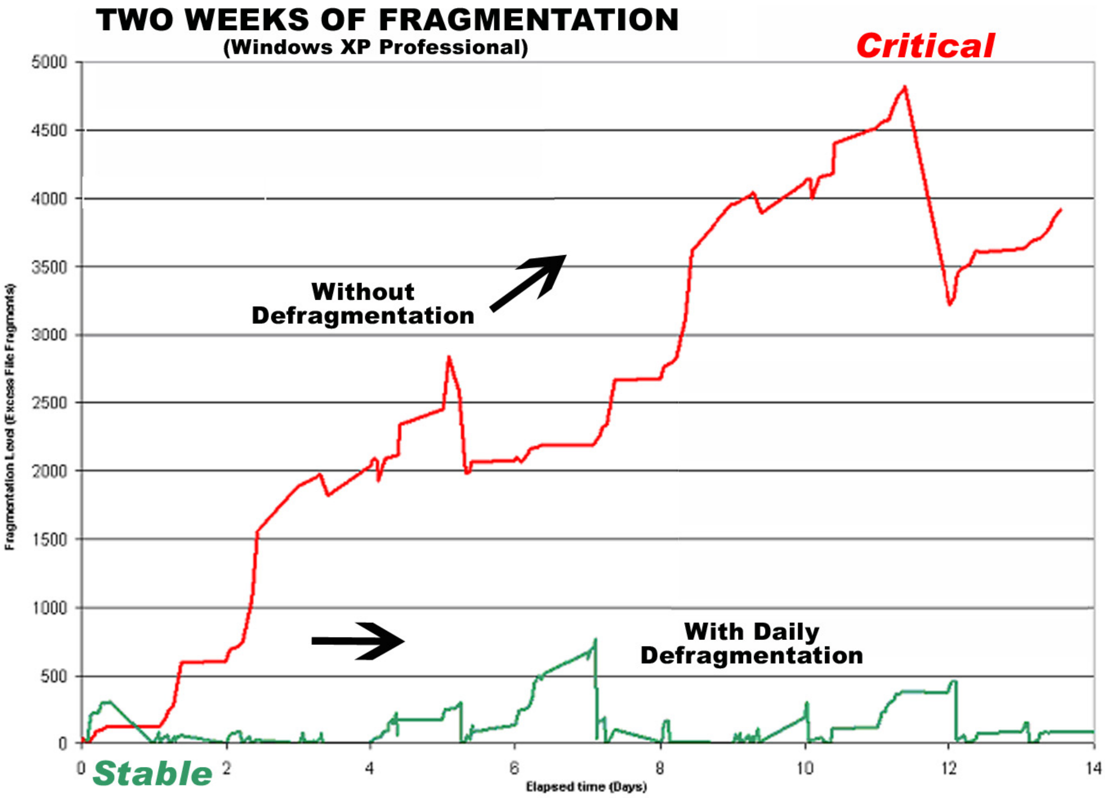
Microsoft is also boasting self-healing capabilities in its operating system "Windows Vista**",** stating that Vista has advanced self-healing capabilities to help "your system" maintain its health and an increased level of artificial intelligence that can help you troubleshoot when things go wrong (Stanek, 2006). Windows Vista also includes reliability improvements to the NTFS file system. Specifically, if Windows Vista detects corrupted metadata on the file system, it invokes NTFS's self-healing capabilities to re-build the metadata. Some data may still be lost, but Windows Vista can limit the damage and repair the problem without taking the entire system offline for a lengthy check and repair cycle (Microsoft, 2006). Windows also includes diagnostics to detect application crashes caused by damaged (disk) system files. If an application attempts to access a system file that is irretrievable because of a bad block (a read error on the disk that cannot be corrected), the application may crash. Windows detects these crashes, and silently repairs the damaged system file from a back-up copy (automatically). This diagnostic turns repeat crashes into one-time crashes with silent recovery (Microsoft, 2006). The most interesting element of this operation is the term "silent recovery". This in itself is self-healing in nature, as the operation occurs silently and without the end-user prompting or executing it. The system continues to operate and the user is left unaware there was a problem, outside of perhaps a single system crash that the system recovered from and later worked again without further outage. Brilliant, or is it? If this occurred on a disk that was not SMART-enabled then the system would crash, repair and crash again at a later date. Eventually, the disk would die completely. This could be avoided by informing the end-user of the problem which would prompt them to follow up on the problem and even replace the disk before it fails completely and everything is lost (because you've even been backing up corrupt files for the last week or two all your back-ups are contaminated). This approach includes the human user, even if it's just a simple alert to say what happened and what was done. The human user and "owner" of the system is then included in the repair and overall "decision".

Other software packages such as DisKeeper 2007 by Executive Software are also starting to create designs of their flagship projects that are based on self-managing models. DisKeeper was traditionally a manual tool for defragging hard disks and thus helping to improve disk I/O [14]operations by simply locating files in a better order on a hard disk; effectively reorganising the contents of the disk by placing files close together. Because Microsoft has never been too concerned with how optimal their products remain (a first install of Windows is a wonderful experience until two months down the line it chugs in slowness), Windows provided a defragmentation tool but DisKeeper has gained a market by doing the job better. Its latest version has steered towards a self-managing application by making the traditional manual task of defragmenting an automatic and continuous thing. Rather than wait a week or month to schedule a defrag of your drive, your drive is now set to always defrag. "In today's environment of bigger disks storing not only larger files but more files than ever before, the effects of fragmentation worsen markedly with each day's use. To keep up with daily performance degradation, disks must be defragmented daily." (Executive Software International, 2005). As demonstrated in the chart below, fragmentation levels rise on the unattended desktop resulting in performance being degraded each and every day (Executive Software International, 2005).

A bold statement one might say, as this type of daily management will have a disk I/O overhead by nature. But judging by the lab tests and graphs, not doing the task on a regular basis or making your system self-managing for this operation, you will experience reduced performance.

[14] Disk I/O: Represents input/output operations, in this example to and from the disk storage device.

Fig 11.0: Effects of disk fragmentation on performance (Executive Software International, 2005).



Automatic defragmentation is powered by the InvisiTasking technology introduced in DisKeeper 2007. User selection of Diskeeper priorities (e.g. CPU) is no longer necessary. InvisiTasking allows defragmentation of files on the fly as fragmentation occurs. This eliminates the need for both scheduling and manual defragmentation.” (DisKeeper Corporation, 2007). InvisiTasking is new technology designed to enhance multi-tasking and address some of the shortcomings related to the lack of “enough information” (DisKeeper Corporation, 2006). “To accomplish true transparency, one needs to be able to monitor CPU, memory and the more significant hardware bottlenecks of the disk drive and network. InvisiTasking takes a pro-active approach to instantly detect resource usage while maintaining complete granular control over its own activity, ensuring that it *never* pre-empts users or services.” (DisKeeper Corporation, 2006).

From a system overhead perspective, InvisiTasking-enabled applications can essentially be said to be “not existent” – they are truly invisible on a system

(DisKeeper Corporation, 2006). They clearly show that an InvisiTasking-enabled process does not interfere with other running applications (DisKeeper Corporation, 2006). In June 1999, the American Business Research Corporation of Irvine, California, performed a fragmentation analysis and found that, out of 100 corporate offices that were not using a defragmenter, 50 percent of the respondents had server files with 2,000 to 10,000 fragments and another 33 percent had files that were fragmented into 10,333 to 95,000 pieces. In all cases, the results were the same: servers and workstations experienced a significant degradation in performance (Kessler, 2009).

At least the application provides the option of setting up a computer with one less manual task to worry about. Now, if some of the major software companies such as Microsoft could concern itself less with features and more with optimal computing, by controlling and maintaining the registry, services, start-up programs and "bloatware" [15], computer may finally allow their users to be concerned with their use, rather than how to keep them running effectively or at all. The user experience with these types of problems have reduced over the years with the introduction of Windows XP (Vista has yet to prove itself in the field) which has helped the user experience by reducing the amount of time one must spend fixing issues or struggling to keep a computer running optimally or at all. Perhaps other key application developers could benefit from a similar process design such as "InvisiTasking", allowing management applications to manage them automatically and without affecting the end-user experience. Anti-virus applications are a prime example. Recently, I was forced to temporarily turn off my Network Associates "Virus Scan" service because it was using 100% of my laptop's CPU, rendering it unusable during a time period when I was answering a support call (within which every wasted second costs money). I fixed the issues, forgot to enable it again and got a virus. Would an application such as this be better designed for user experience if it ran "invisibly" to the user and managed itself?

[15] Bloatware: Term used to describe the tendency of computer programs to have larger installation footprints than needed, or to have many features included that are not used by the end-users.

As computer users, we spend far too much time keeping machines running and unless you understand the inner workings very well, you have no way of knowing if they are working optimally or even as they were designed. The average user must be concerned with drivers, defrags, scandisks, anti-virus, spyware, malware [16], identity theft, patching and the list goes on. This is only with basic workstations; the complexity and resources required to maintain database systems and web applications are far more numerous and require more effort and skill to keep these systems operational.

So there is still a road ahead for system designers to build systems that don't require "constant" human intervention. Until software and hardware designers work together to achieve this common goal, the end of the road will always remain a long way off. If modern applications manage to operate transparently to the user (whatever resources they consume), then many of the required self-management operations could be performed without the user needing to know they are even running. "InvisiTasking is specifically designed to address the "background" application to ensure it truly does run in the background and does not interfere with higher-priority processes such as transaction processing, print queuing and, quite frankly, anything else other than wasteful system idle time." (Kelton research, 2007). In the example of DisKeeper, the user would eventually forget about the need to defrag hard-disks (solid state or standard drives) as the need to manage manually will simply become a thing of the past. Moving towards the goal of the end-user not needing to know how the system remains operational, but is just concerned with using it.

2.6 Autonomic Computing:

One of the most developed and mature strives towards self-healing computing is IBM's autonomic initiative. The term "autonomic" is more identifiable with humans than computers. It is the human body's autonomic nervous system that has inspired autonomic computing. The human autonomic nervous system

[16] Spyware and Malware: software installed surreptitiously on personal computers to collect information about users and also designed to infiltrate or damage a computer system without the owner's informed consent.

controls various bodily functions, without the need for controlled or conscious actions of the human. This is the main aspiration of the Autonomic Computing Initiative (ACI). "Autonomic computing is just the next logical evolution of these past trends to address the increasingly complex and distributed computing environments of today" (Ganek et al., 2003).   As IBM states, the autonomic system must anticipate needs and allow users to concentrate on what they want to accomplish rather than on how to rig the computing systems to get them there (IBM, 2001). Autonomic computing is emerging as a potential architectural philosophy and design approach that promises to cope with complexity and scale up to the needs of today's distributed systems. Its fundamental goal is to increase the intelligence of individual components so that they become "self-managing," and thus reduce the need for human interaction (Tewari, 2006). Some software manufactures have already started a drive to reduce complexity in their systems. Oracle made a big leap in its 10g release, (Oracle, 2006) reducing the expertise needed to get the system running, by simplifying its installation and configurations, while offering a lot more features. Microsoft lists one of the objectives with its latest operating system and Office packages (Vista and Office 2007) was to simplify the user interface, allowing users to concentrate less on how to get something done and more on the content they were accessing. These products offer dramatic improvements that enable users to focus on content and tasks rather than the interface itself, making it easier to find information and access useful features (Ballmer, 2007). This is a complete change for the company, which once was more concerned with adding more and more options and now seems to be more concerned with making the experience easier for the user to interface with. It admitted that the typical user only knows (and thus uses/needs) 20 – 30% of its products, hence admitting there is too much complexity and features in its products.

These are all steps in the right direction, but they also present another problem. The administration has been simplified by offering fewer options to manage, with some self-healing facilities in memory management, while the same operations as before are carried out behind the scenes. Many older (and new) options are now hidden within an even more complex system. One potential future problem could be the administrator being down-skilled as a result of not being exposed to certain options/parameters on a regular basis. The administrator is forced to rely on the software vendor to provide support and expertise, that in older versions of the software they may have had the options available, and hence, the skills to remedy on their own. The system is actually more complex, but presents itself as less. As a result of built-in self-managing capabilities, additional automated administration capabilities are available that further streamline operations and reduce operational cost (Oracle, 2006). Although the statement was made in 2006, you can see how (with modern IT budgets shrinking) that in 2009 and beyond, IT managers could look towards software solutions of this type to simplify and just help reduce the cost of IT. One such example is memory management in Oracle. In previous versions of the Oracle database, the task was manual, where the administrator not only needed to monitor and tweak the memory available to the database with changing loads, but also had to be aware of what effects these changes had on other components of the database and thus database server. Oracle 10g [17] has this operation automated. Now if administrators choose to do so, they can let the database engine take care of this task and "free" up their time. On one hand, it's one less manual task to monitor and change; however, on the other hand, what happens when manual intervention is required? The administrator is in danger of not just knowing how to react to a required change in memory tweaking, but also no longer understands the finer fundamental elements of how memory management works. Because it is no longer a daily task and a need to manually intervene is so uncommon, the skill and knowledge gets lost.

One of the key self-management enhancements in the Oracle 10g database is Automatic Shared (SGA) Memory Management. This functionality automates the management of shared memory used by an Oracle Database 10g instance and frees administrators from having to manually configure the sizes of shared memory components. Besides making more effective use of available memory and thereby reducing the cost incurred of acquiring additional hardware memory resources, the Automatic Shared Memory Management feature significantly simplifies Oracle database administration by introducing a more dynamic and flexible management scheme that has the self-ability to adapt to changes within

[17] 10g: Oracle RDBMS (Relational Database Management system) version 10, where by the g represents the Grid feature(s) of the software.

the database system (Lahiri, et al., 2005). Autonomic computing systems have the ability to manage themselves and dynamically adapt to change in accordance with business policies and objectives. Self-managing systems can perform management activities based on situations they observe or sense in the IT environment. Rather than IT professionals initiating management activities, the system observes something about itself and acts accordingly. This allows the IT professional to focus on high-value tasks while the technology manages the more mundane operations (IBM, 2001).

Using a simplified view of autonomic computing, the goal could be confused with basic automation. On a simple level, it is accurate, but if you relied on an expert human operator to simply attempt the same fix when an issue occurred, then it must be an error that could be fixed with a permanent solution. If they attempted the same fix for different issues, they would fail. Hence, to make it automatic you would need to know what each and every possible issue was and put a fix in for each and every occurrence. Autonomic monitoring and reacting to alerts and change is more complex than basic automatic responses.

Autonomic computing lays out a vision of information technology in which systems manage themselves based on policies. With these policies are the new currency of interaction between people and computers, creating a new paradigm for interaction with autonomic systems (Kandogan, et al., 2008). This statement seems to suggest that policies are the key to future information technology developments. These are part of the puzzle, but only part. Other parts will be discussed during the course of this thesis, merging together into a collection of identified key elements which together may help with the success of autonomic computing. As discussed, the term “autonomic” is more identifiable with humans than computers, and should be, as it is the human body’s autonomic nervous system that has inspired autonomic computing. The human autonomic nervous system controls various bodily functions, without the need for controlled or conscious actions of the human. This is the main aspiration of the Autonomic Computing Initiative (ACI) (Ryan et al., 2008).

Although the autonomic initiative has been expectedly documented and championed by IBM, it has remained on the page as pure theory. It could in fact be labelled as pure "marketecture [18]", outlying and suggesting standards, but doing little or nothing to implement its own theories and designs. Autonomic computing in its present state is simply too broad and not defined enough to consider it a success, The initiative needs to be more specific and thus measurable and scientific to judge how practical it is in real-world practice. In essence, autonomic computing needs to move from the page to the "stage", to become more scientific and not just a collection of beliefs and ideas. It is, however, a solid approach in defining a problem. Combining autonomic ideals with specific applications and hardware and thus defining "real" and practical standards with actual software may in the future yield real benefits and thus actual success for the initiative and the computer industry as a whole.

2.7 Standards and policies: The need for open design standards.

Many IT infrastructures have components supplied by different vendors. For multi-vendor components to participate in autonomic systems, there needs to be a set of standards for the managed elements' sensors and effectors and for the knowledge to be shared between autonomic managers that describe the interaction between the elements of an IT system. Some existing and emerging standards relevant to autonomic computing include:

• Distributed Management Taskforce
• Common Information Model
• Internet Engineering Taskforce (Policy, Simple Network
Management Protocol)
• Organisation for the Advancement of Structured Information
Standards (OASIS)
• Java™ Management Extensions

[18] Marketecture: Any form of electronic architecture that has been produced purely for marketing reasons.

- Storage Networking Industry Association
- Open-grid systems architecture
- Web Services Security (IBM, 2001).

If the big players in the computer industry such as Oracle, Cisco and Microsoft continue to implement their own self-managing elements into their software and hardware designed, tuture self-managing systems will only succeed on a small scale. This is what we are witnessing today, with product such as Oracle 10g and SQL server 2005. On basic levels, the SQL server boasts self-healing capabilities in its installation. This essentially is similar in execution to what Microsoft has been doing with it operating systems since Windows 2000. Systems files will automatically check and replace them in order to maintain stability and prevent the system files from becoming invalid, thus is protecting the install of the software's "runtime" files. This method is now integrated into SQL server 2005 (Scalability Experts, 2005). Hence, the self-healing features of these systems are only concerned with providing new features in their own products. Recognising these demands, Oracle (with Database 10g and onwards) introduced a sophisticated self-managing database that automatically monitors, adapts, as well as attempts to repair itself (Kumar, 2006). Oracle 10g introduced a sophisticated self-management infrastructure that allowed the database to learn about itself and use this information to adapt to workload or to automatically opt for a selection of potential problems (Kumar, 2006). The system can adapt to changes (with specific events and guidelines) and make adjustments or tweaks that it believes will improve its own operation(s).

Adopting new software systems can sometimes promise reduced service desk costs, which was one market strategy for Windows Vista**.** Based on TAP participant experience, organisations that adopt Windows Vista will save an average of $11 (which is factored as 8%) per PC in service desk-related IT labour costs/time, when compared to Windows XP SP2 [19] (service pack 2). According to IDC research, TAP participants experienced fewer calls to the service desk, which is believed to be most likely related to the improvements in the reliability and

[19] SP1 and SP2: Naming conventions reflection with service pack is being used with the software to patch bugs and faults.

security features, and the self-healing capabilities built into Windows Vista (Gillen, et al., 2006). But a certain percentage of these may also be down to fresh installs of the operating system on the PCs, as Windows is notorious in becoming more unstable the longer it is on a machine. This is a particular problem that historically meant people needed to rebuild the windows machines at least every six months (or so), by wiping the operating system and re-installing a fresh copy, otherwise they would be forced to run a computer that operates slower and less reliably than the machine could do if they opted to re-install windows. If the SQL server database was integrated (using standards) with the healing abilities of the hardware, both could operate in conjunction with each other as well as be aware of what effect their actions are undertaking on items within the system other than just its own application. People with limited technology training should be able to manage complex systems, so reducing operational costs. It also opens up the possibility for those with non-technical expertise to manage systems according to policies and goals that are associated with non-technical aspects of the business (Kandogan, et al., 2008). Policies also provide an automation benefit that could improve the performance of current system administrators, enabling them to manage more systems at once, perform configurations faster and with fewer errors, catch system problems sooner, and reduce repetitive tasks (Kephart et al., 2003). Administrators must have a mental model of what the policies will do in a given situation to deploy them effectively. In this sense, the representation of the policies will be critical for them to be understood and used (Kandogan, et al., 2008). The human administrator may either forget or never learn how the system actually works, or simply not have a clear understanding of how efficiently it should operate (Kandogan, et al., 2008). The administrator with sufficient experience would be prone to lose skills through lack of use and the new administrator may never have the need to learn the skills in the first place. Thus the adequate skill set would be non-existent or unavailable to deal with an issue when the computer administrator fails to deal or is not operating correctly. One could argue this has little difference with today's "lean [20]" IT teams, where one administrator has the skills to deal with databases while another deals with networks, neither are cross-trained and are unable to step into different roles. This

[20] Lean: A process improvement discipline.

is true, as modern systems are more complex. But with systems becoming more distributed over time, the modern administrator needs to be more skilled to be more effective. Single domain skill sets are no longer sufficient.

When designing and coding any software with self-healing or self-management abilities' it is important to be aware of the foundations and designs adopted within other self-healing agents/engines.

The ability to dynamically repair a system at runtime based on its architecture requires several capabilities:

1. The ability to describe the current architecture of the system.

2. The ability to express an arbitrary change to that architecture
that will serve as a repair plan.

3. The ability to analyse the result of the repair to gain confidence
that the change is itself valid.

4. The ability to execute the repair plan on a running system
without re-starting the system." (Dashofy et al., 2002).

2.8 Learning from non-computer systems.

As mentioned, the ACI believes one can learn a lot from how the human body manages and heals itself. The human autonomic system controls functions of the body without conscious actions by the human themselves. It can heal, react to changes in temperature or to external threats, and control different systems through adapting to change. It does all of this automatically, each separate system interacting and reacting to the others' needs. This model is what the ACI is trying to achieve, by applying the various elements of the human autonomic system to computer components such as databases, hardware and middleware. This defines the ability to control them without external (human) intervention.

Some modern software monitoring solutions offer the ability to monitor elements of systems. Often these offer a selection of “closed loops” (Herrmann, 2005) software checks that fire at time intervals, checking and alerting specific issues as they arise. If an issue arises, the human is alerted and the software does its task. It is simple but effective. These solutions go some way to aiding with managing complex systems. But, what if rather than alert, they drew on previous knowledge for the issue and carried out pre-defined actions to fix the problem? This is the question that will be investigated in this research project. Working in much the same way as a human, the system will fix only what it has the knowledge to repair; when it does not have the ability, it asks for help and draws on the human element to intervene. This works much the same as humans; when they cannot identify a problem or repair it, they need to seek further knowledge or help from a third party.

2.9 Self-healing and learning from experience

Statistical models of large networked systems will let autonomic elements or systems detect or predict overall performance problems from a stream of sensor data from individual devices. At long time scales, during which the configuration of the system changes, we seek methods that automate the aggregation of statistical variables to reduce the dimensionality of the problem to a size that is amenable to adaptive learning and optimisation techniques that operate on shorter timescales (Kephart et al., 2003). Self-healing can only be successful if it is both aware of its environment as well as having the ability to adapt and draw from knowledge. The only way to gather this type of information is through calibration and execution within the same system environment for a defined time period. Many off-the-shelf monitoring solutions such as Quest Software’s Foglight and IWatch products, provide “canned” monitoring elements that will monitor defined elements of the system and alert on an issue when the elements operate outside the defined thresholds. These definitions would only be accurate and effective if run under a system during their design and conception stages, gathering information and data on what can go wrong and what needs to be alerted on. S.H.A.D.E. was been designed in much the same way; it has been operationally gathering data on system changes during different phases of development. S.H.A.D.E. defined the most common errors through its own use, allowed for designed watches and proposed heals to be constructed during this development time.

Chapter 3: Materials and methods used to design and build the healing engine.

3.1 Introduction

The approach outlined in this section is to detect and heal faults is based on an Oracle database. Oracle 9i RDBMS enterprise edition running on Windows servers is the platform of choice because of system/server availabilities and administrator experience on the platform. Funding would have been required to construct a Linux/Oracle environment for the experiments. The Windows environment was not only fully functional, but was in daily use in a real-world environment, which allowed for the introduction of faults and ill-health through normal system usage. The laboratory systems were also accessible in and out of standard working hours, allowing for queries and further development from either a local or remote workstation.

3.2 Defining health and repairs

The distinction between "healthy" and "broken" is often confused and is often indistinct, but in most cases can be greatly simplified by the statement: if a system is not operational, it is broken. This, however, is often not the case. If the system is up but not able to process anything or less than it was designed to do, isn't it still broken? This is where the entire "problem" of defining health starts and increases the time required to effectively design the engine to detect and repair faults. To a certain degree, it is a matter of opinion or subjective, but is also a matter of need. If the system performed to your satisfaction yesterday, it should in theory be able to repeat the same operation at the same speed a finite number of times. If it can't, it is not 100% healthy. But often (unfortunately), this satisfaction of performance speed is often tarnished by inaccurate comparisons and expectations. What people recall and expect are often very different to what actually happened in real time. These situations or realities can further divide the opinions and thus working relationships of the end-user with the system administrators.

Hence the initial problem was defining what could cause a system to operate ineffectively and in a state of reduced health. In order to build the initial designs for the engine, a full of list of potential system faults was drawn up. This list was based on a collection of 3rd party products combined with the experience of a database administrator. The list became of a collection of the top faults for a database system as a complete system. A listing of the elements that were classed as those that could not just lead to system downtime or faults, but also the elements that could cause the system to slow down. This approach was adopted based on findings in the literary review, to not just having the administrator's view of what causes a fault, but also the end-user.

3.3 Overall Research Strategy

To gain enough knowledge to build a software engine that can help reduce the need for human administrator interaction with a database system, by investigating success in self-healing computing to gain a better understanding of what can work. The final S.H.A.D.E. engine was designed with expandability in mind, allowing it to develop beyond this research paper and be used as a software solution for database management (with future builds).

Initially, a wide selection of possible options was selected that would all be considered options that could be used in a system to detect and fix errors within a database system. Any of these options that could be realistically designed and programmed with the required timeframe built into the system were possible. Any options that were not possible or posed too much of a potential risk were listed as potential future features but not introduced into the S.H.A.D.E. engine. Any of these options not introduced can be investigated and attempted in later builds of the system (unless listed within this document as not feasible).

3.4 Human vs. Computer: choosing the best element to use

Humans, although better at drawing on experience and intelligence (for the present time at least). are however more prone to distraction and outside stimuli.

Humans are not limited to work responsibilities and operations the same as computers. Humans also get tired and work in shift patterns, suffer from stress and fatigue, can also make mistakes and have to be trusted. Computers can be configured to work 24/7 (fault and breakdown dependent) without distractions from the outside world and other elements. Hence, there is a trade-off in advantages and disadvantages from human to computer monitoring administrators.

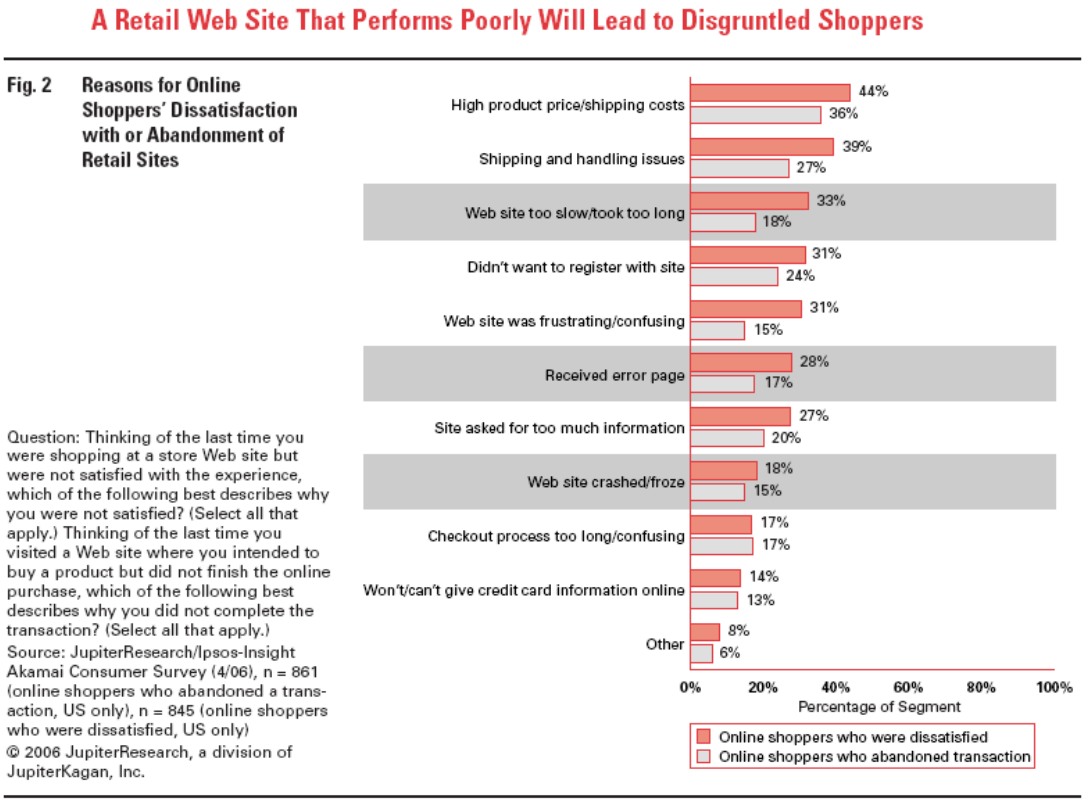
In stressful, complex, dynamic situations, the element of time criticality is one of the most distinctive features of decision-making (O'Hare, 1992). Stress and lack of time can increase the chances of making a mistake or choosing the wrong approach. The RPD model may provide an explanation as to why military commanders are able to make decisions faster than what would be considered normal using the rational choice model. RPD focuses on assessing the situation rather than considering multiple courses of action. More effort is said to be expended on understanding and assessing the situation, which results in a reasonably good course of action to take. In this way, the decision-maker does not generate a list of options; they make a decision and act upon it as soon as the minimum information is acquired (Daly, 2002).

Not only are humans affected by overload and stress, but there is also the distraction of lack of concentration during times of low workloads. Humans are also distracted by other elements of their life outside of their working environment. People do make mistakes, despite elaborate training and precautions, especially in time of stress and crisis (Borning, 1987); we were specifically interested in evaluating performance changes to high workload events as a function of the operator alertness state that preceded those events. If the availability of cognitive resources was a function of operator alertness during work under-load, then task performance should be directly related to measures of the operator alertness state during these under-load periods. In particular, higher rates of performance errors would be related to lower levels of pre-task alertness, as operators failed to attend to task components that they could normally handle (Murray, 1997). In other words, boredom can also lead to mistakes, much the same as over-working a person. In April 2006, Jupiter Research designed and fielded a survey to online consumers selected randomly from the Ipsos US online consumer panel. A total of 1,058 individuals responded to the survey. Respondents were asked approximately 15 closed-ended questions about their behaviours, attitudes and preferences as they relate to buying and researching products and services online. Respondents received an e-mail invitation to participate in the survey with an attached URL linked to the Web-based survey form. The samples were carefully balanced by a series of demographic and behavioural characteristics to ensure that they were representative of the online population (Jupiter Research, 2006).

With most transaction-based systems, slow response translates into loss of income, because frustrated users look elsewhere because of slow transactions. Each shopper type also had expected response time thresholds; longer delays resulted in the shopper leaving the site without submitting further requests, which resulted in missed sales and dissatisfied customers." (Kandogan, et al., 2008). Shoppers are likely to abandon a website if it takes longer than four seconds to load, a survey suggests (Jupiter Research, 2006). Hence people are getting frequently less patient with the speed of technology, expecting systems to run efficiently all the time. Roughly 75 percent of online shoppers who experience a site that freezes or crashes, is too slow to render, or involves a convoluted check-out process would no longer buy from that site **(**Jupiter Research, 2006). "Thirty-three percent of consumers shopping via a broadband connection will wait no more than four seconds for a Web page to render." (Jupiter Research, 2006). "Forty-six percent of online shoppers insist on a rapid check-out process, while 40 percent stated that quick page loading is critical to their loyalty." (Jupiter Research, 2006)

For retail websites, this basically translates into, you will lose customers if your site and systems are not optimised to operate quickly. People don't care or should care about system loads, the number of transactions you are dealing with. People are basically impatient and want everything to be easy and quick. If you fail to perform to their expectations, they will go elsewhere for their business; in much the same way as bargain hunters, people as consumers don't like to waste their time. Hence systems need to deliver a steady and constant level of operation. A system that slows operations to the point of delaying transactions as a result of overload is not adequate in today's world. As most of the time this occurs because the load increases to high for a system to cope, it hastens sufficient resources to deal with demand. This basically translates into not be able to cope well enough during its busiest times, when it could be making even more sales, it is in fact, risking losing those sales.

Fig 12.0: Retail website performance (Jupiter Research, 2006).



A Retail Web Site That Performs Poorly Will Lead to Disgruntled Shoppers

Fig. 2 Reasons for Online Shoppers' Dissatisfaction with or Abandonment of Retail Sites

Question: Thinking of the last time you were shopping at a store Web site but were not satisfied with the experience, which of the following best describes why you were not satisfied? (Select all that apply.) Thinking of the last time you visited a Web site where you intended to buy a product but did not finish the online purchase, which of the following best describes why you did not complete the transaction? (Select all that apply.)

Source: JupiterResearch/Ipsos-Insight Akamai Consumer Survey (4/06), n = 861 (online shoppers who abandoned a transaction, US only), n = 845 (online shoppers who were dissatisfied, US only)

© 2006 JupiterResearch, a division of JupiterKagan, Inc.

Just as the human element can make mistakes and thus affect the performance of a system (if involved in the running administration of such), humans also have a short attention span and low tolerance to boredom. Humans simply expect too much within modern times because they are exposed to so much more through media and computer use. They expect things to work first time and run at the same speed. The human operator has no need to understand the inner workings of a computer system (although often they are expected to); if a system slows, they only need to know it is failing to operate as quickly as it should. When they are dealing with on-line services, this can lead to loss of business and the operator will simply get bored and frustrated and move to the next website. This is much the same as a shopper moving to another store because the service is poor or slow. If they are queuing too long, the shop-owner runs the risk of losing their business. Hence the need for rapid web applications and online shop services isn't anything new, it just requires computer systems that keep themselves running at optimal levels of operations.

The human element of any system is thus more prone to both internal and external stimuli. A human is affected by other elements outside of the system. Humans have emotions and are dealing with other "things", such as their own lives and memories. How happy a person is (i.e. not under stress or presently dealing with anything traumatic) can affect how effective that human is in their operations. The performance of the system can also affect the human. If the system is operating poorly, it can lead to frustration and even stress, as the human struggles to get their work done. A computer component would not be affected in the same way. On November 9, 1979, a test tape containing simulated attack data, used to test the missile warning system, was fed into a NORAD computer, which through human error was connected to the operational missile alert system. During the ensuing six-minute alert, ten tactical fighter aircraft were launched from bases in the northern United States and Canada (Borning, 1987). Humans get tired, bored and affected by time of day, amount of time spent working without breaks and even how their home livess are affecting them (how much sleep they are getting, do they have addictions?). The accident at Three Mile Island [21] occurred because the operators adhered rigidly to a picture of the system that did not tally with the facts. This was due to "vigilance fluctuation" The accident at Three Mile Island took place about 4:00 a.m. It is well known that mental alertness is associated with the diurnal rhythm which characterises most physiological processes. This rhythm adapts slowly to shifts in the pattern of sleeping and waking hours. When someone changes to the night shift, his adrenaline secretion –(important for alertness),- is at the bottom of its daily rhythm during working hours and this

[21] Three Mile Island: Nuclear power plant in the United States that suffered a partial meltdown due to an accident in 1979

safety is seriously threatened when an operator on the graveyard shift is “out of step” with his daily rhythm. A person/worker, therefore, cannot be expected to function at peak level during a crisis (Frankenhaeuser, 1997)

Human performance is a function of willingness or volitional control, capacity, and opportunities. Research into vigilance research shows that humans cannot sustain attention for long periods of time. There is evidence that this negative degradation can be counteracted with exterior help for a certain amount of time. But a performance degradation over time is inevitable even with the most motivated operators (Wellbrink, 2003) Hence, any attempt to remove or replace human operations will reduce the risk of mistake or error due to stress, long working hours, distraction, fatigue or any other external or internal event or stimuli that can possibly affect the human element within a human/computer system operation.

A completely self-managing database will help to change the way enterprise data centres can and will be managed in the future. The automation of routine administrative tasks will enable DBAs to concentrate their time and effort on managing the end-user experience rather than supporting technology and simply “keeping the lights on” (Kumar, 2006). “However, IT systems exhibit different behavioural characteristics and pose different constraints on human operators. For example, nuclear plant operators deal mostly with continuous variables, such as fuel level and temperature, whereas IT system administrators control mostly discrete variables, such as communication port numbers. Nuclear plants are built once for a specific context, whereas IT systems are meant to operate under a wide spectrum of conditions, as configurations and workloads change over time as business demands change.” (Kandogan, et al., 2008). Hence, IT task monitoring is more demanding and subject to constant change. So the operator needs to adapt to change quickly or risk failing.

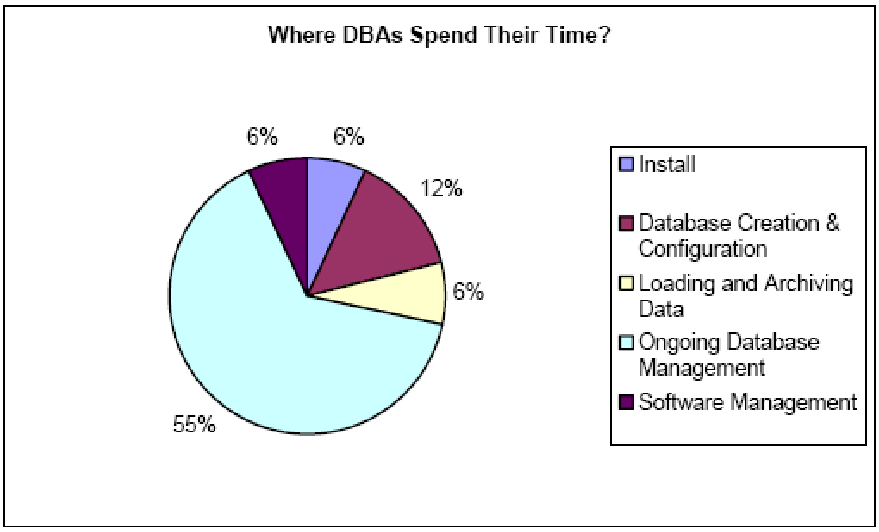
Just how intelligent or smart will computers get? Can we ever expect artificial intelligence to ever be advanced enough to outperform its human counterpart? In the turning point of man versus machine, the 1997 version of Deep Blue -- a chess-playing computer designed at IBM's Thomas J. Watson Research Centre -- defeated human world chess champion Garry Kasparov by 3.5 games to 2.5. The power behind IBM Deep Blue is an IBM RS/6000 SP parallel supercomputer equipped with chess-specific co-processors. The Deep Blue system is capable of examining 200 million moves per second or 50 billion positions in the three minutes nominally allotted for a single move in a chess game (IBM, 1997). One could argue that the computer had an unfair advantage as it was designed to play and thus win at chess. But the computer was built by a man playing a game originally designed by a man, hence the other argument would be the computer was playing within a human arena, thus it is the element at a disadvantage. But would Deep Blue deal with unknown entities or external anomalies? Of course not, it was running its primary programming; playing chess. So computers can better deal with single, clear and concise operations. . So was Deep Blue a genuine advance in artificial intelligence or an excellent public relations exercise? Well, a bit of both really. But it does show that a computer with enough power and proper design can beat a human being at their own game, literally. Kasparov stated that the computer did make a “human-like” error during one of the games and demanded a re-match, but IBM declined and retired Deep Blue. Should designers try to make artificial intelligence by duplicating how humans do it, or instead try to exploit the particular strengths of machines? Humans are slow but exquisitely good at pattern recognition and strategy; computers, on the other hand, are extremely fast and have superb memories but are annoyingly poor at pattern recognition and complex strategy. Kasparov can make roughly two moves per second; Deep Blue has special-purpose hardware that enables it to calculate nearly a quarter of a billion chess positions per second (Stork, 1998).

Controlled psychological experiments have shown that human chess masters are far more accurate than non-chess players at remembering chess board positions taken from real games, where the placement of pieces arose in strategic play and represented meaningful tactical positions. However, these masters were no better than non-chess players at memorising random arrangements of pieces. Chess masters remember positions based on certain patterns, alignments and structure whereas, of course, computers have no difficulty remembering -- storing -- all the games or random arrangements ever made and need no "meaning" in the placements (Stork, 1998). There are other differences, too. Humans are also affected and driven by emotions. Computers don't get tired, and don't have "bad" days. But they do break down (Stork, 1998). It’s not to say when you have a human involved that in fact there is a “weak link” within the system; on the contrary, humans can adapt better to situations and changes, can draw on knowledge and make decisions far better than present computer system equivalents. But humans are not better at repetitive, mundane tasks over long working periods. During times where boredom, fatigue and stress can interfere with their operations and decision-making processes. Human errors play a major role in accidents such as car or airplane crashes. Performance effectiveness depends on several factors which are described in the next section (Wellbrink, 2003) Hence, when a human is “at the wheel”, designers must be aware of their limitations.

3.5 Building the S.H.A.D.E engine – from first to final build.

Before any coding could be undertaken, present systems in the marketplace needed to be looked at in order to gain both an understanding of what was needed as well as what was already in place. The major challenge facing IT managers in the 21st century is an increased need for system availability. A simple enough challenge, but with the challenge of reducing costs (normally through manpower); the challenge can put a strain on any support staff. S.H.A.D.E. can help cut costs by reducing the manual workload of the administrator without reducing the quality of service, effectively allowing the administrator to manage more systems by not being on the “fire fighting” mode continuously. A database administrator’s time is often balanced amongst the chief operations of: tuning, space management and back-up/recovery operations. According to a survey conducted by Oracle, DBAs typically spend about 55% of their time performing these activities (Kumar, 2006). Hence, any time recovered by automating these tasks, makes better use of resources as well as saving time and money.

Fig 13.0: Breakdown of DBA time (figures taken from same survey) (Kumar, 2006).



Oracle's goal with 10g (release 2) was to build in more self-management options to tackle this very issue. These options are geared towards the database's operations only. And not all features are available "out of the packet". You must further invest in their performance packs, adding expense into each database that you use the options on. A lot of new enhancements to undo management make it easier to use than ever. The enhancements are based on a self-learning system that can automatically size an undo tablespace [22], dynamically tune undo retention, and provide fast ramp-up during sudden bursts of activity (Ganesh, et al., 2005). "Space Management functionality in Oracle Database 10g is a key contributor in making it a self-managing database. It is designed to have self-awareness and self-learning built into the core database engine. That, combined with intuitive, integrated, uniform and easy-to-use Enterprise Manager User interface, makes the task of space management in the Oracle Database 10g, significantly easier and more powerful than ever." (Ganesh, et al., 2005). Building upon the data captured in AWR Oracle Database 10g includes the Automatic Database Diagnostic Monitor (ADDM), a holistic self-diagnostic engine built right into the database. Using a medical analogy, using ADDM is very much like visiting your General Practitioner. It looks at the whole system, gives a diagnosis and then either

[22] Tablespace: A storage location where data underlying database objects can be stored.

suggests treatment itself or it may refer you to specialists, other 10g advisory components such as the SQL tuning advisor (Wood, et al., 2006).

The performance packs and said options are also only compatible with the more expensive database flavour: Enterprise Edition. Hence, expense on top of licence expense. This is simply a backwards business strategy in a modern world. But companies like Oracle measure success of their level of growth. “Oracle Diagnostic pack 10g includes a self-diagnostic engine built right into the Oracle Database 10g Kernel, called the Automatic Database Diagnostic Monitor (ADDM). This is a revolutionary, first-of-its-kind performance self-diagnostic solution that enables the Oracle 10g Database to automatically diagnose its performance problems, thereby completely liberating administrators from this complex and arduous task.” (Oracle, 2005). This translates into ‘We know of a better way that could be included with our database engine but rather than enhance our primary product to the point that it will save you money and be easier to manage, we are instead going to package it as a separate product that will insure further licence costs, if you indeed want to get the full potential from the product you’ve already paid for’. A bit cynical perhaps, but factual. However, this is typical of many of Oracle’s licence models. They design strong products, but have nightmare licence models to administrator and budget for. They encourage their customers to play it “cheaper” and not make use of their database products. A company can reduce costs by bundling their databases onto single servers with low amount of CPUs to save money. But the flip side is that they are using an advanced system without enough resources to drive it sufficiently. Considering one of the main goals of IT groups these days is to reduce costs and follow a lean initiate, this doesn’t mould with a partnership mentality that Oracle claim they want.

There are two aspects to Oracle’s manageability strategy. Firstly, it seeks to make each of Oracle’s products, particularly the database, as self-managing as possible so that they require minimal manual administration (Kumar, 2006). Self-management mechanisms exploit different approaches to execute a set of common steps, depending on goals and application domain.” (Tosi, 2004). The Oracle 10g Database is a step towards Oracle’s vision of creating a self-aware, self-leaning and completely self-managing database. A large development effort was put into simplifying every aspect of the Oracle Database 10g administration in order to serve the dual objective of enhancing administrator productivity and helping customers reduce their operational cost by 50% (Kumar, 2006).

Fig 14.0: Feedback loops in self-managed systems (Tosi, 2004).

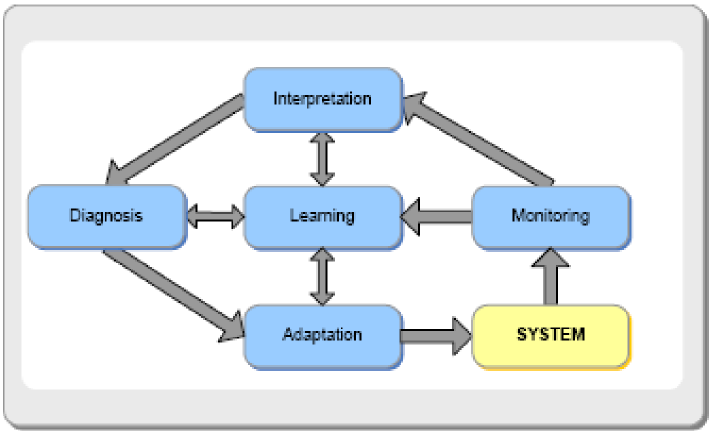


Figure 2: Feedback loops in self-managed systems

• Monitoring runs rules defined in a knowledge base (that supports decision-making at other phases) and checks if there is any inconsistency between knowledge and the current system behaviour. It provides statistical analysis related to system performances such as CPU usage, memory usage processes in execution or network latency. Dynamic data must be compared with standard data in order to determine if the actual system behaviour is not consistent with normal behaviour. This module must catch exceptions raised by system modules and also it must provide analysis related to the environment where the system is running. Monitoring mechanisms can observe either the behaviour of the system (internal monitoring) or the behaviour of the operating environment (external monitoring);

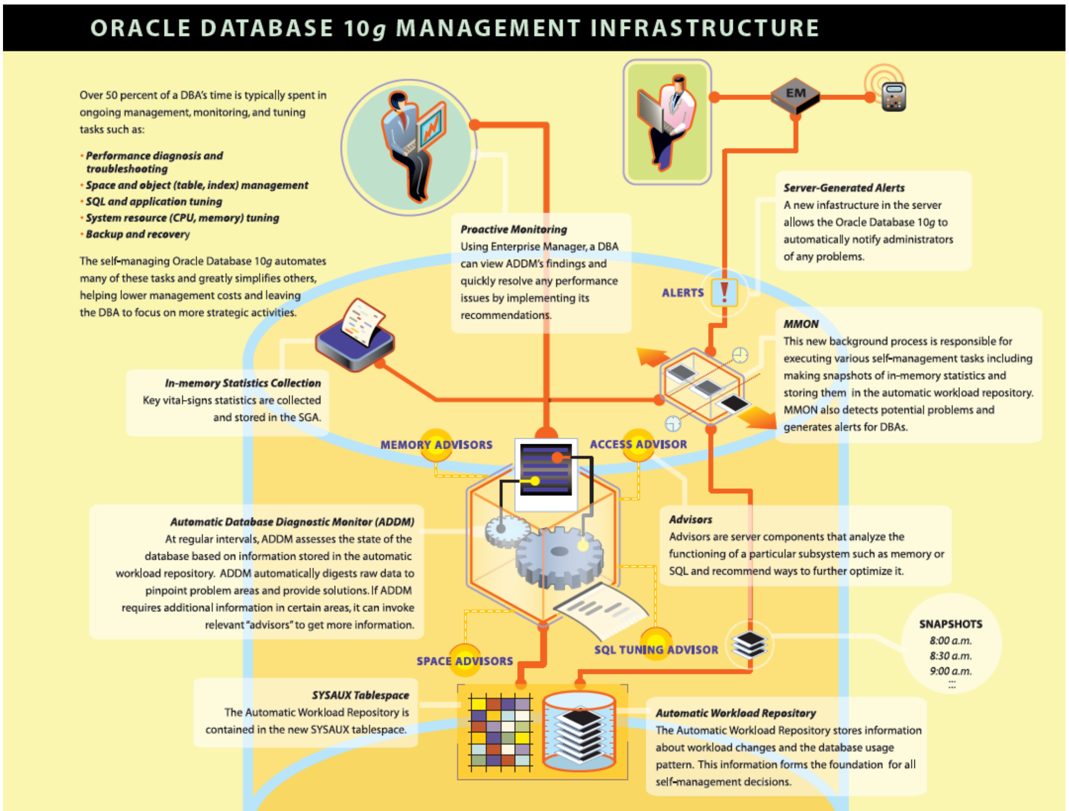
• Interpretation analyses data collected by monitor and verifies if there is knowledge related to the problem report interrogating knowledge-based module. If an appropriate problem report was found, the detection module tries to retrieve the problem resolution record. If a knowledge-based module does not contain a report for the specific problem, the detection module updates knowledge of knowledge-based module, adding the report through learning module;

- Diagnosis tries to find the causes of the problem and it verifies that applied solutions can fix the problem;

- Adaptation is a problem resolution module that tries to execute problem resolution cycle, starting from the solution record identified by detection module. This requires mechanisms to dynamically plan deploy and enact changes, to remove either the diagnosed faults or their effects;

- Learning creates and updates the knowledge base, acquiring new knowledge learned from data collected by the monitor activity (Tosi, 2004).

Fig 15.0: The Oracle 10g infrastructure (Oracle Corporation 2009).



ORACLE DATABASE 10*g* MANAGEMENT INFRASTRUCTURE

Over 50 percent of a DBA's time is typically spent in ongoing management, monitoring, and tuning tasks such as:

- ***Performance diagnosis and troubleshooting***
- ***Space and object (table, index) management***
- ***SQL and application tuning***
- ***System resource (CPU, memory) tuning***
- ***Backup and recovery***

The self-managing Oracle Database 10*g* automates many of these tasks and greatly simplifies others, helping lower management costs and leaving the DBA to focus on more strategic activities.

Proactive Monitoring
Using Enterprise Manager, a DBA can view ADDM's findings and quickly resolve any performance issues by implementing its recommendations.

Server-Generated Alerts
A new infastructure in the server allows the Oracle Database 10*g* to automatically notify administrators of any problems.

MMON
This new background process is responsible for executing various self-management tasks including making snapshots of in-memory statistics and storing them in the automatic workload repository. MMON also detects potential problems and generates alerts for DBAs.

In-memory Statistics Collection
Key vital-signs statistics are collected and stored in the SGA.

Automatic Database Diagnostic Monitor (ADDM)
At regular intervals, ADDM assesses the state of the database based on information stored in the automatic workload repository. ADDM automatically digests raw data to pinpoint problem areas and provide solutions. If ADDM requires additional information in certain areas, it can invoke relevant "advisors" to get more information.

Advisors
Advisors are server components that analyze the functioning of a particular subsystem such as memory or SQL and recommend ways to further optimize it.

SYSAUX Tablespace
The Automatic Workload Repository is contained in the new SYSAUX tablespace.

Automatic Workload Repository
The Automatic Workload Repository stores information about workload changes and the database usage pattern. This information forms the foundation for all self-management decisions.

Oracle lists the main "common" problems affecting database performance as:

- CPU bottlenecks.
- Poor connection management.
- Excessive parsing[23].
- Lock contention.
- IO capacity.
- Under-sizing of Oracle memory structures e.g. PGA, buffer cache, log buffer.
- High load SQL statements.
- High PL/SQL and Java time.
- High checkpoint load and cause (Kumar, 2006).

Other database products have also had some success in their move towards autonomic computing. Database systems, in particular, have been an early success within the AC initiative due to the evolution of the DBMS towards more complex features and a resulting move towards self-tuning. SMART DB2 provides for the reduction of human intervention (Sterritt, 2005). Where Oracle performance packs are limited to database resources only, S.H.A.D.E. will address these issues along with other database and operating system faults in a holistic design. Time will tell how effectively these changes in Oracle are working in the field. On one hand, they are masking complexity by making it easier for administrators to install, manage and recover the database. This, if effective, will save time and thus money. But on the other hand, with rewrites of optimisers and other features, migrating older systems may require more flexibility or risk expensive rewrites and thus validations. “The rule-based optimiser (RBO) is the archaic optimiser mode from the earliest releases of Oracle Database. The rule-based optimiser has not been updated in nearly a decade and is not recommended for production use

[23] Parsing: One of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure.

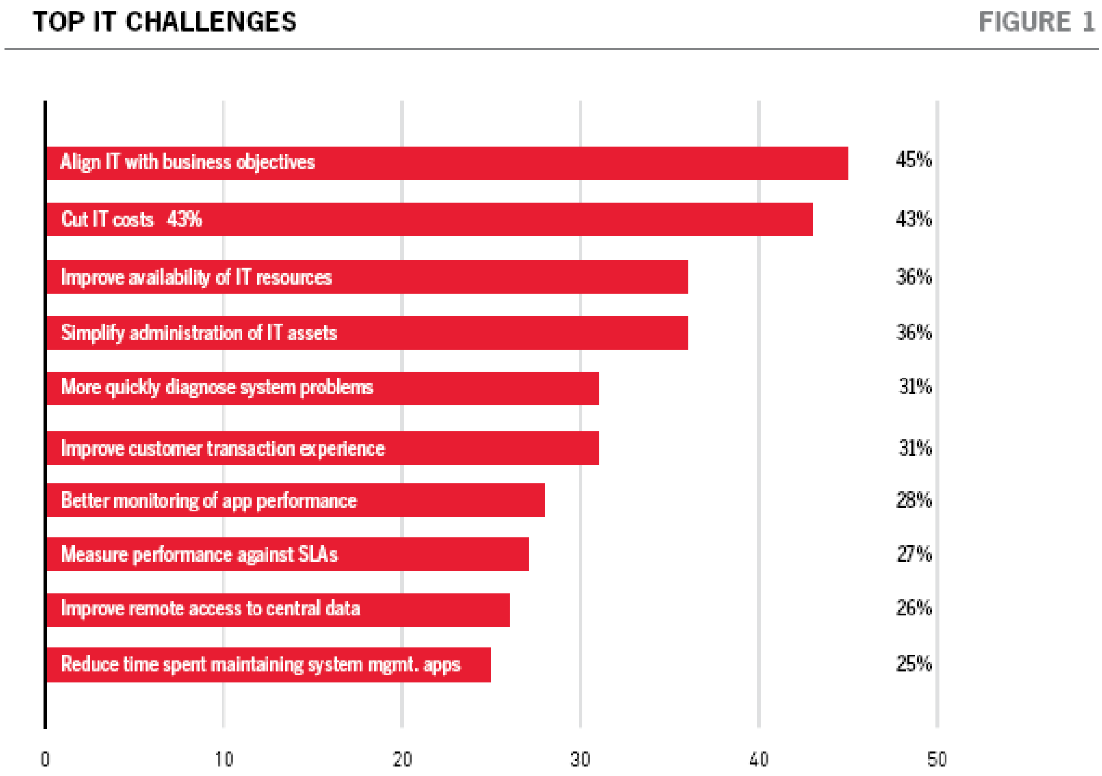
because the RBO does not support any new features of Oracle since 1994 (such as bitmap indexes, table partitions, and function-based indexes).” (Burleson, 2001).

Here, Oracle states an older “flexible” optimiser option is no longer available and will be no longer supported. This could possibly make it easier for Oracle to support single versions and enhance with less complexity. But it could seriously jeopardise backwards compatibility, hence simplicity should not be introduced at the expensive of flexibility. Computer information systems are becoming more feature-driven and complex with each passing year. “IT components produced by high-tech companies over the past decades are so complex that IT professionals are challenged to effectively operate a stable IT infrastructure” (IBM, 2006).

When asked about the top five IT challenges facing their organisations in 2006 (Figure 14) **(**BMC, 2006), the responses were:

- Aligning IT with business objectives (45%)
- Cutting IT costs (43%)
- Improving the availability of IT resources (36%)
- Simplifying the administration of IT assets (35 %)
- Diagnosing system problems more quickly (31%)

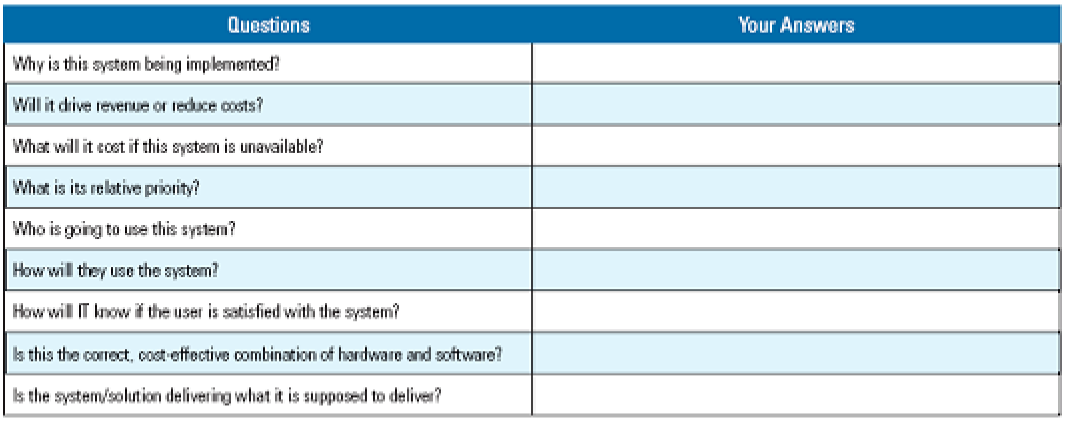
Fig 16.0: Top IT challenges (BMC, 2006).



Approving the cost and availability are two of the main concerns of modern IT business. Cutting cost cannot be achieved simply by reduction of head count as this typically just strains staff even further, putting availability at greater risk. The best approach is to reduce what the staff is doing on a daily basis, but minimising the complexity of their jobs. This helps to make better use of these valuable limited resources (people and time) to work on improving processes, rather than reacting to them. When alerting applications are in place, the typical scenario is an agent-based application, running checks and alerting the administrator staff. The administrators then react as they see fit. If they are bombarded with alerts and issues, they have to prioritise on the fly. This is based on skill and experience regarding the issue.

However, this is not as straightforward as it appears. Distinguishing between a faulty system and a system that's in a sub-optimal state isn't always easy (Hermann, et al., 2005). With S.H.A.D.E., the objective is to minimise the number of tasks the administrator has to manually fix, but also repairing common issues. This effectively frees up the administrator more to work on other tasks, as well as making it possible to administer more physical systems effectively.

Table 2.0: BMC: Sample Questionnaire (Armstrong, 2005).



Questions	Your Answers
Why is this system being implemented?	
Will it drive revenue or reduce costs?	
What will it cost if this system is unavailable?	
What is its relative priority?	
Who is going to use this system?	
How will they use the system?	
How will IT know if the user is satisfied with the system?	
Is this the correct, cost-effective combination of hardware and software?	
Is the system/solution delivering what it is supposed to deliver?	

The sample questions (formatted by BMC software) clearly define cost as a clear deciding factor in the modern business. But with any “improvement” changes, quality cannot come a clear second. It is simply to introduce a low-cost solution that “may” provide improved services. But unless the implementation and execution is measured in some form, how can this be accurately measured? For this reason alone, S.H.A.D.E. will be measured not only on its apparent success rates but also against another system with no solution in place. This will give an accurate metric not just on what the system heals, but also answer the question of whether it is actually needed. S.H.A.D.E. will also measure and store statistics that will all measure events that are not configured to be healed. Capturing the most common issues will allow a predictive model to be built, which will allow the highest problems to be captured and built into future releases of the engine. S.H.A.D.E. gathers faults and issues in much the same fashion as Google’s system health infrastructure. Although the Google system is responsible for a much larger farm of servers (and a different operating platform), the key structure of S.H.A.D.E. is very similar in design. “It consists of a data collection layer, a distributed repository and an analysis framework. The collection layer is responsible for getting information from each of thousands of individual servers into a centralised repository" (Pinheiro, et al., 2007).

The Google system does operate with Google's own advanced technology such as Bigtable, Mapreduce and Google file system (Pinheiro, et al., 2007). S.H.A.D.E., on the other hand, is designed as a stand-alone system using its standard operating system/database features to manage its operations. Related to the concern of coupling between the repair engine and the target system are issues of the interaction between the two and its impact on the target system (Griffith, et al., 2006).

- "How does the repair engine affect the repair of the target system? (Griffith, et al., 2006).

- "What is the scope of the repair actions that can be performed; for example, can we perform repairs at the granularity of entire programs, sub-systems, components, classes, methods or statements? Further, can we add, remove, update, replace or verify the consistency of elements at the same granularity? (Griffith, et al., 2006).

- What is the impact of the repair engine on the performance of the target system when repairs are/are not being performed? (Griffith, et al., 2006).

- How do we control and co-ordinate the interaction between the repair engine and the target application with respect to the timing of repair actions, given that application consistency must be preserved? (Griffith, et al., 2006).

Our problem with software doesn't stop with faults and bugs, but not all software even does what it was designed to do. There is little point building more advanced features into software systems, if the core elements don't work 100% of the time and to specification.

According to AusCERT, Australia’s Computer Emergency Response Team, the two most popular and deployed AV products fail to prevent 80% of new viruses. (Bloor, 2007). The cost of this type of performance failing can be quite staggering.

Table 3.0: The costs of mass viruses (as calculated by Computer Economics) (Bloor, 2007).

Year	Virus	Cost
1999	Melissa	($1.5 bn)
2000	I Love You	($8.75 bn)
2001	Code Red et al	($5.5 bn)
2002	Klez et al	($1.65 bn)
2003	Slammer et al	($4 bn)
2004	MyDoom	($4 bn)

Hence, money is not just lost through work time-wasted patching, updating and basically getting products to work as they should, but money is lost because programs are failing to do what they are designed to do. Virus outages and security flaws are becoming a serious issue with each and every computer user on the planet. Considering most of us are interconnected through use of the internet and faster broadband speeds, faults have the increased ability to cause even greater damage and thus incur much more expense.

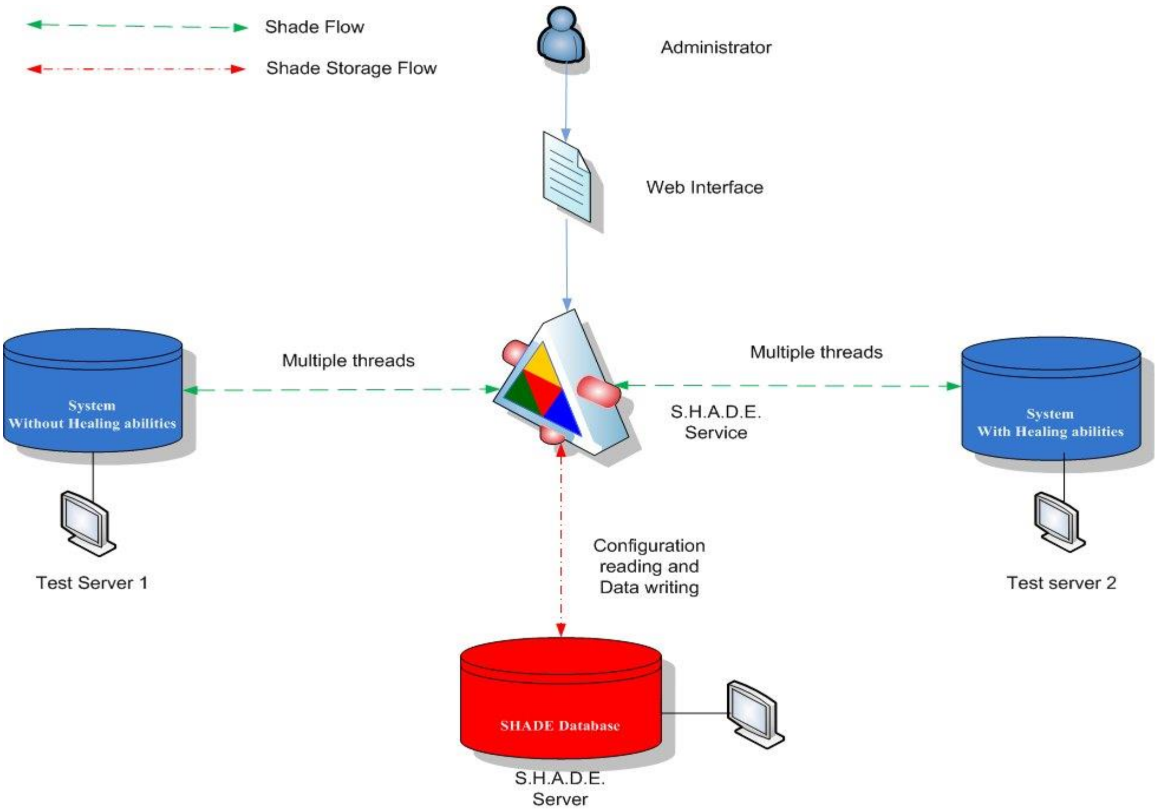
The Windows platform was chosen at the conception stage for three reasons:

1) Access to Windows-based servers and software on a 24/7 basis within a test environment used by real people.
2) To build my own personal knowledge and experience of the Oracle RDMS on the Windows platform.
3) To further enhance knowledge developed as part of a BSc project in which a fully functional mobile monitoring system was developed for the Oracle/Windows platform (Ryan, 2005).

The Self-Healing Autonomic Database Engine (S.H.A.D.E.) runs as a costumed coded agent that is housed on a separate server running as a Windows service. This agent polls the required servers at defined intervals using custom “watches”. Each of the watches are individually defined elements of the server operating system and database, which are checked and healed when and if issues arise. Each watch has its own autonomic features and is concerned with its own system elements. The system design needed to incorporate multiple sub-checks for certain elements to ensure the S.H.A.D.E. engine itself does not cause issues as a result of its own actions (Tesauro et al., 2004). Housing the engine on its own server, allows it to monitor multiple systems, while maintaining a low-level footprint on the systems it is watching and healing.

To conduct an effective experiment, two test database servers were needed (Test Server 1 and Test Server 2) with actual “real” daily usage and users. Test Server 1 has multiple S.H.A.D.E. monitoring threads running against various collection elements but has no fixes (either automatic or manual) as undertaken within the application (See Figure 15.0). Test Server 2 also has multiple S.H.A.D.E. monitoring threads running against its resources, but will also have healing elements incorporated within it (See Figure 15.0). The S.H.A.D.E. Server (as illustrated in Figure 15.0) runs the S.H.A.D.E. service while probing the other two test systems at configured intervals. The service runs from a parameterised coding configuration with an Oracle (10g) database. This allows new “watches” to be incorporated as well as altered without subsequent code changes. The S.H.A.D.E engine also stores results in the same repository, allowing for historical data to be queried whenever required.

Figure 17.0: S.H.A.D.E. Process Flow



The self-healing engine is called S.H.A.D.E. (Self-Healing Autonomic Database Engine). S.H.A.D.E.'s primary purpose is to monitor, alert and heal (where viable) events that occur on a database server in relation to operating system and database issues.

The S.H.A.D.E. engine runs on a Windows platform, running an Oracle 9i RDBMS. This allows extensive database administrator experience to be built into the healing engine, rather than focusing on a standard file server. The test systems do not house any form of mission critical data. The platform also allows the reduction in overall permutations that could possibly contaminate the experiment through lack of knowledge of working with an unfamiliar platform. To achieve self-healing and aid a system to run itself, the engine must be designed with solid awareness of the platform it will be performing on, otherwise it would run the risk of introducing other factors that could potentially harm the system (through operation using inadequate skills) (IBM, 2001). This is viewed as operating in a similar fashion to a human administrator interfacing with a computer system. It would not be practical or responsible to expect a Linux administrator to be an administrator over another system platform without building experience (and confidence) in the new platform first. Experience reduces risk and enhances the level of success (Ryan, et al., 2008).

A system cannot monitor what it does not know exists, or control specific points if its domain of control remains undefined (IBM, 2001). As well as awareness, another key component of any self-healing engine is the engine's ability to adapt to changes within the system and system components. A fire and forget design has the potential to cause greater harm to the system, by simply reacting to alerts and making changes without verifying if an element is actually fixed, or if the said fix has caused further problems (Ryan, et al., 2008). One example of this is memory management within database system. Assigning more memory to a database's cache could potentially yield benefits by the amount of information the system needs to read directly from the "slower than memory" disk(s), hence increasing the systems amount of disk I/O. However, if there is not enough sufficient memory for the operating system, more disk I/O will occur (through system trashing) and could potentially reduce performance and stability further than before the repair action was taken. Each and every potential cause and effect needs to be analysed and built into the engine to minimise if not eliminate risk with every action undertaken (Ryan et al., 2008). The ability to adapt is critical for self-healing systems (Kephart et al., 2003).

The S.H.A.D.E. engine was developed on a .Net [24] platform, making use of Client side JAVA to allow the agent to be managed (from a user standpoint) from a standard web browser with the applications configuration being built upon an Oracle repository database. This allows S.H.A.D.E. to operate using Oracle, as well as monitor/heal Oracle Databases. S.H.A.D.E. needs to be able to support repair on the test system during runtime operations. To do such, it requires several "abilities" to be built into the engine.

[24] .NET: A software framework running on Microsoft Windows operating systems, which includes a large library of coded solutions to common programming problems.

1. The ability to interrogate its own environments.
2. The ability to express an arbitrary change to that architecture that will serve as a repair plan.
3. The ability to analyse the result of the repair to gain confidence that the change is valid.
4. The ability to execute the repair plan on a running system without re-starting the system (Dashofy et al., 2002).

The goal of the experiment is to investigate whether the self-healing agent being proposed will have any impact on the level of human intervention required in the administration of the monitored systems (Ryan, et al., 2008). “The goal of autonomic computing is to create computing systems capable of managing themselves to a far greater extent than they do today” (Tesauro et al., 2004).

The decision to monitor both a healing-enabled and non-healing-enabled systems allows comparisons on resulting data between a system with self-healing and a system without self-healing on similar platforms, as well as user loads over a 24x7x365 [25] operating period (the servers remain operational as faulted). The key to measuring the experiment is the gathering of statistics and results for as many operations as possible. The embedded watches are based on database administrator experience over a ten-year period. Drawing on what can potentially affect a system’s reliability (based on historical knowledge) and thus operational capabilities. The healing elements were introduced only to the watches that could be implemented with minimal risks associated. The test could have simply focused on a reduced selection, but this would have required building a fault injection mechanism to ensure failures occurred (at some point), which would have meant the experiments were too staged and controlled, affecting the overall result (good or bad). Each watch stores audit data regarding its own operations. The system stores data on executions as well as problems diagnosed on the test systems.

[25] 24x7x365: Represents 24 hours over 7 days a week, for 365 days of the year. It signifies being online and available all the time.

A simple heal operation can have several “possible” paths or potential decisions to make to effectively complete its task at hand. This operation could also be dependent on the Oracle version, operating system version or even the database options in use. All could have an effect on the actual “possible” action one must undertake to “heal” a possible error or fault (Ryan, et al., 2008). This clearly shows some of the complexity regarding the decision-making process in S.H.A.D.E. The human administrator would normally be familiar with every element of the system and could make these decisions and changes with little effort or verifications. If an administrators were working on a system that they possibly had less exposure to and experience with, they would have to follow similar steps to the S.H.A.D.E. operation outlined. Already having the experience and the knowledge of the hardware/software/operating system layouts, allows decisions to be made more quickly, sometimes without the need to check or verify. But if they make an error, the results will be the same: performance degrading or system halts. IBM states in its autonomic initiative: “initially, healing responses taken by an autonomic system will follow rules generated by human experts” (IBM, 2001).

The initial build of S.H.A.D.E was concerned with connectivity and monitoring specific elements of the Oracle database system, selecting which elements would be better suited for a self-healing engine.

Table 4.0: Initial list of proposed monitoring/healing elements

Element (Watch)	Effect	Heal	Misc
CPU usage queue +	Reduce CPU if peaks for time period < 20 secs	Kill process if in list of possible to prevent killing other processes	Scale and graph over time. Score limits
Virtual memory usage	Peak at 1.6 (standard) 2.6 PAE	Check for extended memory usage and kill ghost sessions	Better to kill one user session with many ghosts than errors with all new sessions
Check for memory leaks	Check for 4031 error (leak)	Flush	
Check shared pool for fragmentation	Not enough continuous free space to execute SQL	Flush shared pool	
High number of active sessions (> 10%)	Caused by system resources or lock	Check and kill deadlock, wait for minute and repeat. Call high CPU heal	Log sessions and terminals – possibly kill

			sessions when large number for one terminal "ghost" sessions.
Network contention	Check for response time and % of network used	Check server for Network Hogging processes	List of none offenders – VNC etc
Check event log for ORA errors		Alert only	
Check alert log for ORA errors		Alert only	
Backup check (1)		Flag if any files "never backed up" – if system not shut down in one week. Generate a backup script and backup whole database to local disk with most space	
Backup check (2)		If files not in backup mode for time greater than 3 days – may a copy of set of archive logs to another disk volume.	Alert dba that failsafe back ran
Stats check	Slow response	Generate stats for schema with none – or too old – using estimate for large tables	
Check for corrupt blocks	Count and exports		
Schema protection	Export schema only	Protect database residing code	
High number of sessions		Keep track of average in table – if greater than 10% average run ghost session killer	Logged
Max sessions		If close to max processes run ghost session killer	Logged and not to alert parameter
Disk I/O	Set diskperf –y and gather stats	Check disk I/O counters + queue bottlenecks and other processes using.	Alert (could move)
Page file check		Compare to amount of physical memory on server and resize as needed x 1.5 times	
Large amount of invalid objects	Database residing code fails	Recompile objects and log failures	Log objects causing and alert to dba if compile fails
Datafile freespace < 10%	Error if expands	Set last file in set to auto expand if not set and check high water mark	Flag and log
Fragmented index	Slow queries and updates	Rebuild online	Flag as issue
Extent failures	Not enough space in files	Check if file reaching 32 gb if not and free disk space = expand. If close create another file (as naming of first)	Must alert the Dba regarding disk space, database increase and need for backup changes.
Max number of extents		Check if not locally managed – if not check max extent and only flag if within 20% (increase max)	Alert DBA

Hung sessions		Check Oracle for hung sessions hanging resources in deadlock situation. If found use pid to run orakill from server.	Alerted – as wrong pid = dead server
Blocking locks	Free resource	Kill blocking session if blocking object > 10 minutes	Logged
Blocking ddl lock	Timeout on object	Kill offending session	Logged
Open cursors < 5 % of max cursors	Will cause a session error	Increase open cursor proportional in inifile (remarked)	Flag DBA that restart needed (if alter system not version supported)
Objects in correct tablespace	Errors and fragmentation of system tablespace	Move objects if below certain size (especially if in system)	Flag to dba
Alive status	Uptime check	If system up for less than 7 days gather alerts bundle and run a health check	Send stats and logs to dba
Manage alert and event logs	Harder to read and may lose alerts	Check size < megs, save to back dir as date and flush old. Keep specific ora- and system alerts in repository table. Date and time for historical analyze of worst events	Alert dba of new event with detail and contents of table in html format.
Security check	Lock none used accounts	Check for list of accounts that are required to remain locked – lock if found open	Log dba
Pinning code	Pin code to reduce loading	Make a selection of most frequently called packages and pin as needed – set limit in packages and pool usage	Log to DBA
Large sorts	Monitor large sorts	Shift users to different temp file (create 2nd if needed) with large size to reduce usage and fragmentation	
Memory request failures	Errors	Resize pool, cache etc	Flag to dba
Level of disk reading	Too high	Increase cache and flag sql offenders	
Log switch levels	Too many per hour	> 5 – than increase logs and drop old < 5 decrease size to insure more archiving and easier recovery. If no switch in an hour – force it.	Flag to dba
Check for new datafiles/tablespaces	Possibly not in backup scripts	(will be flagged In backup check) – backup controlfile to trace and copy trace file into database repository – possible to recreate with new files	
Flag tables/files with logging off		If file not temp – turn on logging for tablespace – if objects needed – turn on logging	Alert dba that a fresh backup is needed
Recommend tables for rebuild	List tables that are largest with most extents and fragmentation	Flagged as rebuild candidates	Flag dba
Monitor latch waits	Internal locks	Decrease cpu load, free lists in key tables	Flag dba

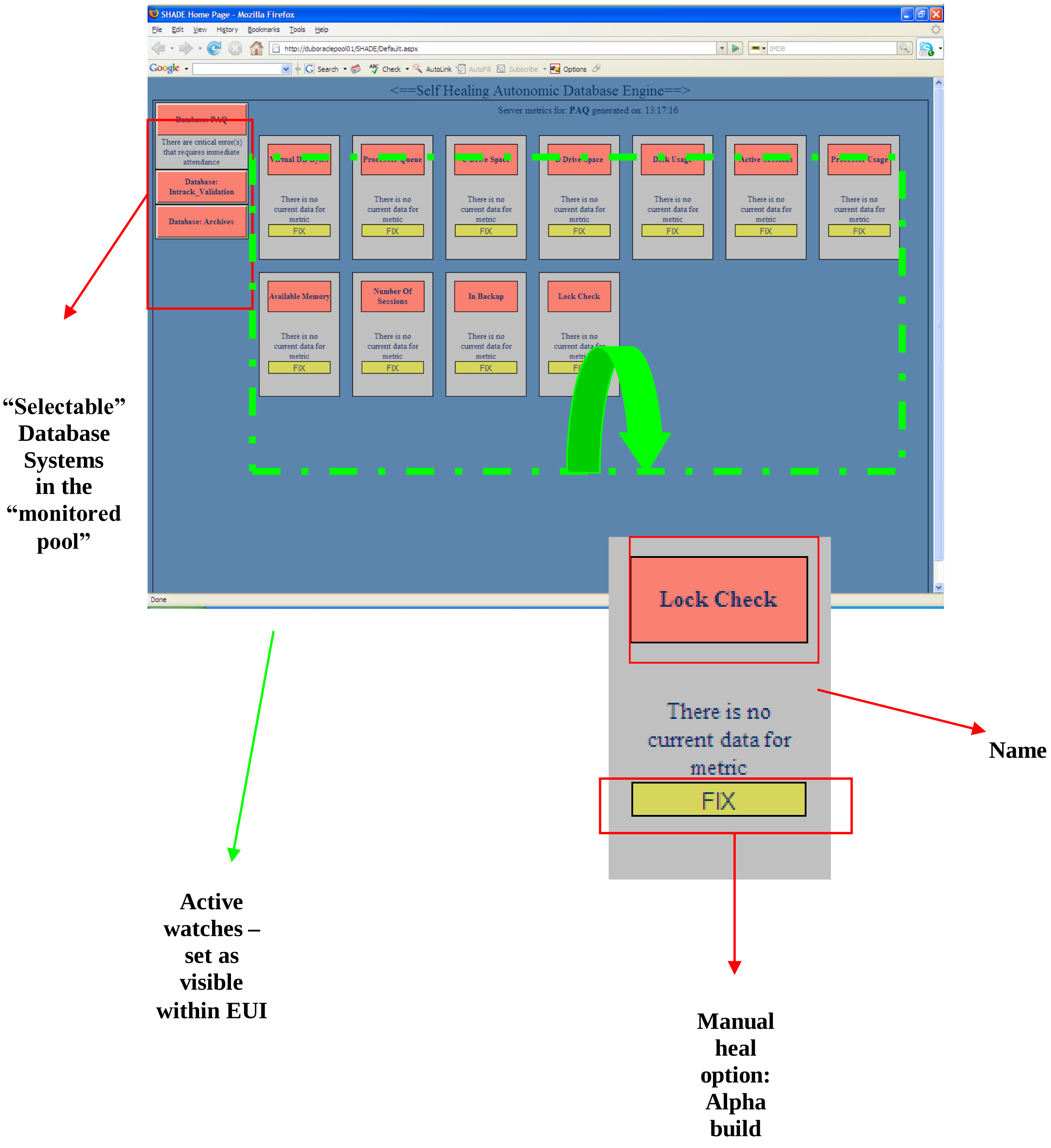
Expire old expire accounts	SOX issues	Insure account expired prior to sysdate = actually displayed as expired (logon once and fail)	
Check archive log disk for space	Prevent halt of database	< 10 % - archive off archive logs > % days to another drive (even local c).	Alert dba
Cache monitors	Contention in buffer cache/library cache and dictionary cache	If too big = shrink. If too small = increase.	Alert dba
General memory health	If free "physical memory"	Check for indirect buffers = increase if free space in physical memory. If server "paging". Decrease element of database that is most oversized and produce minimal impact. Gather info on top ten processes using server memory	Alert dba of shortage with table of 10 ten processes and what database elements changed(if any)
Database sorting	Sort area too big = large amount of physical memory that could be used elsewhere	If sorting low and area-size too high = reduce to conserve server memory	
* Continuous SQL capture	Track SQL in play	Capture running SQL with high costs and low hit rates, along with long running execution times.	Alert and grade with worst offender first. With time at top and no of actual executions.
Track top memory users (inside and outside database)	Largest memory offenders	Resize if possible	Alert dba
Track best times for doing rebuilds	Keep track of time with minimal system operations	Rebuild and shift during these times	
Top HOT files	Keep track of biggest disk i/o	Track offenders and move if possible	Flag dba
Sorts monitor	% on disk and memory	Resize as need – sort area. Disk % to high increase sort_area and sort_area_retained	
Unbalanced index check	Indexes on tables with lot of deletes	Rebuild in maintenance window.	
Check health of disks	Status on volumes	If not healthy – flag as error (or move objects?): run a backup	Alert dba
Check for unrecoverable objects	Objects set to no-logging	Alter object to logging and export object to disk	Flag DBA to do a backup
Space monitoring	Gather weekly row and table sizes - gathered after stats collected	Store in table and compare over time.	Alert DBA of top 10 and when they "should" run out of space: capacity planning histogram. Graph can show overall increase per year of data increases and growth.
Snapshot	Use oracle stats pack to gather stats on overall database at intervals	Query key elements and store in historical table	Alert dba

Fragmentation	Check Datafiles (index and tables) as well as Operating system fragmentation	Build overtime and move objects in window	Alert when fragmentation reached dangerous levels %
High-water mark	Identify space that can't be used	Move offending objects to new locations	Alert dba of free space that may not be available
Hotspot identification	Identify when system is being hit the hardest and by what	Change internal schedules if running during hotspots	Alert dba with graph of hotspots on sql/active sessions at time.
Top users	Gather stats on job users/jobs. Rows processed, disk reads, CPU usage and all system consumption	Check if multiple user sessions on single terminal – if so kill user (oldest sessions). If batch job "can it be better run at different times"	Alert DBA periodically of users and times. Who is affecting system performance and thus health at regular intervals.
Expected users	Build a table of daily users and i.p./terminals	Monitor new users. I.p. range = alert Not on domain = kill	Alert if users may be a risk

One of the key design features of S.H.A.D.E was the ability to make changes without continuously changing code. Hence, a considerable amount of the system can be altered by changing database configurations for the system. This ability saves time changing code to either add or remove features or to change and tweak parameters and settings to better suit the running of the engine.

It was decided at an early stage that the system would need a user interface to show present state and present conditions of what S.H.A.D.E is monitoring, rather than querying the system log files. The user interface also contains options to manually test and fire heals rather than waiting for the system to poll and automatically execute changes.

Fig 18.0: The user interface: S.H.A.D.E Alpha: Build 1.0 – Initial EUI



<==Self Healing Autonomic Database Engine==>

Server metrics for: **PAQ** generated on: 13:17:16

There are critical error(s) that requires immediate attendance

There is no current data for metric

“Selectable” Database Systems in the “monitored pool”

Active watches – set as visible within EUI

Name

Manual heal option: Alpha build

Various technologies that combine to make up the S.H.A.D.E engine:

- **Web services:** The S.H.A.D.E engine is managed and monitored through a standard web browser.
- **Asp.net:** Will be used to create the elements of S.H.A.D.E between the web page and the agent – the middleware.
- **Java/Javascript:** Programming language used for websites and thus useful in elements of S.H.A.D.E management and controls as well as logging of data.
- **Vb.net:** Visual Studio.net 2005. Ideal platform for the engine, as S.H.A.D.E will be monitoring within a native Windows server platform. The Windows services and "is alive" components are coded under this platform.
- **Windows service**: Windows 2000/2003 throughout. S.H.A.D.E agents will all run as standard Windows services, running under basic system accounts on the monitored database servers.
- **Oracle**: S.H.A.D.E repository stored in 10g Enterprise Database with monitored systems running under 9i Enterprise. Some database-residing code will be housed within the S.H.A.D.E repository in the form of PL-SQL and SQL functions, triggers and procedures, allowing for easy configuration changes by adding and changing database residing parameters.
- **HP DL380 (check version) servers**. Twin processor stand-alone servers with local storage, running windows 2000 within a Cisco Lan (TCP/IP). Both systems are configured with local storage.

The second build of S.H.A.D.E enhance the" Abilities" to heal and alert potential issues discovered on the test system. The engine was given the ability to retry another option, rather than repeat the heal using the same code each time.

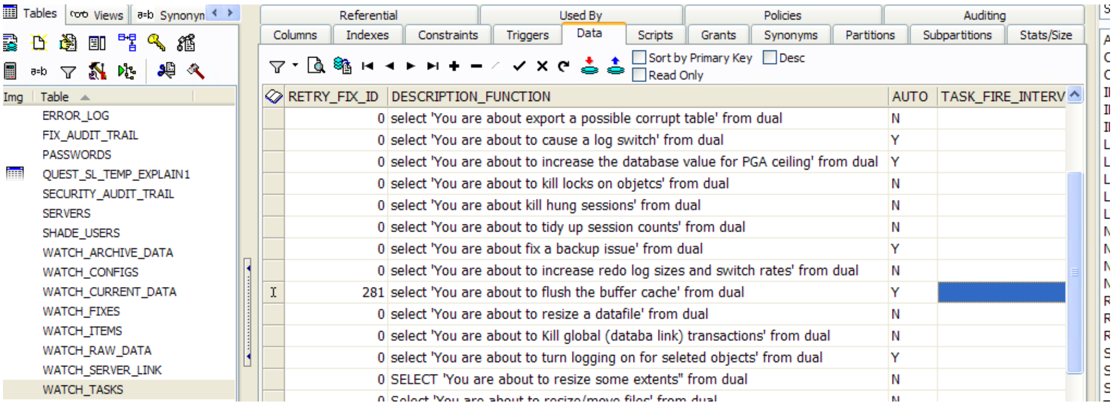
Table 5.0: List of final watches with self-healing options (final build).

ID	DESCRIPTION	Fix #1
1	C Drive Space	net and email alert to dba
2	D Drive Space	net and email alert to dba
3	L Drive Space	Sample removal of unwanted files - . dmp files only
4	M Drive Space	net and email alert to dba
5	N Drive Space	net and email alert to dba
6	O Drive Space	net and email alert to dba
7	P Drive Space	net and email alert to dba
8	Q Drive Space	sample removal of unwanted files - . dmp files only
9	Memory Leaks (Oracle)	net and email alert to dba
10	Open Cursors	set open cursor param to 500 (default + 100)
11	Processor Usage	net and email alert to dba
12	Available Memory	Net stop selection of none critical services
13	Disk Usage	net and email alert to dba
14	Invalid Objects	compile schema
15	Number Of Sys and internal Oracle sessions	net and email alert to dba
16	In Back-up	loop back-up script - end back-up
17	Active Sessions	run a full system (rda tool) dump for analysis
18	Lock Check	
19	Processor Queue	net and email alert to dba
20	Virtual DB Bytes	net and email alert to dba
21	SMART Check	net and email alert to dba
22	Statistics missing	gather stats for selected

		schema
23	Large Data files	create a second/extra file
24	Extents Failures	extend datafile
25	Logging turned off	loop alter tables to logging
26	Global Transactions	net and email alert to dba
27	Potential data file space issues	set last file to autoextend (cams main data)
28	Potential buffer cache hit rate issues	increase buffer cache 300 megs
29	Potential log switch frequency issues: too many	create three new redo log group @ 100 megs
30	Hot back-ups older than 3 days	loop and run backup
31	High level of user sessions operating	net and email alert to dba
32	Hung sessions in database	run orakill for session (risky)
33	Locks on DDL	net and email alert to dba
34	Large percentage of PGA in use	increase PGA = 50 m
35	Potential LACK OF log switch frequency issues	switch logfile
36	Daily check for corrupt rows	export batch file
37	Detect Memory fragmentation	flush shared_pool
38	Memory leaks (windows)	net and email alert to dba

Illustrated example of an automatic “Heal” in operation (below). In this example, the database’s cache hit rate falls below the defined threshold and the autonomic engine steps through an initial heal, followed by another attempt with different heal instructions (only if and when first attempt fails to resolve the issue)

Fig 19.0: Illustrated is setting the sga_target to high level 500 m (giving more cache to db) when cache hit rate is low



ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA
WATCH_CONFIGS
WATCH_CURRENT_DATA
WATCH_FIXES
WATCH_ITEMS
WATCH_RAW_DATA
WATCH_SERVER_LINK
WATCH_TASKS

RETRY_FIX_ID	DESCRIPTION_FUNCTION	AUTO	TASK_FIRE_INTERV
0	select 'You are about export a possible corrupt table' from dual	N	
0	select 'You are about to cause a log switch' from dual	Y	
0	select 'You are about to increase the database value for PGA ceiling' from dual	Y	
0	select 'You are about to kill locks on objetcs' from dual	N	
0	select 'You are about kill hung sessions' from dual	N	
0	select 'You are about to tidy up session counts' from dual	N	
0	select 'You are about fix a backup issue' from dual	Y	
0	select 'You are about to increase redo log sizes and switch rates' from dual	N	
281	select 'You are about to flush the buffer cache' from dual	Y	
0	select 'You are about to resize a datafile' from dual	N	
0	select 'You are about to Kill global (databa link) transactions' from dual	N	
0	select 'You are about to turn logging on for seleted objects' from dual	Y	
0	SELECT 'You are about to resize some extents'' from dual	N	
0	Select 'You are about to resize/move files' from dual	N	

Steps S.H.A.D.E took to “Heal” the issue:

Retry id is current task (28) with 1 for task retry 1 = 281. 281 executed if the issue is not resolved by 28. 281 is thus an additional option the engine can take if the issue still exists and first heal fails.

Increase by 100 Megs and then sleep for 60 secs (note sleep time only simulated for purpose of example).

Code executed for 281: alter system set SGA_TARGET=600M – if this heal again fails, task 282 will set the parameter to 700 Megs and then sleep for a further 60 seconds.

282 has no retry task defined (Heal will enter fail state if fix assigned to 282 fails to resolve the issue).

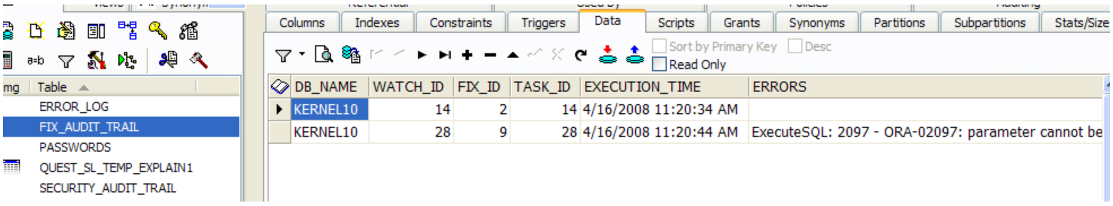
Hence, normal case even for this one heals option:

- Cache hit rate detected as lower than defined setting.
- Execute heal task 28: increases buffer cache and sleep detection.

- If rate still below defined level, execute 281: increase cache and sleep again
- If the issue is still not resolved, alter the parameter further and fail the" heal" if still not resolving the problem.

Initially, this appeared to be an issue with the base setting.

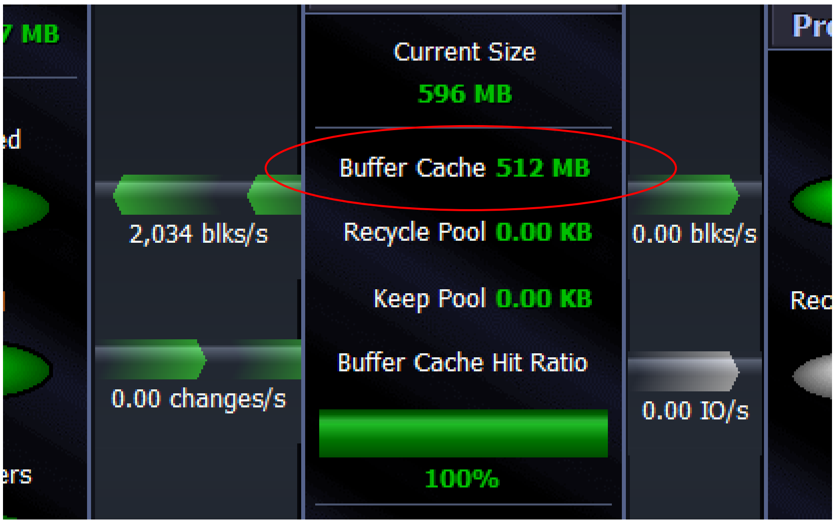
Fig 20.0: The database parameter defined in repository.



DB_NAME	WATCH_ID	FIX_ID	TASK_ID	EXECUTION_TIME	ERRORS
KERNEL10	14	2	14	4/16/2008 11:20:34 AM	
KERNEL10	28	9	28	4/16/2008 11:20:44 AM	ExecuteSQL: 2097 - ORA-02097: parameter cannot be

Database bounced – buffer cache 500 Megs – illustrated visually.

Fig 20.1: Database cache size.


Current Size 596 MB

Buffer Cache 512 MB

Recycle Pool 0.00 KB

Keep Pool 0.00 KB

Buffer Cache Hit Ratio 100%

2,034 blks/s

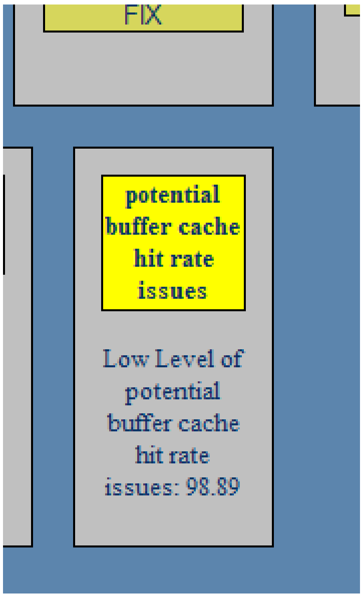
0.00 changes/s

0.00 blks/s

0.00 IO/s

Buffer hit rate high – hence now healing invoked (only medium error).

Fig 20.2: Error alerted in S.H.A.D.E interface.



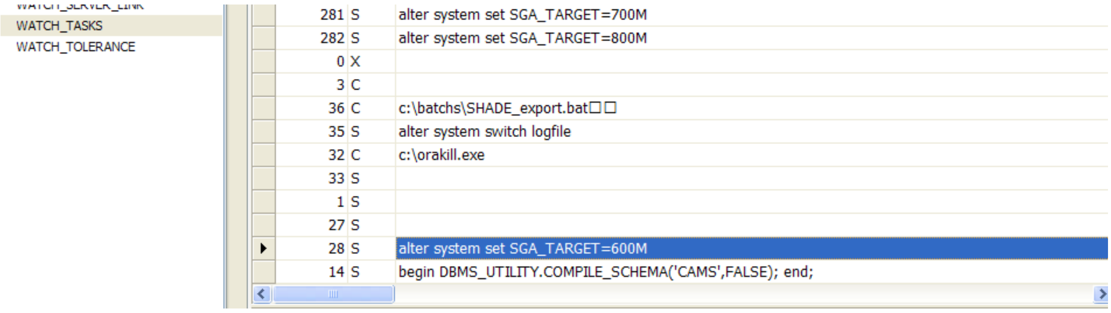
potential buffer cache hit rate issues

Low Level of potential buffer cache hit rate issues: 98.89

Simulate a heavy cache read - Kicked up to 600 Megs (healed by S.H.A.D.E + wait)

First heal executed – as per assigned parameter.

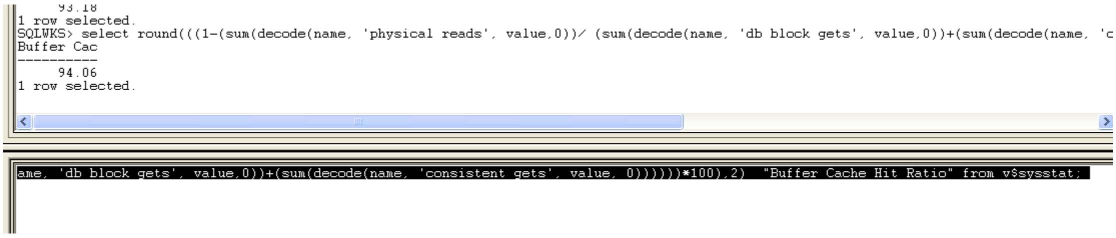
Fig 20.3: Parameter held in database repository: heal code.



281	S	alter system set SGA_TARGET=700M
282	S	alter system set SGA_TARGET=800M
0	X	
3	C	
36	C	c:\batchs\SHADE_export.bat□□
35	S	alter system switch logfile
32	C	c:\orakill.exe
33	S	
1	S	
27	S	
28	S	alter system set SGA_TARGET=600M
14	S	begin DBMS_UTILITY.COMPILE_SCHEMA('CAMS',FALSE); end;

Cache still low – 2nd heal should kick in – verify in sql

Fig 20.4: Verify cache hit rate.

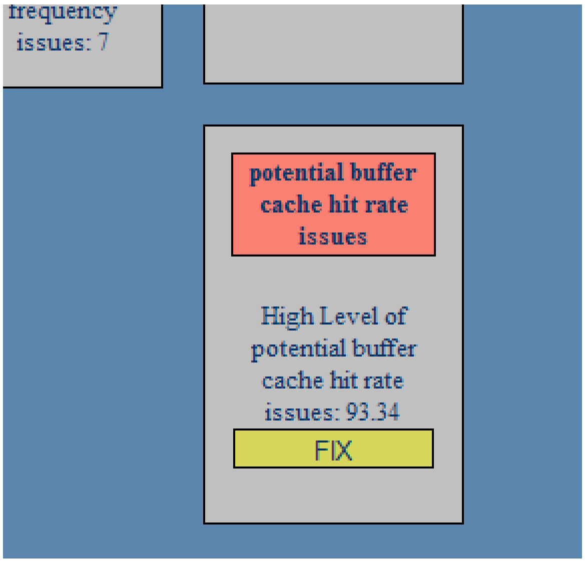


```
      93.18
1 row selected.
SQLWKS> select round(((1-(sum(decode(name, 'physical reads', value,0))/ (sum(decode(name, 'db block gets', value,0))+(sum(decode(name, 'c
Buffer Cac
----------
      94.06
1 row selected.
```

```
ame, 'db block gets', value,0))+(sum(decode(name, 'consistent gets', value, 0))))))*100),2)  "Buffer Cache Hit Ratio" from v$sysstat;
```

Still highlighted as issue in S.H.A.D.E

Fig 20.5: still alerted in S.H.A.D.E interface.


frequency issues: 7
potential buffer cache hit rate issues
High Level of potential buffer cache hit rate issues: 93.34
FIX

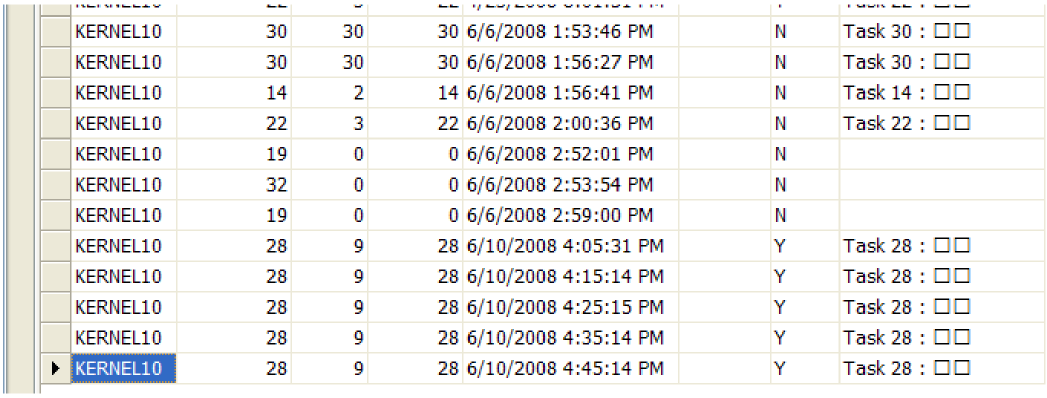
Working heal with multiple “choices” for heals – i.e. if first event fails to heal issue, will attempt a different solution x 3 times until the event is parked as faulted (prevent the use of heals that loop forever).

Event 28 – buffer cache hit rate low on database solution. Assign more memory to Oracle for SGA + buffer cache – should improve rate over time (unless poor code running in db – in which case, human will need to investigate and isolate root cause).

Event fires without issue and heals the issue – rate improves

Listed and logged in fix audit trail

Fig 20.6: Database repository logfile displays events.



	KERNEL10	22	3	22	[illegible]		Y	Task 22 : □□
	KERNEL10	30	30	30	6/6/2008 1:53:46 PM		N	Task 30 : □□
	KERNEL10	30	30	30	6/6/2008 1:56:27 PM		N	Task 30 : □□
	KERNEL10	14	2	14	6/6/2008 1:56:41 PM		N	Task 14 : □□
	KERNEL10	22	3	22	6/6/2008 2:00:36 PM		N	Task 22 : □□
	KERNEL10	19	0	0	6/6/2008 2:52:01 PM		N	
	KERNEL10	32	0	0	6/6/2008 2:53:54 PM		N	
	KERNEL10	19	0	0	6/6/2008 2:59:00 PM		N	
	KERNEL10	28	9	28	6/10/2008 4:05:31 PM		Y	Task 28 : □□
	KERNEL10	28	9	28	6/10/2008 4:15:14 PM		Y	Task 28 : □□
	KERNEL10	28	9	28	6/10/2008 4:25:15 PM		Y	Task 28 : □□
	KERNEL10	28	9	28	6/10/2008 4:35:14 PM		Y	Task 28 : □□
▶	KERNEL10	28	9	28	6/10/2008 4:45:14 PM		Y	Task 28 : □□

3.6 Refining a solution: example of space deficit

Here the watch/probe is looking for files with free space limited. This is a basic problem, but one that could cause the database to halt for numerous reasons:

Cause of halt:

1) File is close to max file size for that block size: i.e. 8k database and 32 GB [26] file.
2) There isn't enough space on the disk volume to expand the file.
3) The data file is not set to auto expand.

S.H.A.D.E must determine how many files make up the table space. The administrator may simply have set all files to auto expand, which is fine. But ideally it is better to keep them all uniform and set the last file in the set to expand – in case of emergency.

S.H.A.D.E must first determine if the system has enough disk space, before checking the file size and verifying it the file is set to auto expand; if it is and the file size is within tolerance, the engine needs only to flag a future potential issue.

[26] GB: Signifies a gigabyte of storage space.

If, however, the file will not expand, S.H.A.D.E must check size. If the size of the file is too large, it must add a new file (if disk space OK) and then communicate the change. The new file will be automatically populated when the database system needs more space. The engine must select the best cause of action as determined by the rules contained within the code. The administrators could simply set the file to expand and manually check growth over time but they may miss one factor or not be aware of the growth until the database halts due to lack of space.

IBM has shown a very strong interest in the field of Autonomic Computing, listing eight defining characteristics of an autonomic system (Reference here No. 33). This project will follow similar characteristics in the design of an agent; however, it shall be reduced to four main functional elements.

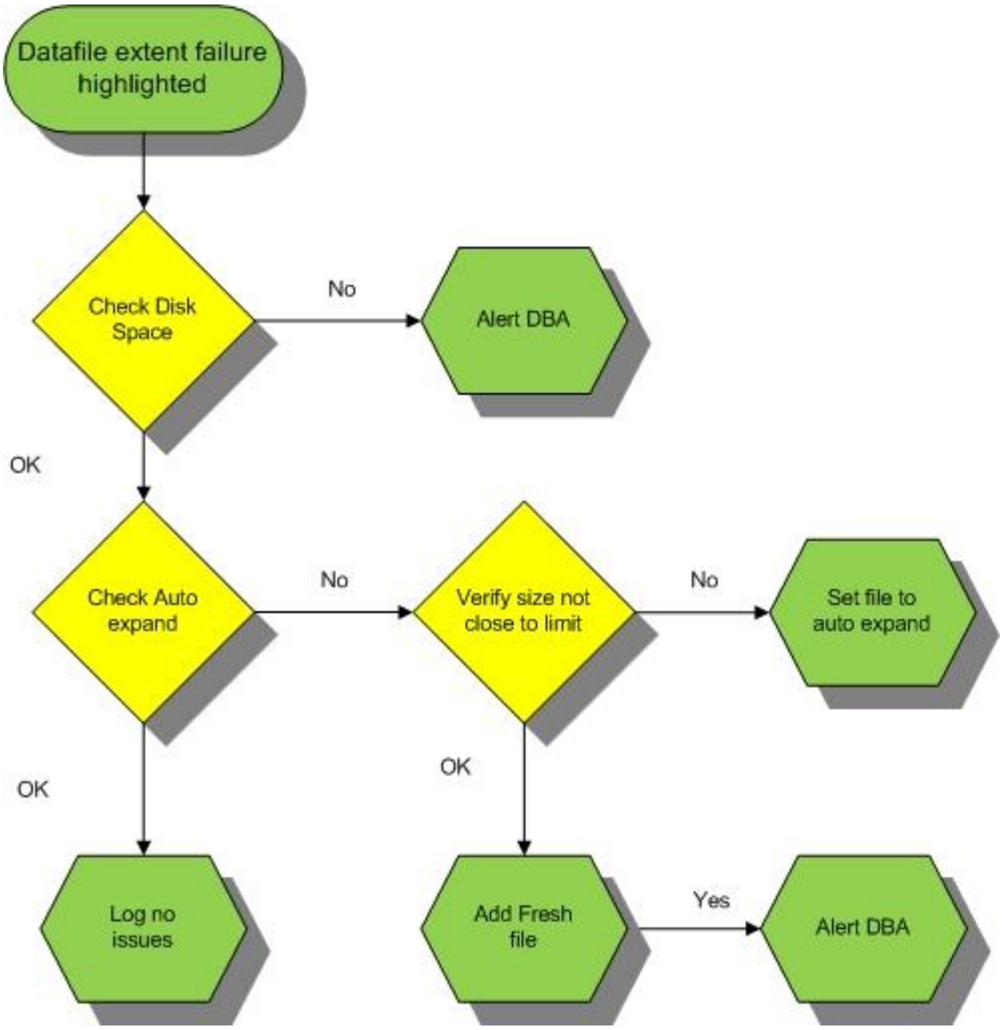
1. The agent engine needs some way of “knowing itself" and how it works; its components must also possess a system identity.

2. The agent engine must configure and reconfigure itself under different conditions and circumstances.

3. The agent engine must look for ways to optimise its workings and not just heal break conditions, but improve performance issues that affect the monitored system overall.

    1. The agent engine must be able to “heal” when alerted to events. These events will be a combination of database and operating system events. The engine will not attempt to heal hardware faults, as redundant hardware resources will be limited on the test systems.

Fig 21.0: The flowchart for this operation:



3.7 Autonomic agents and the platform of choice.

In 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry was a crisis caused by software complexity. The company cited examples of environments where software applications were constructed by some tens of millions of lines of code and require skilled IT professionals to install, configure, tune and maintain (Kephart et al., 2003). Although self-healing and autonomic computing are not new terms in the IT sector, a lot of research is still underway and remains ongoing in relation to identifying the key factors required to succeed. “Its realisation will take a concerted, long-term, worldwide effort by researchers in a diversity of fields. A necessary first step is to examine this vision: what autonomic computing systems might look like, how they might function, and what obstacles researchers will face in designing them and understanding their behaviour." (Kephart et al., 2003). The goal of autonomic computing is to create computing systems capable of managing themselves to a far greater extent than they do today (Tesauro et al., 2004). One of the key goals of autonomic computing is evident in the name and thus the system it is modeled upon: the human autonomic system. This key goal is for systems to manage the low-key but vital functions automatically. Similar to how humans regulate breathing, heart rates, blood pressure etc., these operations occur without concise thinking. These are not actively controlled by the human, but without correct operation, we would get sick and even die. Similarly, computers of the future will be better able to manage elements of their management automatically without human intervention or risk fault.

The study of how human cells "aid" each other in the process of human healing is also an important factor in the construction of self-healing computer systems. Building systems with single repair functions or processes may indeed be insufficient in tackling the "problem" of efficient self-healing. "The capacity of organisms to adapt to changing and often hostile environments, tolerate limited failures and heal damaged organs is not because of the robustness of individual cells, but because of the interactions between large numbers of cells." (George, et al., 2003).

The interaction and awareness between such cells is vital for the healing process, hence one process with self-awareness may indeed be too limiting even in computer system design. Instead, building a system that leans design to how human cells heal and aid each other may indeed yield greater success. "Cells can induce nearby cells into performing specific actions by using chemical emission or diffusion. Similarly, the death of a cell causes cessation of chemical diffusion and induces nearby cells into actions such as regenerating the dead cell. This awareness is essential for self-healing mechanisms." (George, et al., 2003). "The notion of making wireless clients snoop the environment to ensure secure and correct routing has been suggested for ad hoc networks" (Adya, et al., 2004). Using a specific application to "watch" for system faults and problems was a key element in the design of S.H.A.D.E, where the system polls specific “watches” to look for specified design errors in the databases. Much the same was discussed (Marti, et al., 2000) for Network node detection where a “watchdog” (Marti, et al., 2000) mechanism was designed with the key purpose of detecting network problems. “The watchdog identifies misbehaving nodes” (Marti, et al., 2000). “The basic idea is to have watchdog nodes observe their neighbours and determine if they are forwarding traffic as expected” (Adya, et al., 2004). “Human operators add a quality to management systems that current artificial systems can't match; humans can handle unknown situations and learn from their experiences.” (Hermann, et al., 2005). Thus, humans can adapt and react depending on levels of knowledge and experience, but can also make mistakes and misinterpret an issue.

The system must be able to adapt to change by drawing on experience and remain aware of the overall system. A suggested possibility for future autonomic system design is through the use of artificial intelligence. But present A.I. has, as yet, not achieved the ability to dynamically adapt to change or have routines/abilities to learn. Yixin Diao and colleagues have shown that a running database running specific (e-commerce) applications can be optimised so long as defined rules are followed. “If we eventually remove the human operator from the management control loop, then the management system must be able to similarly learn from its experiences and improve its capabilities.” (Hermann, et al., 2005). Oracle refers to its latest generation of database as “The “invisible” Oracle database will thus enable enterprises to become more profitable (Kumar, 2006). The term “invisible” is interesting and ties in with what IBM see as autonomic computing. The administrators need be less aware of many system operations, as the system manages most of its own operations independent of the human element. To do such, the autonomic management system needs to be aware of its environment, or at least aware enough to make changes without affecting other elements of the system. If a self-managed database system similar to 10g boasts about some of its features as being “It is the industry’s first truly self-managing database featuring an intelligent self-management framework, revolutionary self-diagnostic engine, Automatic Tuning Optimiser and automatic memory management capability”. (Wiseth, 2003), then perhaps it isn’t sufficient unless the overall self-management is carried out at the hardware and operating system level. Hardware and software manufacturers must work together, following standards to succeed 100%, otherwise elements of autonomy will be included in various computer components over time, but each will be following different guidelines and goals, unaware of the others' existence. Hence, the goal of releasing the first self-managing database engine may be admirable from an innovation standpoint but may also be a little limited in scope.

"Serious customer problems can take teams of programmers several weeks to diagnose and fix, and sometimes the problem disappears mysteriously without any satisfactory diagnosis." (Kephart et al., 2003).

Fig 22: Aspects of self-management under autonomic computing (Kephart et al., 2003).

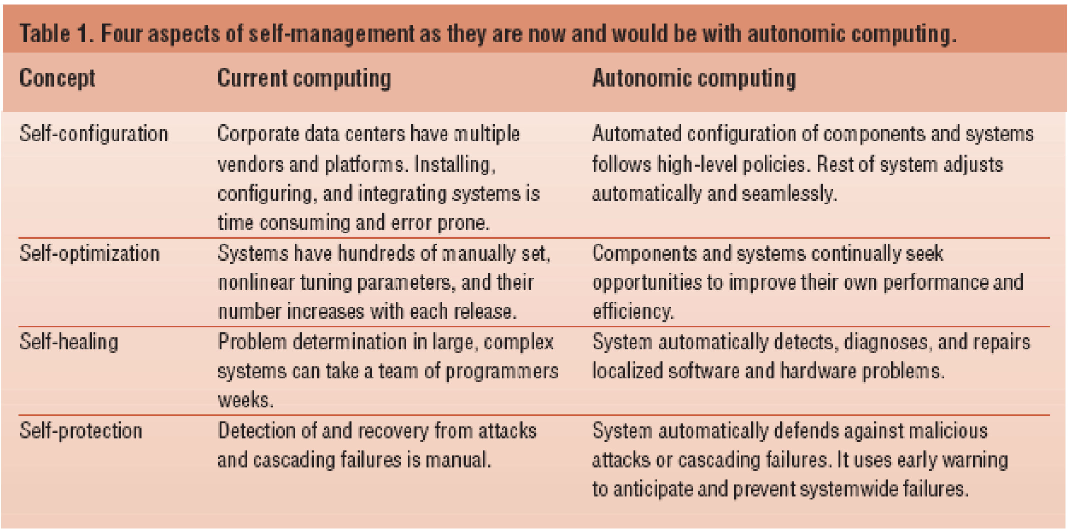


Table 1. Four aspects of self-management as they are now and would be with autonomic computing.

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

In order for autonomic computing to succeed at any level, it must be designed and be able to draw from knowledge, must be the same as the human administrator would do but would benefit from being able to react based on a pool of experts rather than the knowledge gathered by the limited number of administrators and the knowledge they have built through time and exposure. The administrator knowledge costs money to develop but also can greatly affect a corporation's ability to operate if that knowledge is lost. Building knowledge into systems can reduce this dependency as well as make the said systems more reliable. Hence, the fully autonomic system will require artificial intelligent elements in its design as well as expert systems. The system, therefore, risks having the information but does not know what to do or how to do it, much the same as an inexperienced human may have knowledge but might never have used it. Experience is an important element, but how is experience built into a system?

The chosen Windows-based application was selected because of certain key factors:

1) The chosen laboratory within the Bausch & Lomb plant is Windows-based for servers and clients (Corporate standard as of May 2007).

2) The programming application of VB.net is again a corporate standard but also allows me to use a licensed product without incurring further expense.

3) Linux [27] was looked at during the concept stage of the application but limited to "in operation" Linux database servers as well as client configurations would have reduced the quality of the study. The S.H.A.D.E application could only have been measured against a system with faults induced through use of scripts and applications, rather than actual users. The Linux operating system would also require access to a Linux-specific version of Oracle and programming language (JAVA) which would have introduced further delays in obtaining, building as well as learning to interface with the Linux kernel for operating system-specific events. The key element of S.H.A.D.E and thus the goal to work towards reducing system complexity, was the Windows interface (because of my skill set) which complements this goal (as a novice can use and get it operational with minimum knowledge). From limited experiments with Linux during the concept stage of S.H.A.D.E, Linux with numerous "Distros "(Linux distribution version), kernel builds and driver support didn't present itself as a system a novice could work with. The Windows operating system seems more in line with the goal of "being less concerned with how computers work and more concerned with using them".

[27] Linux: An open source operating system based on UNIX.

4) In the initial stages of system design, numerous elements of the overall computer system - hardware, operating system, database, other software and network - were analysed and considered as potential healing components of the S.H.A.D.E system. Initially, planning to inject healing for different components possible within software detection and prevention would require expert knowledge on each and every component of a system, as well as funding and time to carry out research into the hardware elements. This also may have made it difficult to accurately measure and analyse the beneficial effects of a self-healing system (as a whole). “By being specific, it becomes specific and measurable -- that makes it science and not philosophy” - General Manager, Microsoft Research

The database component of the data system (with some operating system healing) was selected for:

a) The database system is commonly the main software used to manage mission critical data.
b) The database is constantly evolving and changing and thus must adapt to change – either manually or automatically. This alone allows for a greater possible selection of fault detection and prevention.
c) Although future versions (10 + 11 g of Oracle) boast simplified management interfaces and elements of self-management. A large number of operational databases used are early 8i and 9i which could benefit from simplified and more cost-efficient management.

An IBM objective for its Autonomic Computing project is to free people from having to think how computers continue to operate. “The best measure of our success will be when our customers think about the functioning of computing systems about as often as they think about the beating of their hearts.” (IBM, 2001).

Usage of the Bausch & Lomb (test) systems and thus corporate platform allows the laboratory systems to be in use 24x7x365 days a year, with real users in operation. Oracle had been using a marketing strategy of Oracle on Linux as "unbreakable". They even started to release the Windows version of newer databases later than the Linux build, suggesting in some circles that the Windows platform may be dropped in the future. This worked in persuading some IT Managers that the combination was the true, serious platform, with Windows being a system that was both cheap and unreliable. Experience of using Oracle through versions 7, 8i, 9i and 10g on Windows 3.1, NT, 2000 and 2003(servers), over 10 years of personal experience. The Windows environment has only failed a total of approximately 5 times, through a combination of blue screens. Hardware has failed more often, with the Windows platform becoming more reliable with newer versions. One Windows 2000 cluster running a 350 gb Oracle8i database only failed (from a windows point of view) once during a year of 24x7x365 operation, with a simple reboot clearing the issues. The systems were simply up too long. Even Oracle is now taking the Windows/Oracle platform more seriously. Plugging Oracle on Windows is a reliable cost-effective solution, but this is more a customer-controlling Oracle, as the Windows platform is heavily used as an Oracle Server solution.

Windows has managed to get it labelled as a type of "poor" cousin for server usage, although a lot of medium to large firms are starting to use it because of cheaper and reliable hardware support coupled with reduced ease of management. . "The balance of performance and usability is what attracted emsCharts to an Oracle/Microsoft solution" (Baum, 2007).

Oracle even "emulated" the windows style in their installation for Oracle 10g, making the system easier to get installed and running, after much criticism with the 9i version being too slow and complex. Ease and speed of installation became one of their marketing ploys which includes "Including easy-to-use installation wizards that allow a person with little or no database expertise to successfully install a well-configured instance of the RDBMS" (Olofson, 2005), again showing that complexity (at any level) is causing issues.

"Oracle has dramatically reduced the number of installation steps since version 9*i*, and Oracle Database 10*g* may be fairly described as one of the easiest full-function RDBMS products to install. According to Oracle, SE1 typically installs in a matter of minutes, requiring only a couple of user-defined input parameters. The product comes with many sensible installation values pre-set (users can always over-ride these defaults if desired) that formerly had to be set manually. In addition to easy-to-use wizards, the process includes software that analyses the target system and automatically generates database settings, greatly reducing both the effort and expertise required of the person doing the install." (Olofson, 2005).

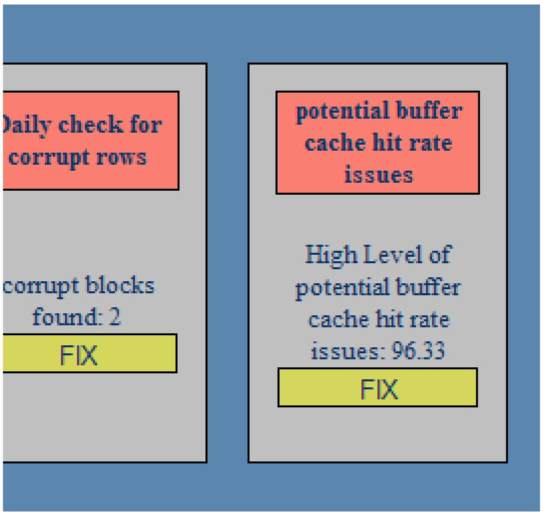
"We've worked hard to ensure that Oracle Database 10*g* is easy to install, manage, and develop on Windows" - Oracle's senior director of Windows development (Baum, 2007). Hence the platform for the S.H.A.D.E (in this Thesis) is Windows/.Net/Oracle. "In conjunction with our Microsoft software assets, Oracle database technology gives us a real competitive advantage." (Baum, 2007). "The ability of a system to heal itself without requiring administrator assistance greatly simplifies management." (Ling, 2004).

3.8 The final build

The final build of S.H.A.D.E enhanced the end-user interface for the administrator to interface with. Another key feature here was the ability to break watches to prevent continuous execution, even when the task fails to heal.

Illustrated: Same watch executing after three times, it is failed and "parked"

Issue flagged by S.H.A.D.E



Daily check for corrupt rows

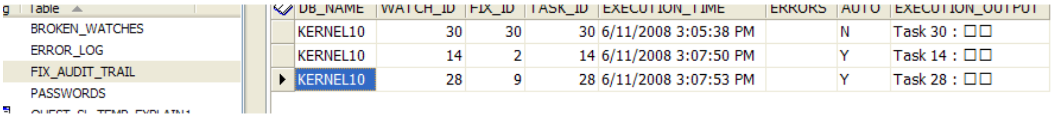
corrupt blocks found: 2

potential buffer cache hit rate issues

High Level of potential buffer cache hit rate issues: 96.33

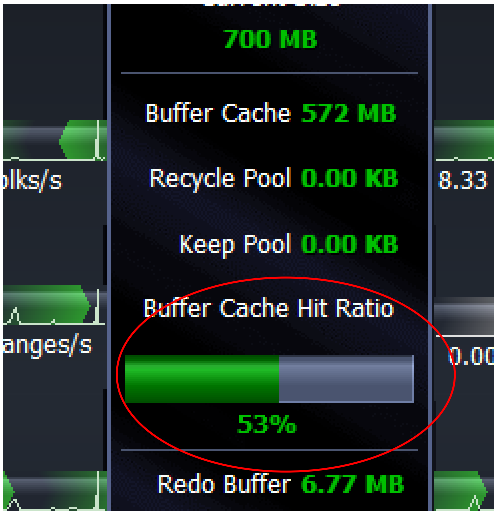
Wait for heal (10 minutes)

First heal



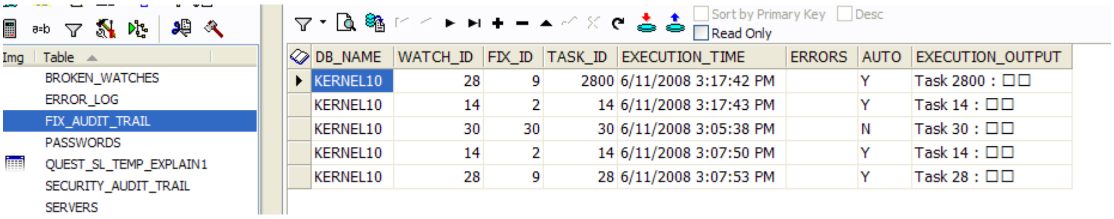
DB_NAME	WATCH_ID	FIX_ID	TASK_ID	EXECUTION_TIME	ERRORS	AUTO	EXECUTION_OUTPUT
KERNEL10	30	30	30	6/11/2008 3:05:38 PM		N	Task 30 : □□
KERNEL10	14	2	14	6/11/2008 3:07:50 PM		Y	Task 14 : □□
KERNEL10	28	9	28	6/11/2008 3:07:53 PM		Y	Task 28 : □□

Issue continues – low buffer cache hit rate = 53%

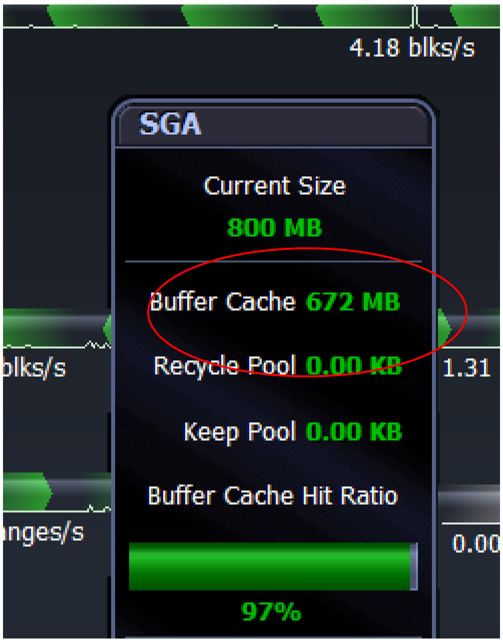


- **10 minutes later- heals with 2nd heal (as first obviously failed to fix 100%)**

- **2nd attempt – increasing SGA to 800 megs**



DB_NAME	WATCH_ID	FIX_ID	TASK_ID	EXECUTION_TIME	ERRORS	AUTO	EXECUTION_OUTPUT
KERNEL10	28	9	2800	6/11/2008 3:17:42 PM		Y	Task 2800 : □□
KERNEL10	14	2	14	6/11/2008 3:17:43 PM		Y	Task 14 : □□
KERNEL10	30	30	30	6/11/2008 3:05:38 PM		N	Task 30 : □□
KERNEL10	14	2	14	6/11/2008 3:07:50 PM		Y	Task 14 : □□
KERNEL10	28	9	28	6/11/2008 3:07:53 PM		Y	Task 28 : □□



SGA

Current Size
800 MB

Buffer Cache 672 MB

Recycle Pool 0.00 KB

Keep Pool 0.00 KB

Buffer Cache Hit Ratio

97%

Wait for third and final fix – if needed.

Third fix

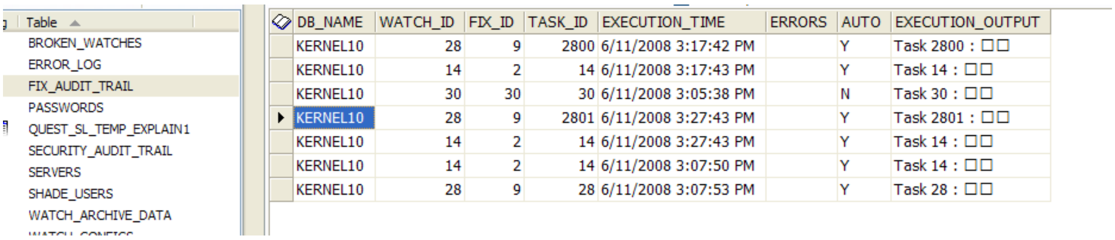
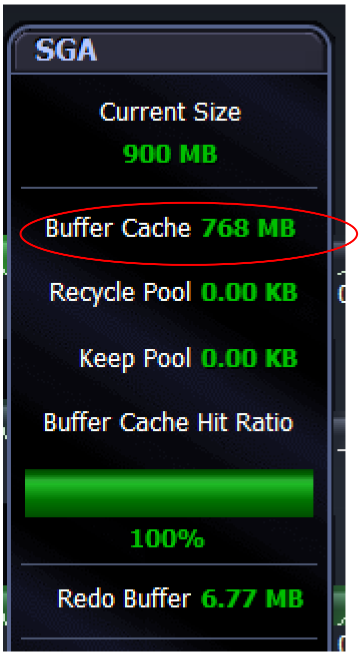


Table
BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA

DB_NAME	WATCH_ID	FIX_ID	TASK_ID	EXECUTION_TIME	ERRORS	AUTO	EXECUTION_OUTPUT
KERNEL10	28	9	2800	6/11/2008 3:17:42 PM		Y	Task 2800 : □□
KERNEL10	14	2	14	6/11/2008 3:17:43 PM		Y	Task 14 : □□
KERNEL10	30	30	30	6/11/2008 3:05:38 PM		N	Task 30 : □□
KERNEL10	28	9	2801	6/11/2008 3:27:43 PM		Y	Task 2801 : □□
KERNEL10	14	2	14	6/11/2008 3:27:43 PM		Y	Task 14 : □□
KERNEL10	14	2	14	6/11/2008 3:07:50 PM		Y	Task 14 : □□
KERNEL10	28	9	28	6/11/2008 3:07:53 PM		Y	Task 28 : □□

Verify 900 Megs on db (SGA) – 768 for cache



SGA

Current Size
900 MB

Buffer Cache 768 MB

Recycle Pool 0.00 KB

Keep Pool 0.00 KB

Buffer Cache Hit Ratio

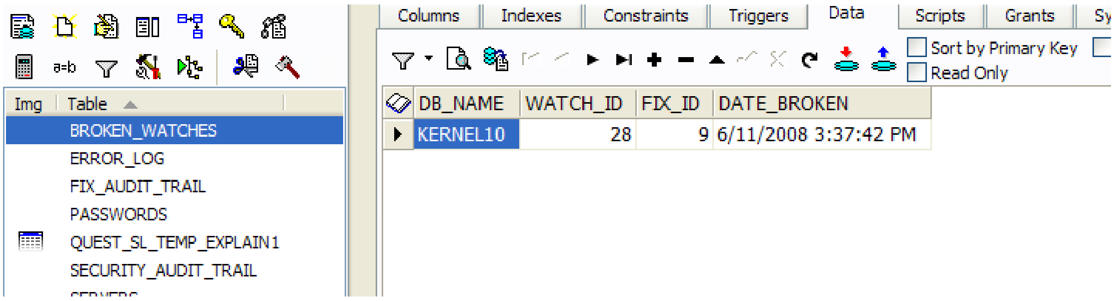
100%

Redo Buffer 6.77 MB

S.H.A.D.E waits 10 minutes (timer) and will fault the future healing to prevent continuous healing -

3:37 – task 28 – increase buffer cache if detected now in broken jobs – will no longer fire until administrator fixes the issue.

Watch breaks

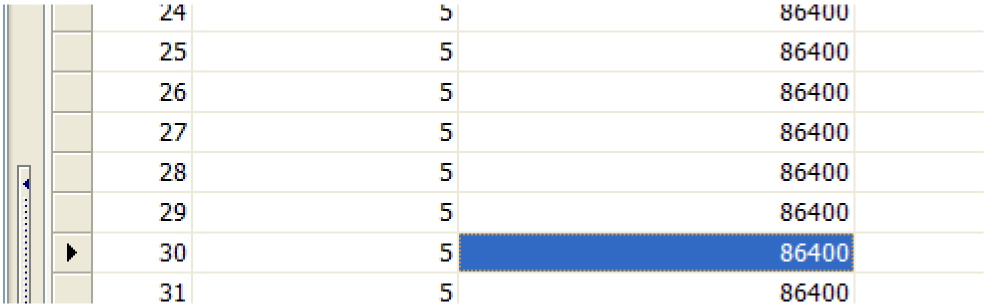


BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL

DB_NAME	WATCH_ID	FIX_ID	DATE_BROKEN
KERNEL10	28	9	6/11/2008 3:37:42 PM

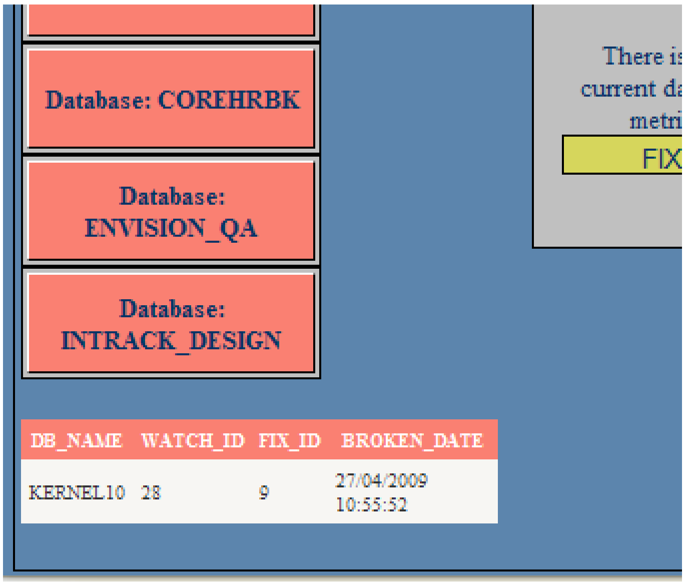
Shown in user interface as broken: will need administrator to intervene and remove otherwise S.H.A.D.E will not fire heal again – but will alert

In this case, the job was set to fail if it failed 5 times in 86400 seconds:



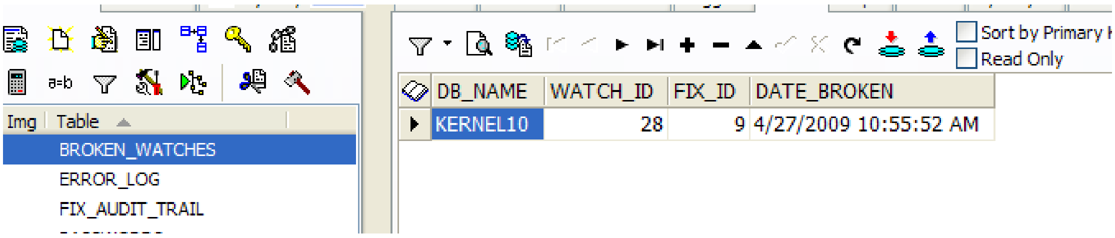
24	5	86400
25	5	86400
26	5	86400
27	5	86400
28	5	86400
29	5	86400
30	5	86400
31	5	86400

Show as broken in interface



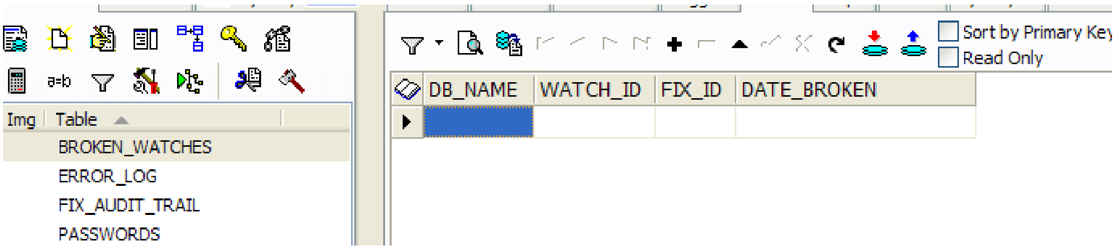
DB_NAME	WATCH_ID	FIX_ID	BROKEN_DATE
KERNEL10	28	9	27/04/2009 10:55:52

And in database repository log



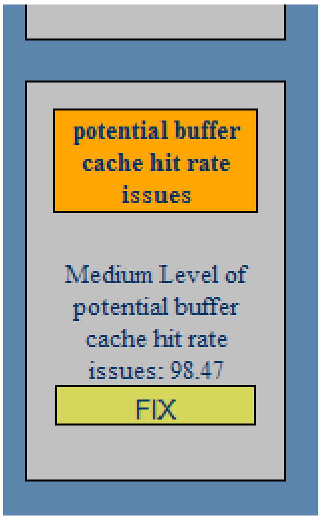
DB_NAME	WATCH_ID	FIX_ID	DATE_BROKEN
KERNEL10	28	9	4/27/2009 10:55:52 AM

Removing it from log allows the heal to operate again



DB_NAME	WATCH_ID	FIX_ID	DATE_BROKEN

Removing the job from the watches table allow for job to run again.



potential buffer cache hit rate issues
Medium Level of potential buffer cache hit rate issues: 98.47
FIX

This means the heal/watch will require manual intervention by the S.H.A.D.E administrator, otherwise the buffer cache check and thus heal will remain in a parked/broken state.

3.9 Summary

The S.H.A.D.E engine could have benefited with more development time, allowing for more of the initially defined options to be developed and introduced and thus making for a more complete system for use against any live environment. But the goal of this thesis was not to produce a commercial product but to measure how effective a system could be at being more self-sufficient by reducing human interaction by automatically fixing its own issues. Too many options would have also resulted in more time required for tweaking application code and analysing results produced by fails and fixes. A percentage of watches was introduced but configured to alert only. These fixes will be redeveloped in later builds of the engine, introducing more features and heals as the S.H.A.D.E engine matures with further development in future builds.

Chapter 4: Results and Evaluation

4.1 Introduction

The initial problem was in defining what a fault is, as well as defining what makes a system healthy. Even when a healthy system level is clearly defined, maintaining this level of health can be an even more complex problem (Ryan, et al., 2008). The easiest way is to define a set of parameters the system must meet; failing to meet or measure up to these parameters flags the system as being in a less than optimal or unhealthy state. But to define such parameters, one must be familiar with each and every component that makes up the system as a whole. It is also vital to know each and every corresponding component that can affect the health of a system, as well as what affects each of these components. Every element that maintains the health must be accurately defined, along with their dependent and dependencies. However, “the distinction between "healthy" and "broken" is often indistinct” (Shaw, 2002).

“Maintaining system health requires knowing what "health" is and recognising when the system needs to be healed. The first problem is establishing the criterion for health, which depends on the way the user is depending on the system” (Shaw, 2002). The average user may or may not need to know what makes the system tick or function at any level, but they do know what they want from a system. The administrator’s definition of a fault may be simply defined by a failed component; if a system remains up, it is operational. To the end-user, if response time is slow, it is in a fault state (Ryan, et al., 2008). The system is not operating to its specification, no matter how unrealistic that expectation is (Jupiter Research, 2006). This is where the clear definition of a fault becomes unclear, as it differs from person to system. What one operator accepts may not be sufficient for another. “Approximately 75 percent of online shoppers who experience a site that freezes or crashes, is too slow to render, or involves a convoluted check-out process would no longer buy from that site” (Jupiter Research, 2006). This is why performance-monitoring elements have been included in the design of S.H.A.D.E. and not just failed elements. However, this in itself is not as straightforward as it appears. “Distinguishing between a faulty system and a system that is in a sub-optimal state is not always easy” (Hermannet al., 2005).

The results from the experiment presented in this chapter follow the structure of the DMAIC framework.

4.2 The research question

This research was chiefly concerned in the investigation into the factors that allow a computer system to self-heal and thus be more self-sufficient. This investigation required not just research into current and past trends within this field but also the design and implantation of a custom-built engine that could be analysed over a sufficient time period. This analysis is presented within this chapter, based on the findings produced by the engine’s log files.

4.3 Downtime and performance loss analysis

Each problem alerted in both systems can be individually measured, based on number of occurrences over a defined time period of sampling. Every event alerted as a fault (level 3) has already been defined as possible events that can affect either the actual system “up-time” or overall performance. Each event treated as a fault affects the system’s health. Whether the event effects performance or operational ability, it is still classed as an outage as it affects health.

4.4 System comparison of variance

The potential success cannot just be analysed from the results of how the S.H.A.D.E engine performed on the test system used during the experiment, but also to compare the variance of this success with relation to a system that was monitored within a similar environment. A two-way ANOVA was thus chosen to analyse the overall system performance(s), factoring in permutations such as reboots and system “up-time”/downtime during events that were outside the control of the experiment. One such case is a system patch and reboot. The

S.H.A.D.E engine can detect, but the event is outside the scope of the experiment and outside the control of the experiment to prevent.

The data from the second “non-healing” system was not accurate enough with the use of a fault-injecting engine, as both systems experience completely different issues during the analysis phase. Hence, the ANOVA analysis is not represented in this thesis as it would be presenting completely different issues on two different systems. One set of results revealed how effective the engine was at healing issues, while the other represented how many issues another system experienced during the same time period. Further analysis and further development of the engine has been identified as requiring a fault-injecting engine to simulate system faults and errors.

Results from both systems are represented in all findings contained within section 4.6.1 of this chapter.

4.5 Measure phase

The results of the measure phase are presented in section 4.6.1 of this chapter, with comparison data tables and graphs.

4.6 Analysis phase

All the archived data from the engine was queried over a defined period and cross-compared to highlight where the system was successful in healing and where it failed. The analysis also clearly defines common faults that were detected in the comparison system, but not when healing intervention was introduced. Because no faults were injected to simulate failures, the data presented in these sections has been obtained from real systems in real-world operations.

4.6.1 System comparisons: self-healing vs. stand-alone system.

More accurate comparisons could only be made in relation to overall system faults between the two systems, as both systems didn’t experience the same amount of faults and problems. The only way to accurately compare how both would have handled the exact same faults and errors, would have required the design and execution of a fault injection system, that could and would simulate exactly the same type of faults on both systems and compare how they reacted (or not), depending on their healing abilities.

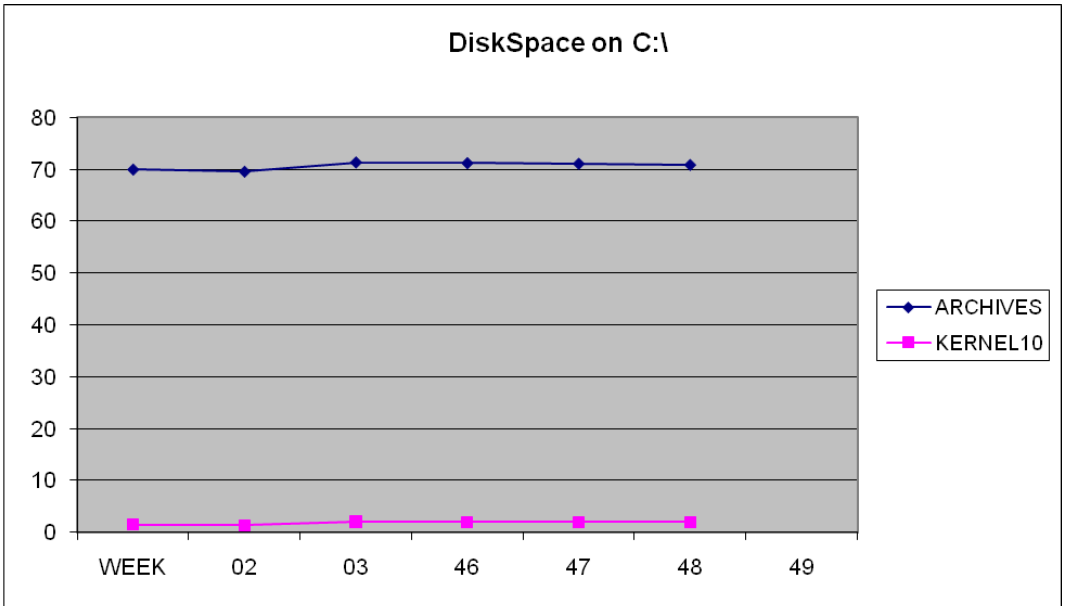
Each watch and thus fault is analysed and compared separately, with individual graphs for S.H.A.D.E-enabled system compared with systems without S.H.A.D.E-enabled engine running (only monitored, not healed).

Each of the graphs and results presented here were taken from a defined timeline from both systems. Each graph presents an element that is monitored and healed (where possible) on the database being monitored.

<u>Watch #1: Disk space free on c:\ volume of server.</u>

The disk space threshold is a very basic query to verify the amount of disk space free on the database server C:\ (Local disk) volume.

Graph 1.0: Watch #1: Free disk space on c:\ volume comparison between systems.

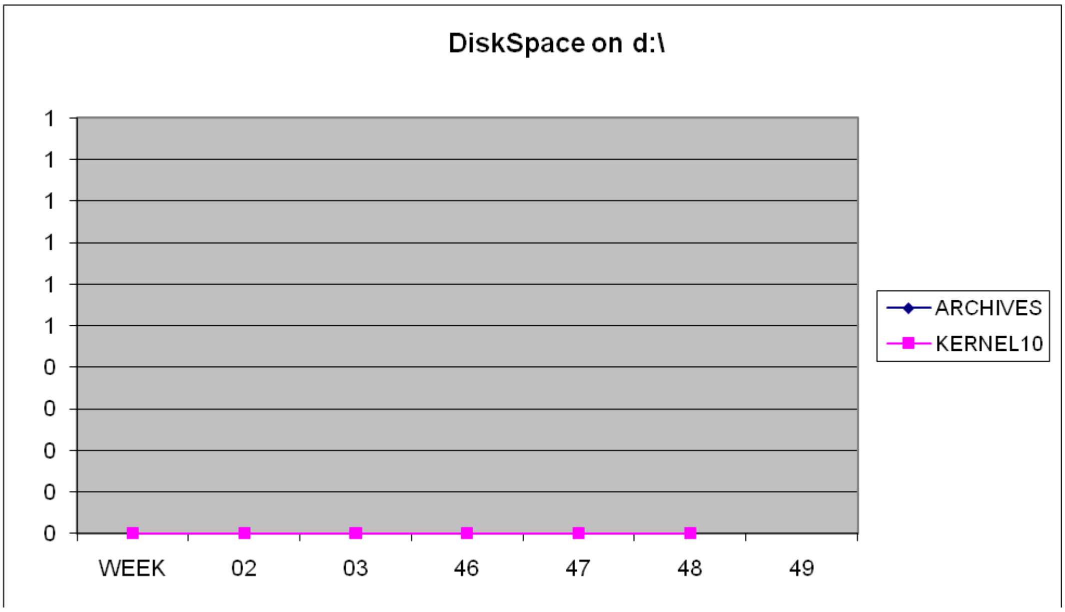


As seen from the graph above, the S.H.A.D.E-enabled engine flagged an error and attempted a repair but in this case was unable to repair the issue under its defined parameters. The defined parameters were the ability to delete certain “safe” file types only, not dynamically adjust disk volumes to make more space available. Hence in this example, S.H.A.D.E was only successful in alerting the administrator to the problem with an email alert (and breaking of the watch = alerted but no longer attempted a heal as the defined attempts ensured the heal was “parked” after a defined amount of attempted repairs). To be truly successful with this type of alerted fault, the engine would need an advanced ability to move disk space or re-size disk partitions to increase disk space (without impacting system “up-time”). It was deemed too high a risk during the design and testing of S.H.A.D.E and the safer option was to remove defined files, if any, and alert the administrator that they “may” have a disk space issue in the future. The monitored-only system also experienced the problem but to a lesser fault level, in that it has more free space on the C:\ volume, which also was flagged.

Watch #2: Disk space free on d:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server D:\ (Local disk) volume.

Graph 2.0: Watch #2: Free disk space on d:\ volume comparison between systems.



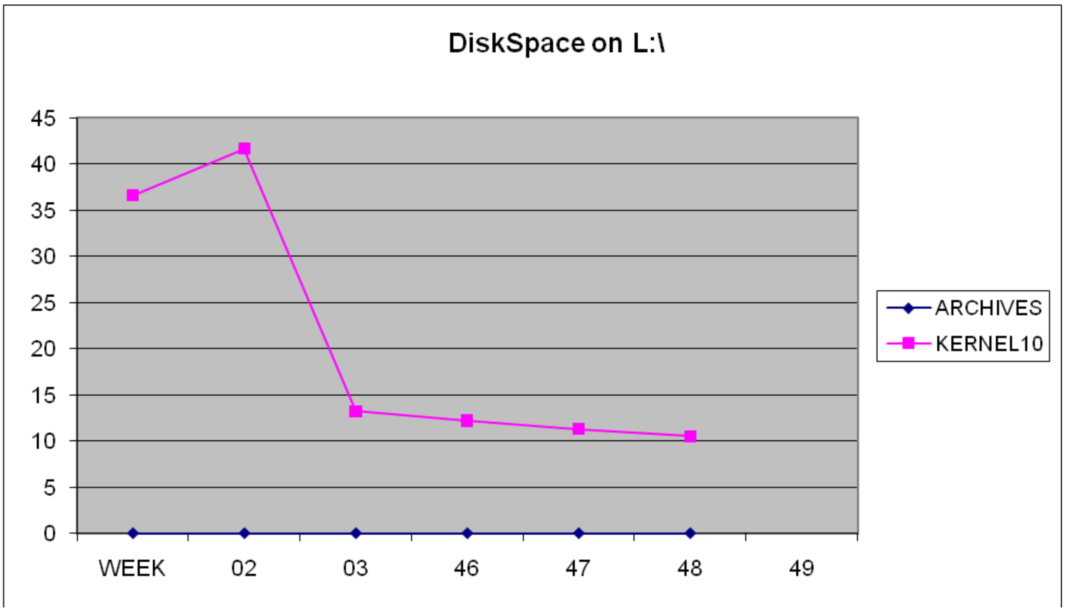
In this case, the D:\ volume on the S.H.A.D.E-enabled system had no data as the D:\ volume was a compact disk/optical drive and no space was determined from the query. The monitored-only system reported zero issues as there was more than enough free space (80 gigs [28]) and thus the alert was not fired as this was much more than the defined parameter.

[28] 28 Gigs: Common term used by IT professional's to represent the space on a disk or area in memory. 1 *gigabyte* is 1000000000 bytes, or a thousand megabytes.

Watch #3: Disk space free on L:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server L:\ (Local disk) volume.

Graph 3.0: Watch #3: Free disk space on L:\ volume comparison between systems.

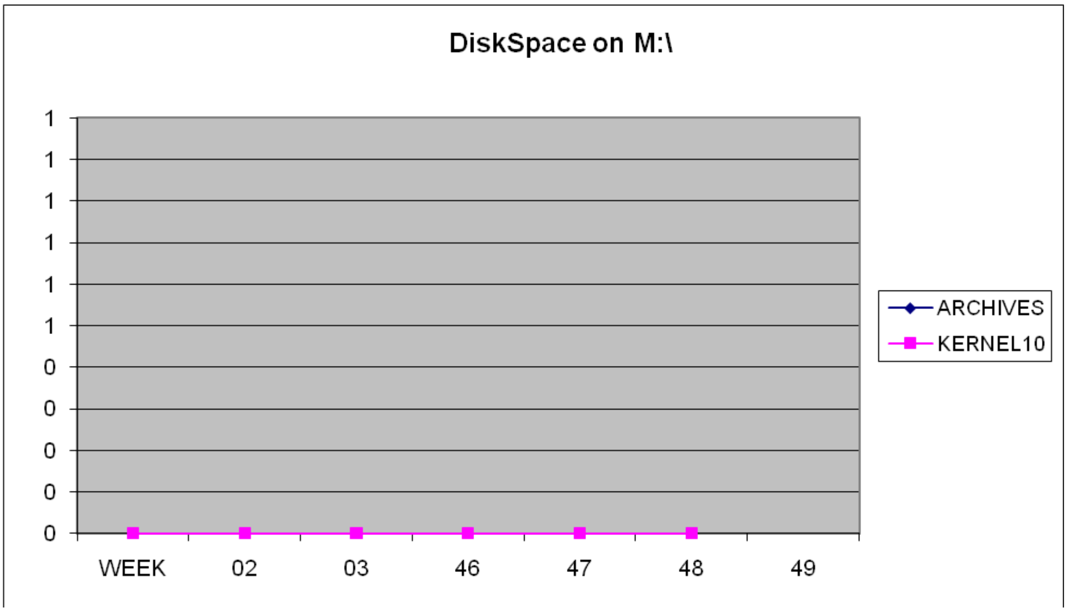


The S.H.A.D.E-enabled system managed to reduced the amount of error alerts by acting from its defined action list and removing files that it was given the ability to delete. These “safe” files allowed the system to operate as normal without the need to move Oracle data from this volume or without the need to re-size or add further disk space - which may have required system downtime to conduct and without human supervision could result in a system halt if they failed, hence the heal in question was structured with parameters that would operate with minimal risk to overall health. If the S.H.A.D.E system was given the ability to move defined Oracle datafiles, it would need more awareness of which other volumes could handle the change along with the knowledge and possible ability to alter back-up scripts or any other system elements that may be affected by these types of changes. The monitored-only system had no data as it was not configured with a L:\ volume.

Watch #4: Disk space free on M:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server L:\ (Local disk) volume.

Graph 4.0: Watch #4: Free disk space on M:\ volume comparison between systems.

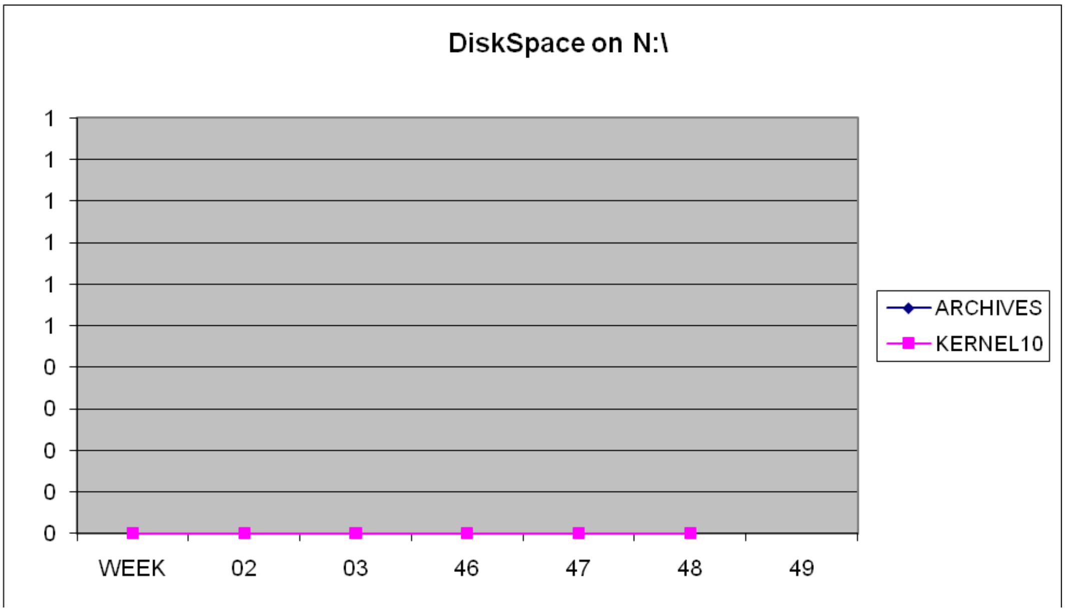


This watch was included to monitor other systems with said volume during the initial design and beta testing of the S.H.A.D.E engine monitoring capabilities and functions. Neither of the selected systems have an M:\ volume, so no data was collected. This result and graph were included to represent the accuracy of the data, by presenting an example of all defined “watches” or monitoring abilities in the system. If not applicable, no data will be collected, archived and stored.

Watch #5: Disk space free on N:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server N:\ (Local disk) volume.

Graph 5.0: Watch #5: Free disk space on N:\ volume comparison between systems.

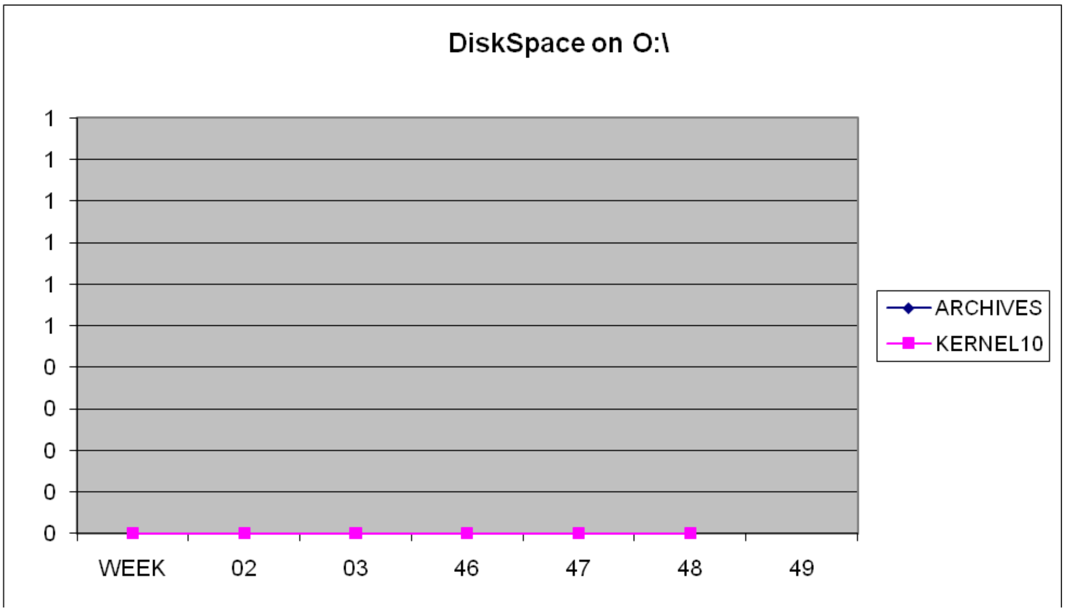


Neither system had an N:\ volume for storage, hence no data was collected or issues alerted on. This graph and watch is only presented here as an example of data collection.

Watch #6: Disk space free on O:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server O:\ (Local disk) volume.

Graph 6.0: Watch #6: Free disk space on O:\ volume comparison between systems.

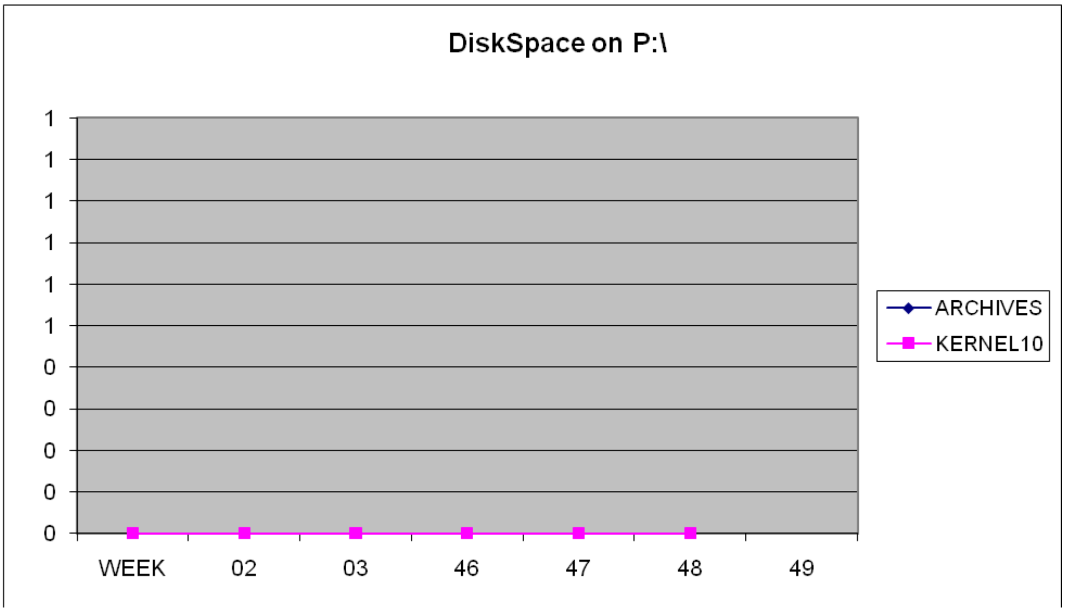


Neither system had an O:\ volume for storage, hence no data was collected or issues alerted on. This graph and watch is only presented here as an example of data collection.

Watch #7: Disk space free on P:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server P:\ (Local disk) volume.

Graph 7.0: Watch #7: Free disk space on P:\ volume comparison between systems.

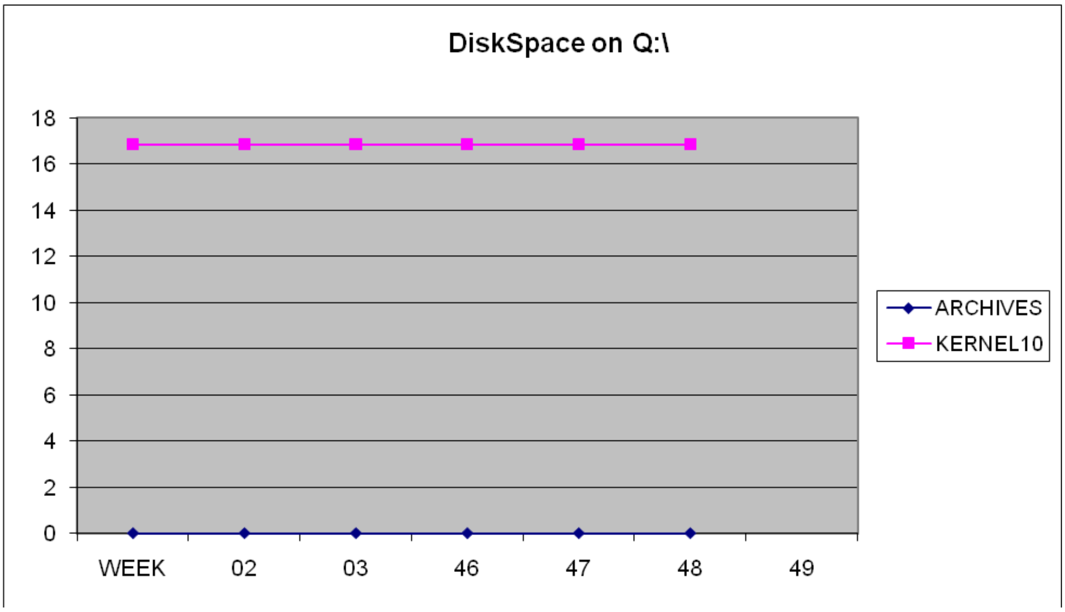


Again, neither system had a P:\ volume for storage, hence no data was collected or issues alerted on. This graph and watch is only presented here as an example of data collection.

Watch #8: Disk space free on Q:\ volume of server (if volume exists)

The disk space threshold is a very basic query to verify the amount of disk space free on the database server Q:\ (Local disk) volume.

Graph 8.0: Watch #8: Free disk space on Q:\ volume comparison between systems.

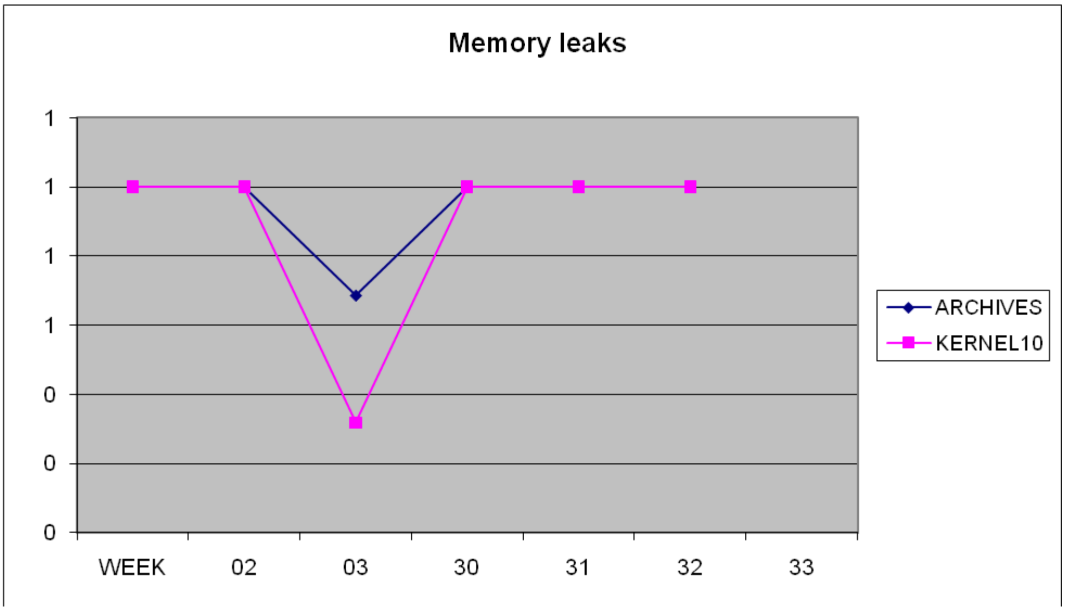


The S.H.A.D.E-enabled system detected the error but was unable to alter the condition of the error or adjust the free disk space using its pre-defined rules. Hence, the “heal” was halted and parked, only alerting the administrator of the problem (and the broken heal). The “heal” remained as such after this, as no files could be removed from the disk. It required a volume increase or movement/reconfiguration of an application to allow the adjustment of disk usage to a level that S.H.A.D.E would accept as “safe”. The monitored-only system didn’t have a Q:\ volume and hence reported no issues as no data was collected. In this case, S.H.A.D.E required more abilities and knowledge of the system to allow it to heal successfully, without failure and thus parking of the healing process for watch #8. It exhausted all its options and then alerted the administrator, in much the same way as a technician operating without enough knowledge to pursue the problem further would call for help.

Watch #9: Number of memory leaks in the database.

This query collects data from a database data dictionary[29] view which collects data regarding “potential” memory leaks with the RDBMS itself while operational.

Graph 9.0: Watch #9: Number of detected memory leaks in both systems.



The data presented and collected caused some concern with initial analysis as both systems seemed to heal at the same time. Further analysis of the events that occurred on both systems outside the scope and control of S.H.A.D.E showed that both database servers were batched and subsequently re-booted. This re-boot flushed the data dictionary of the database and the underlining statistics used to monitor memory leaks from refreshed. Hence, the database appeared to have been repaired, but from the graph, the problem arises again in much the same manner on both systems (as both systems are running similar applications and the same batch set of the database). S.H.A.D.E was only given the ability to flush Oracle’s memory when the issues arise. This can, at best, only reduce the occurrences of the memory leaks, but won’t eradicate the problem. Also, first detection will remain and the issue remains alerted. The administrator is alerted about both systems and can best react as needed. In this event, the only way to completely

[29] A metadata repository or centralised repository of information about data such as meaning, relationships to other data, origin, usage, and format. Basically data about the database.

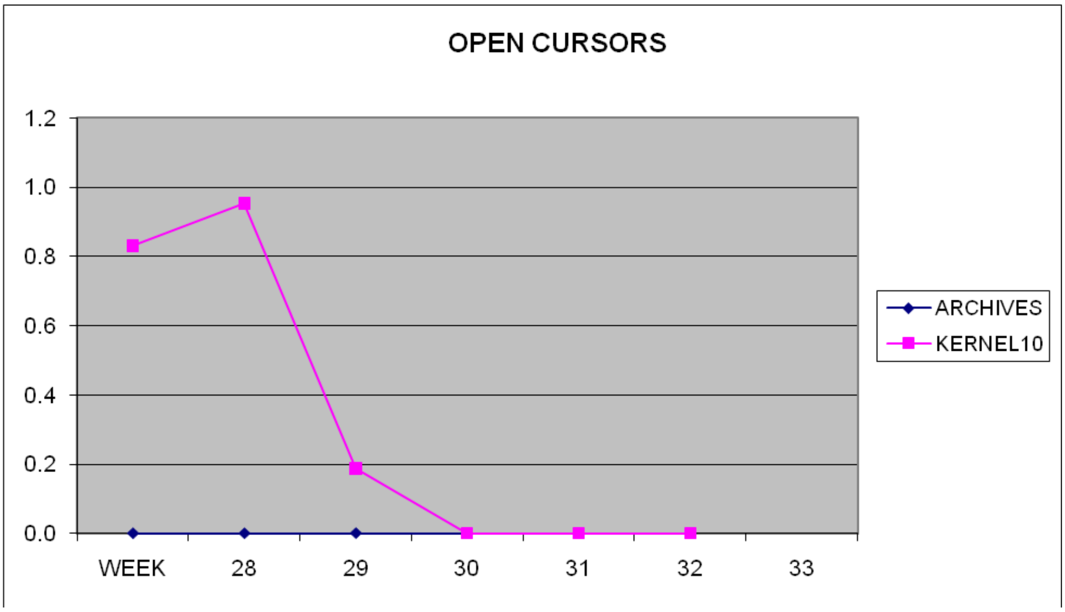
remove the occurrence overtime is to patch the actual database, as the issue was, in fact, a bug. It was deemed too risky to automate this (although Oracle 10g can be configured to do such) as there was a potential to introduce further problems or fail in patching, as well as altering the database version without sufficient change control approval within a regulated environment. So on first analysis, it would appear the problem went away (and it did) but this was because of human intervention outside the scope of the experiment (external action).

Note, this data dictionary view is truncated when the system re-starts, hence the memory leak alert will disappear with a database re-start.

Watch #10: Number of open cursors per session on database

This query collects data from a database data dictionary view regarding the consumption of cursors [30] by user sessions and the overall level per session of open cursors.

Graph 10.0: Watch #10: Max number of open cursors compared between databases.



Maximum cursor levels per session are set at a system level with an overall ceiling. Attempting to go above this defined level will prompt a user session error to be flagged and transaction to fail. If code isn't written efficiently, too many open cursors can be consumed or left open, wasting resources on the database/server. A lazy approach can be to set this ceiling so high that the user session will never consume all the cursors defined by the database. This is simply wasting resources that could be better used elsewhere and if code is very poorly written and fails to release cursors, sessions could potentially and eventually crash the entire system because of over-consumption. The S.H.A.D.E engine is defined with a "heal" to attempt to increase the allowable open cursors by following a step increase pattern with a maximum allowed level. It won't continue to increase indefinitely, hence it is not designed to repair the issue as it is caused by a

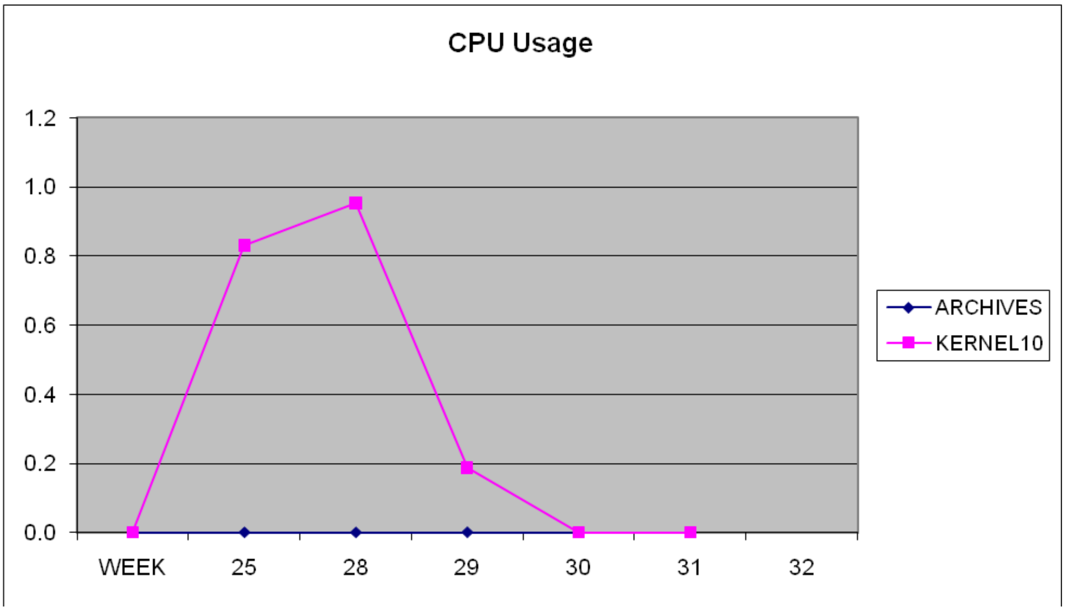
[30] Cursors are a mechanism by which a database client iterates over the records in a database.

potential bug in end-user code, but does however allow the database to be tweaked to improve performance, if the level is set to low for end-user performance levels and transaction level increases over time. From the graph, it can be verified that the S.H.A.D.E-enabled system experienced an open cursor consumption problem and reacted, the situation increased before the S.H.A.D.E engine was able to heal the problem completely and remain on top of the situation. The initial parameter change wasn't enough and the level was increased again by S.H.A.D.E after it was alerted again that the problem wasn't fixed after the first attempt. A subsequent increase fixed the issue and the error disappeared. The non-monitored system didn't experience any open cursor issues, which again suggests the potential need for fault-injection scenarios and engines in these type of experiments to ensure errors occur, as well as are replicated across systems. In this event, S.H.A.D.E was able to successfully heal the cursor problem or shortfall and remain on top of the issue by defining a new parameter for the database to operate under. It could be argued that a watch could also be constructed to reverse this setting if the cursor level is over-defined and under-utilised also, not to prevent an error in this case, but to reduce waste on the database server.

Watch #11: High CPU usage level

This query collects data from the database server regarding the level of CPU usage (weather multi CPU or Multi core).

Graph 11.0: Watch #11: High CPU usage levels between database servers.

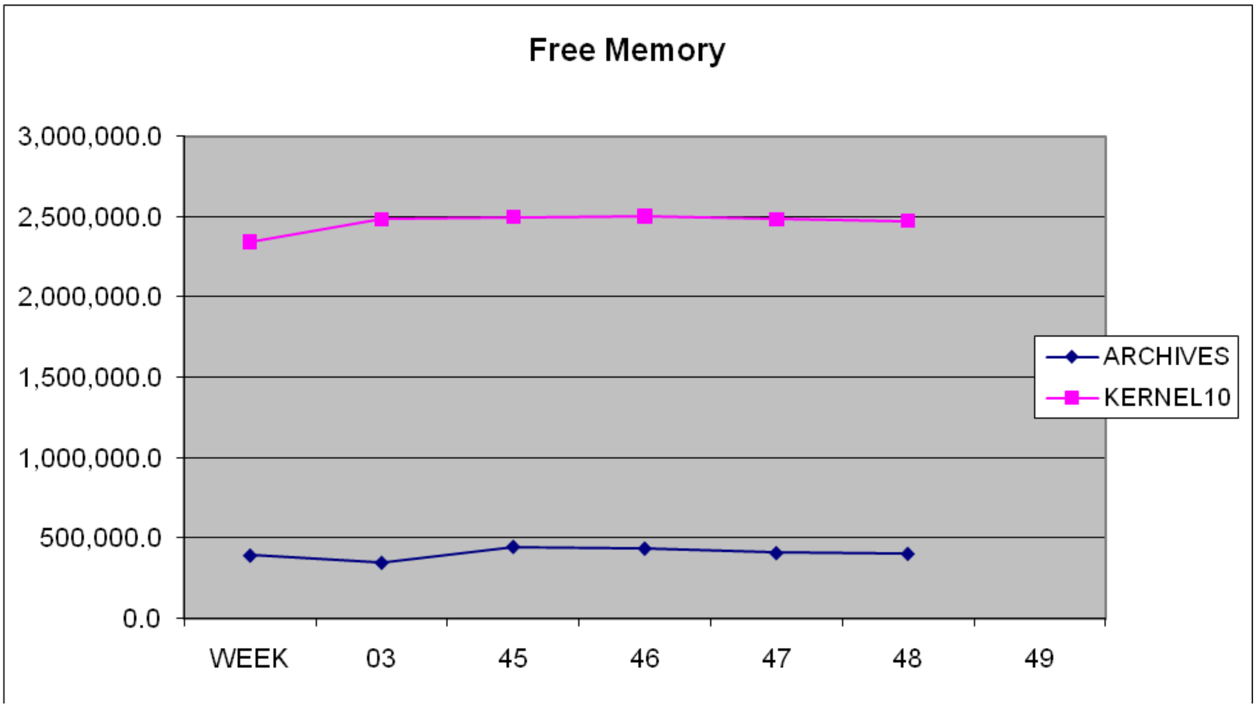


Alerted high levels of CPU consumption continued on the S.H.A.D.E-enabled system until the system was provided with more options or more knowledge in dealing with the recognised problem. S.H.A.D.E was given the ability to use the process kill command from the operating system level to kill processes if they were within a defined list of processes. These processes were known CPU-expensive users such as anti-virus processes. From the graph, the data shows that during early stages, S.H.A.D.E was unable to kill the process that was consuming the most because it was not defined as a process it could kill. After adding more "safe" processes, the S.H.A.D.E engine was able to improve and fix the situation. Later, S.H.A.D.E was configured to just alert the problem, but this example displayed the engine could be configured to make the right decisions without causing risk to the overall system's health – by being allowed to kill any heavy using processes, as some processes would be allowed to consume heavy loads of CPU; when they do so for too long a time period, it can affect the system's health. In the illustrated case, the anti-virus software was consuming and continuing to consume resources that the database needed. But only factors such as security levels on the server caused the ability to terminate processes to be less of an option (changes outside scope to experiment, but caused because of using real-world servers and not servers in a laboratory environment). The example does illustrate again when S.H.A.D.E has the ability and correct “knowledge”, it remained on top of the situation when errors and alerts were identified and raised when the system was flagged as being “unhealthy”.

Watch #12: Level of free physical "RAM[31]" memory on server

This query collects data from the database server regarding the level of free memory still available for processing.

Graph 12.0: Watch #12: Amount of free physical server memory between database servers.



In this example, the S.H.A.D.E engine failed in healing the error and as a result, the heal was "parked" and thus marked as failed after attempting all possible heals with the knowledge and parameterised options at hand. S.H.A.D.E was configured to stop a selection of services listed as non-essential to typical database server operation(s). Doing so frees up memory on the server for processing when the level of free memory was flagged by S.H.A.D.E to be on or above the defined "high water" mark and thus reaching a critical level whereby the Windows system could start to page memory[32] onto disk and slow down the overall system's operational performance levels. The monitored-only system continued to alert on low levels of free memory and this raises potential future problems if not

[31] Volatile memory (such as memory modules), where information is lost after the power off.

[32] Paging is an important part of virtual memory configuration of most operating systems, allowing them to use disk storage for data that does not fit into physical Random-access memory (RAM).

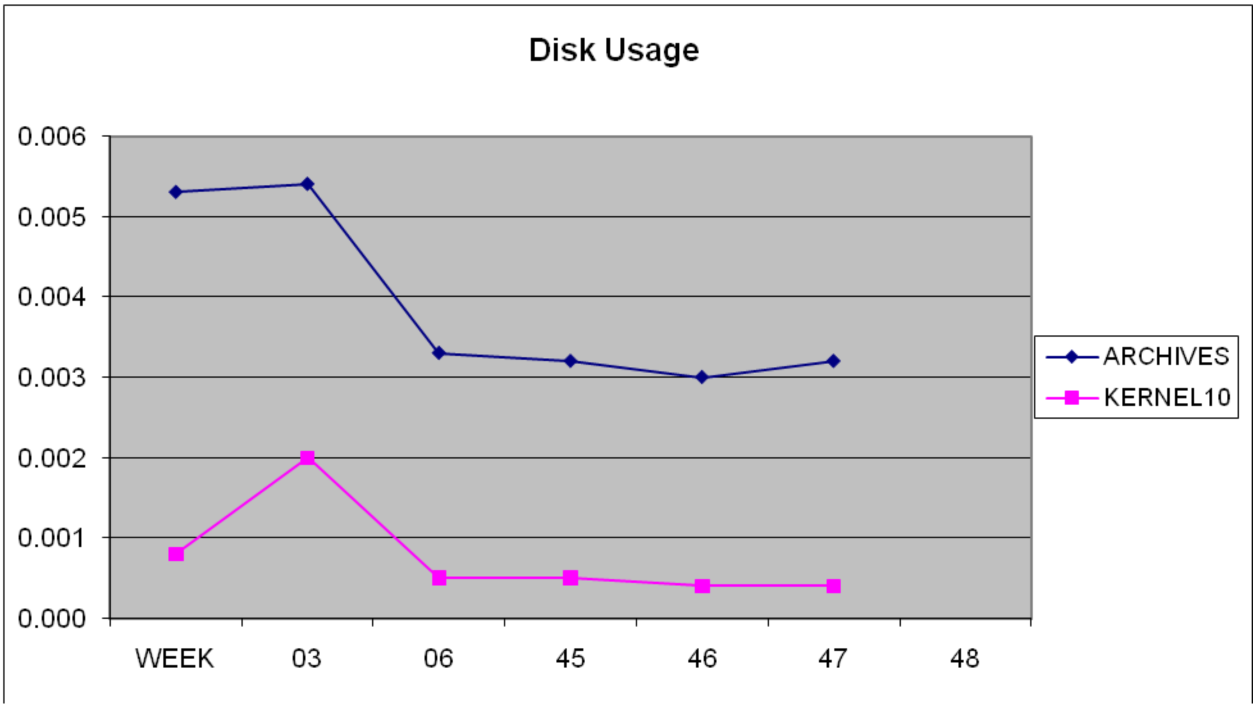
addressed. In this instance, S.H.A.D.E was unable to address the problem and failed to correct or heal the issues because of the restrictions within its own abilities or parameters. If it was running a combination of process termination as well as service management (using the same features or cross-healing features as used for CPU load reduction), and the ability to control more processes when identifying the processes that are listed are using too much (rather than alerting on low overall system RAM), it would better manage the problem. With these abilities, S.H.A.D.E could monitor, capture and stop a service that was consuming too much RAM and operating beyond its expected specification or even possibly using an option such as “poolmon”[33] to terminate/end a process or service that was experiencing an operating system level memory leak. Windows memory leaks are handled as a separate watch with watch 38, which is analysed and explained later in this chapter.

[33] Memory Pool Monitor displays data that the operating system collects about memory allocations from the system paged pools and used to detect potential memory leaks.

Watch #13: Level of disk usage (I/O) on server

This query collects data from the database server regarding the level of disk activity – disk I/O[34].

Graph 13.0: Watch #13: Level on disk usage (I/O) between database servers.



Disk I/O is often overlooked but is a common cause of poor system performance, as high disk activity will slow down overall system performance for operations not being executed in system memory/ram. Anything that requires disk activity will simply be slower, either if disk reading or writing. Within a database system, this will amount to most of the system operations as the database, if operational, will be either written to or read from (otherwise it won't be in use). Often administrators will look at CPU load and free memory and determine if neither appears to be an issue, then the system has enough resources. But if disk reads and writes are high, then the system will be slow. Making better use of memory or

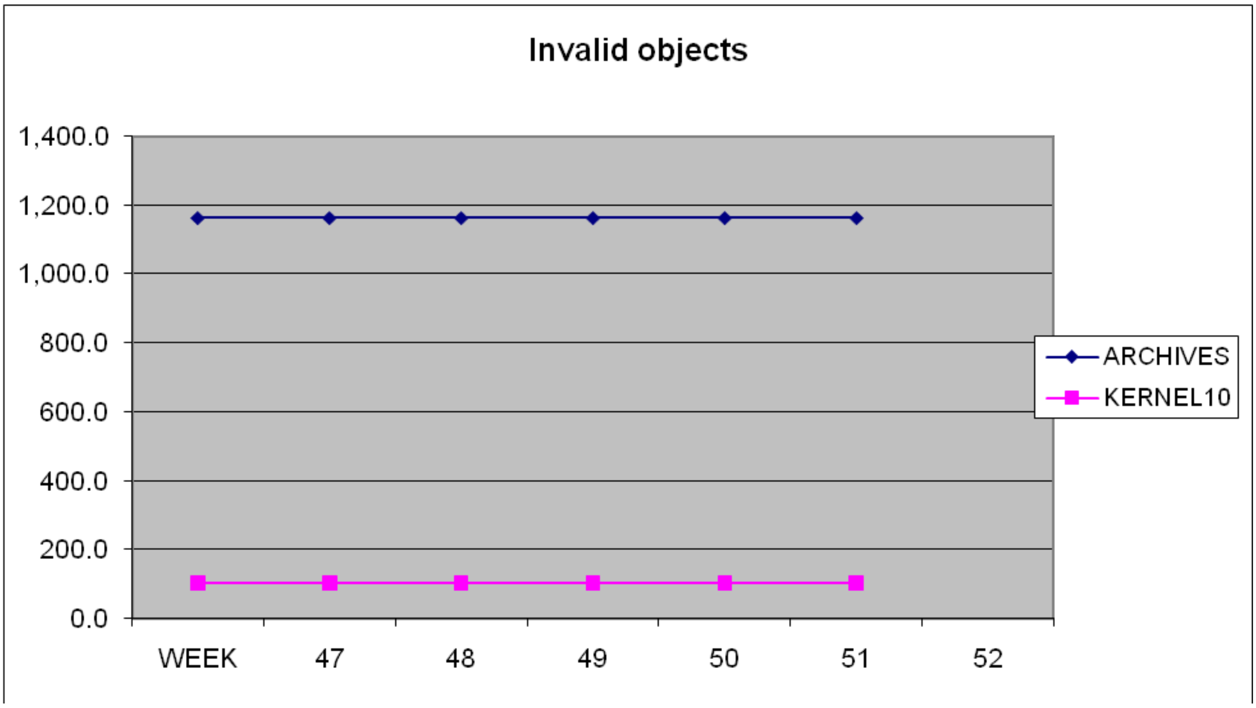
[34] Inputs and outputs or reads and writes to and from a disk device.

adding memory can help reduce this by making better use of caching, as RAM operations will always be faster than disk operations. Hence, the more there is conducted in memory, rather than disk, will operate quicker and should help to reduce contention. Initially, S.H.A.D.E was configured to increase the cache available to the database in an attempt to deduce disk I/O, which seemed to reduce disk activity on the system. It was later removed, as monitoring the database cache hit rate and adjusting to poor metrics here, was deemed as being more accurate an indication that the database would operate better with more memory assigned to it. Cache monitoring and healing will be discussed later in this chapter as part of watch 28. S.H.A.D.E would need to identify individual processes that were causing the high disk I/O by retrieving statistics from the performance monitoring elements of the operating system and adjusting as needed. But it was considered unsafe to terminate or end a service or process that appeared to be using too much in the way of disk operations without investigation, rather than reacting. An example is a virus scan which would have increases I/O during an update or scheduled scan, but you would not want to terminate for this reason alone. The healing ability of the S.H.A.D.E engine for this watch was thus removed and replaced with an alert to the administrator, making the human element of the system the one better qualified to deal with the potential problem. As shown, both systems experience disk I/O overheads as determined by the parameterised high water tolerances.

Watch #14: Invalid database objects in the database.

This query collects data from the database data dictionary regarding the number of invalid database objects on the database i.e. function, procedures, views etc.

Graph 14.0: Watch #14: Level of invalid database objects between database servers.



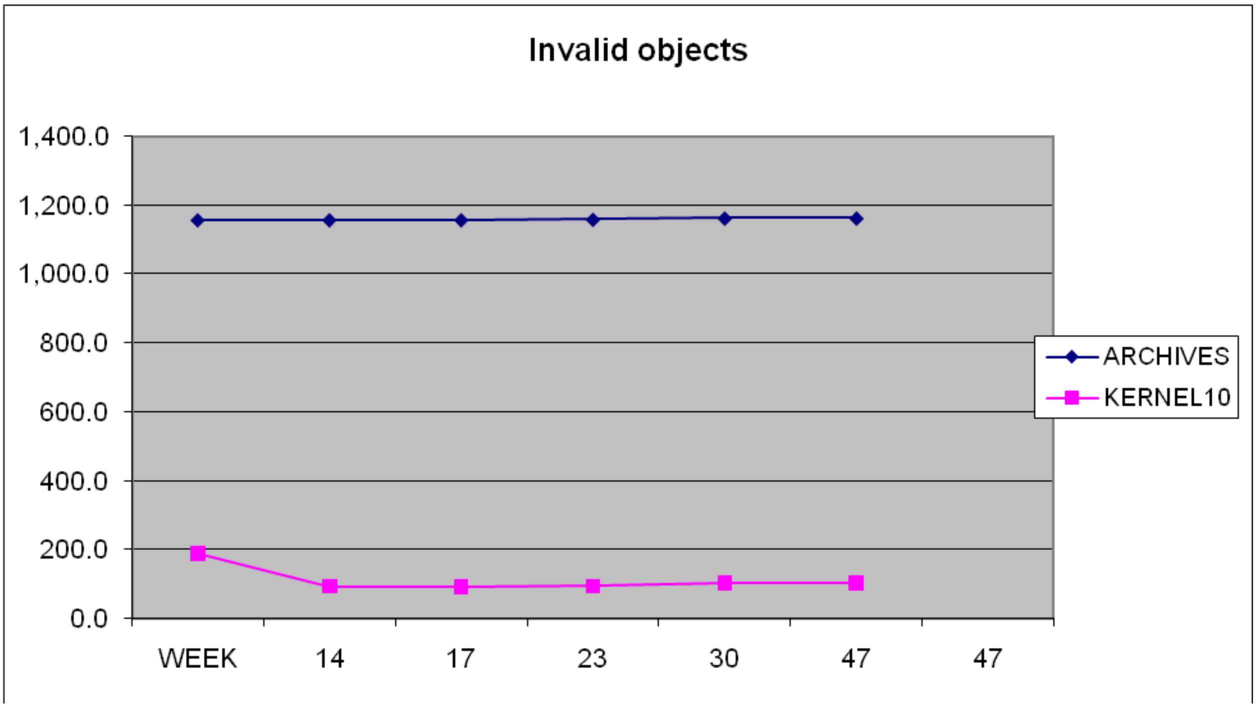
Invalid database objects are objects residing on the database with schemes that are in an invalid state because of references within them that don't exist or syntax errors that prevent the database residing code from compiling and thus being used. Objects can become invalid because of DDL [35] changes on the database. These types of objects can be re-compiled automatically to avoid end-user error. Objects that have syntax errors in code or invalid references will, however, require human intervention to re-write or remove. The graph shows S.H.A.D.E was unable to compile a selection of objects which, upon further investigation, did require human intervention. The objects in question reference database links and items that were not within the database and would require either code changes or links

[35] **Data Definition Language** (**DDL**) is a computer language for defining data structures.

to be created, otherwise the said code would remain invalid and thus unusable. In this case, S.H.A.D.E attempted to heal the issue and re-tried until it was marked as a failure and parked. The administrator was alerted and the broken healing option remained visible within the S.H.A.D.E user interface – remaining parked until the administrator placed it back into operation. In this case, the healing option would again fail and again be parked to prevent it from firing.

Looking back on a different sample of data, it clearly displays S.H.A.D.E was able to heal a percentage of invalid objects but was unable to compile the remaining invalid one, illustrated in graph 23.1 below.

Graph 14.1: Watch #14: Level of invalid database from older time sample

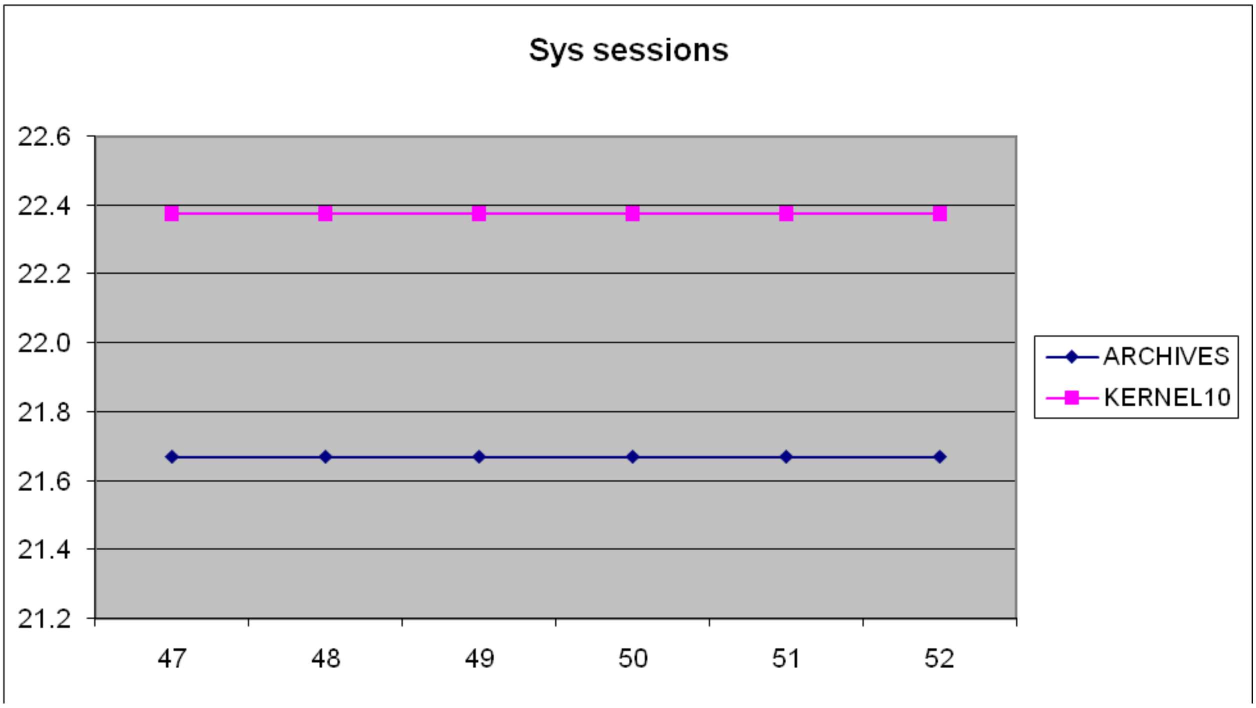


Manual compilation of the monitored system showed that most of its listed invalid objects remained invalid where they need not have; an automatic or manual compilation could have repaired them quickly.

Watch #15: Sys sessions on the database

This query collects data from the database data dictionary regarding the number of user sessions connected to the database using the sys account.

Graph 15.0: Watch #15: Level of sys users connections between database servers.

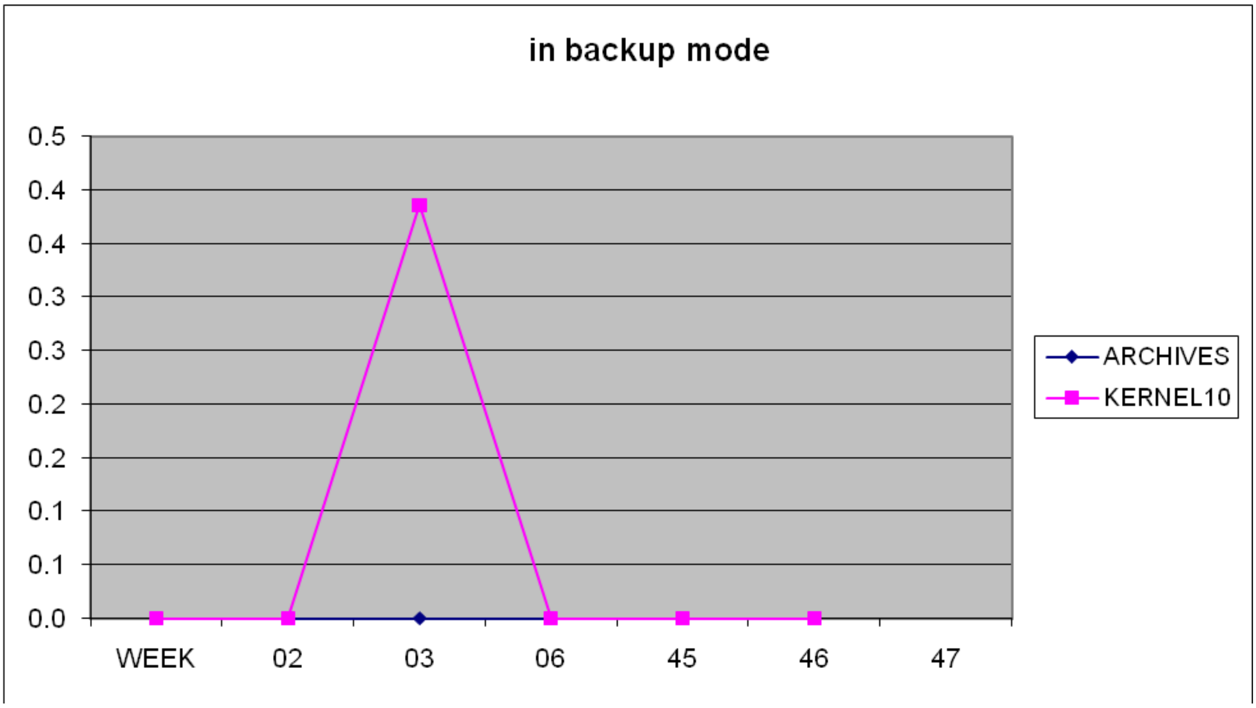


This watch is more of security monitoring that database/system error/fault. The S.H.A.D.E engine monitors for user sessions using the sys user which are sessions that are not running internal Oracle database processes. These sys users have the potential to be users on the system with too much power and access because of the account they are using or a “possible” attempted system breach (sometimes cause by administrators using weak or default passwords and not following password expiry procedures). The sys user has enough power to do just about anything on the database system and should be limited in access and monitored by use. Good practice is to create a named user with the same access for the administrators and not share generic accounts, giving the ability to track change and have accurate accountability for operations carried out. This watch was included into the building of S.H.A.D.E after the much-published Fannie Mae Attack in 2008, where the attack came from inside a company by one of its own UNIX developers when his job was terminated and he planted a logic bomb which would have brought down the companies 4,000 servers (BeyondThrust, 2009). Designed a healing operation or set of defined options for the engine proved to be too risky, as a genuine sys user could be terminated or locked out, which would in itself cause potential issues. S.H.A.D.E was configured to monitor and alert only. If the system was designed to manage or heal this possible issue, it would need to monitor terminal names, log-on times and sessions and determine which sessions are "most likely" genuine and which are not. Logic could even be built in to trace the type of transaction the session is carrying out through the use of either SQL trace or fine grain auditing. This type of heal or reaction to detecting these possible invalid sessions could be built into future versions of S.H.A.D.E whereby it will be evolving into not just a performance healing engine, but one that manages all levels of optimum system function ability, even security.

Watch #16: File stuck in back-up mode

This query collects data from the database data dictionary regarding the number of database datafiles stuck in back-up mode for longer than expected.

Graph 16.0: Watch #16: Level files stuck in "online" back-up mode between database servers.



Oracle online back-up allows files to be placed in back-up mode for back-ups to be taken. In this mode, the databases' table space and underlining datafiles are check-pointed when placed in and out of mode, to allow the database to track which of the current changes in archive logs/redo are needed to recovery these files. Leaving the files in back-up mode can affect the system if it shuts down; it will need human intervention to start up as well as generating more redo while the file is in hot back-up mode. The main risk to the system, however, is that if it remains in hot back-up for a long period of time and then needs recovery at a later date, the database will need every archive log generated from the time it was put into hot back-up until the time it ended hot back-up. Most careful administrators keep at most a couple of weeks' archive logs (not reference an example of using

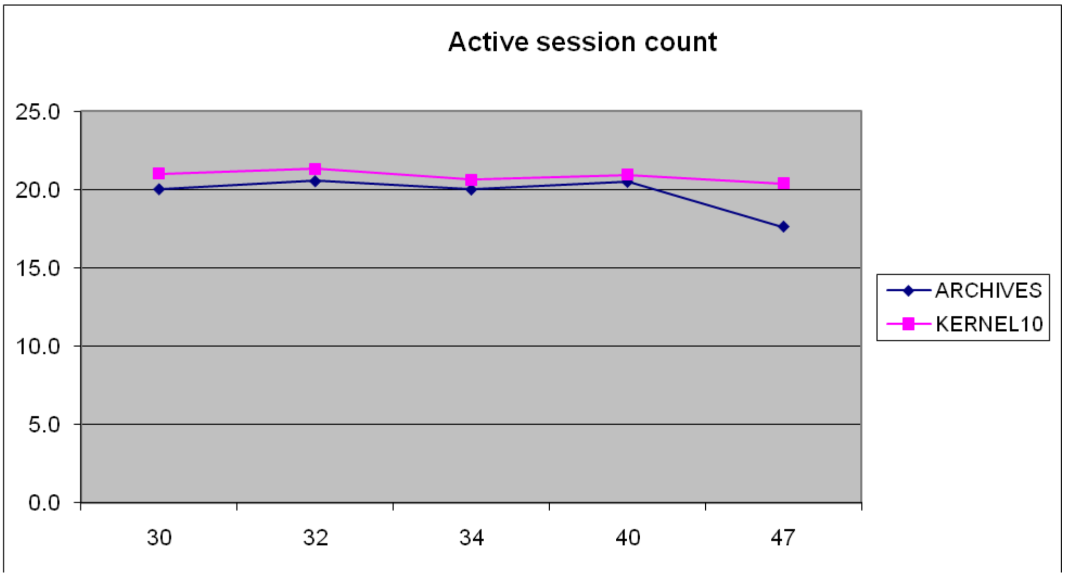
RMAN[36]); hence it could possibly cause an administrator to look for several weeks, if not months, of archive logs if the file in back-up mode is not detected. What does this mean? Potentially, the administrator will not be able to recover or fully recover the event from a disaster. From the graph, S.H.A.D.E was able to detect and alert on the detected issue; here, the files in question remain in back-up mode for a couple of days, S.H.A.D.E then follows it own defined heal for this problem and removes the back-up mode, the alert disappears and the problem doesn't arise again. The monitored-only system is not in archive log mode and thus is unable to perform "hot" online back-ups, thus the issue is not a possibility for this database. The database replies on offline "cold" back-ups whereby the system is periodically taken offline and the files are taken to tape.

[36] A command-line and Enterprise Manager-based tool for backing up and recovering an Oracle database.

Watch #17: Active end-user session count on database

This query collects data from the database data dictionary regarding the number of active user sessions and uses such as a “guide” to overall poor system performance through maxed CPU, hung database, locking or overload disk.

Graph 17.0: Watch #17: Level of active user sessions between database servers.

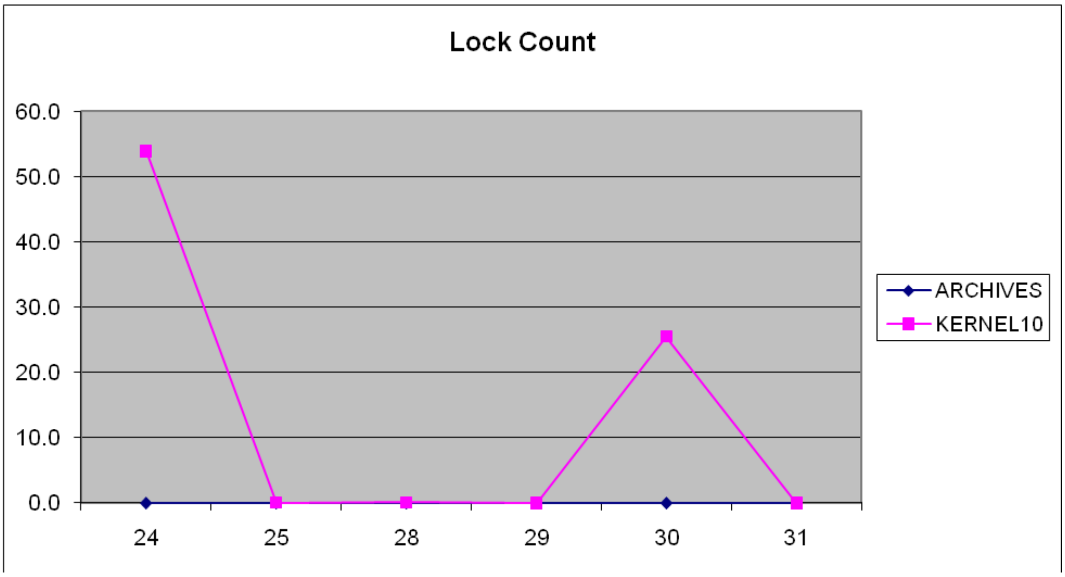


These sessions are considered active not just when they are operating on the database, but when they are actually processing SQL against the database. Typically (with the tested supplication in question), user sessions process within a few seconds, some longer-running queries can take minutes for batch processing or reports. But if a large number of sessions appear as active and remain as such, with the number increasing, it can be an indication the system is performing poorly. Treating such as an impending "potential" system hang, the S.H.A.D.E engine dumps system state to disk for early analysis and alerts the administrator. It would be an unwise attempt for a healing system to guess the cause, as there can be numerous reasons for such an event, from repeated client connections, through to the database stuck in a hang state. Hence, the engine may only add to the problem by attempting to fix this type of issue (without all the facts). S.H.A.D.E will have prepared a system state collection of data on the health and status of the system as it alerts the administrator. The administrator should have enough information at hand from this system state, to further analyse the issue and start the analysis of the cause of the potential problem. With this information at hand, they will not have wasted valuable time working out if and why and would be moving forward with looking into the issue and possibly opening a support call with relevant vendors. Both systems alerted high session levels and were flagged as having errors, levels did increase and decrease during monitoring, but S.H.A.D.E didn’t have any actions that could have been executed/fired to alter these.

Watch #18: Dead locks level on the database.

This query collects data from the database data dictionary regarding the number of database deadlocks detected on database objects.

Graph 18.0: Watch #18: Session deadlock levels between database servers.

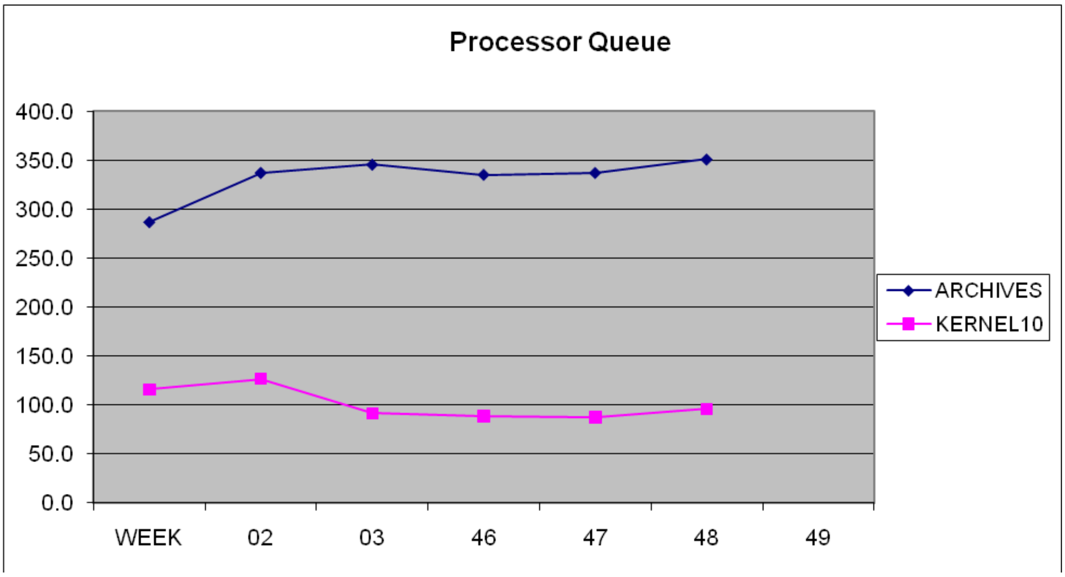


Deadlock detection monitors the database for objects that have been locked as a result of standard transactions. When this happens, other sessions or transactions that need the object will hang or queue for the resource until it is somehow freed. S.H.A.D.E detects the issue and kills only the session holding the session and the resource after a time period, basically when it decides the resource/session will remain in the same condition unless it intervenes. From the graph, it can be seen that S.H.A.D.E healed the issues whenever it arose on the S.H.A.D.E-enabled system. The monitored-only system experienced no deadlock issues. This is again another example that prompts the use of a fault-injection engine with these types of experiments.

Watch #19: Processor queue usage.

This query collects data from the database system regarding the level of processor/CPU run queue in use, when the system consumes more CPU that is on the server and it moves to the processor queue.

Graph 19.0: Watch #19: Processor queue usage between database servers.

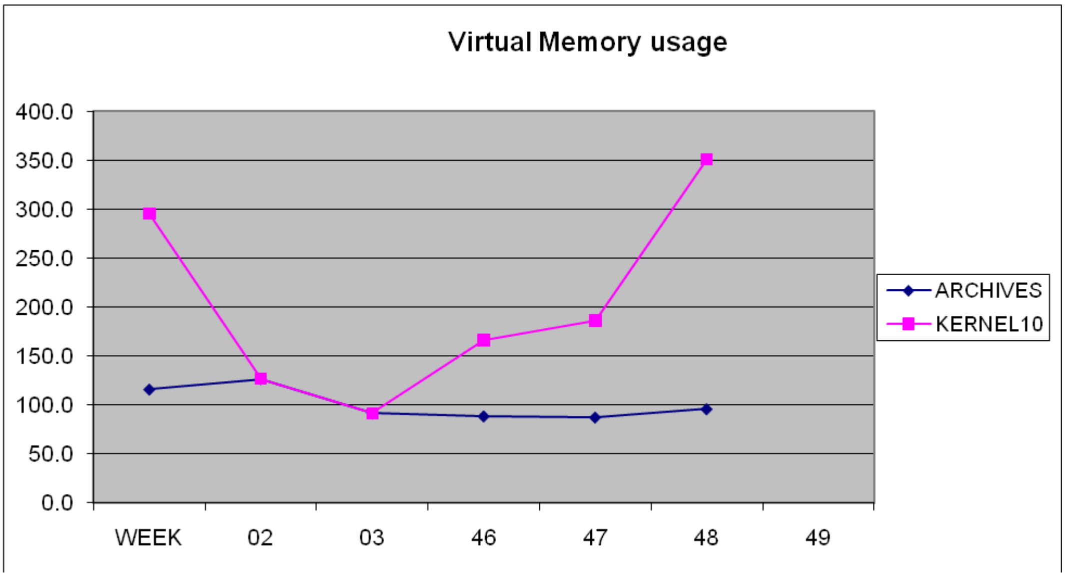


In much the same way as watch 11 earlier (CPU consumption), S.H.A.D.E was configured in earlier stages to identify and kill processes that were flagged as heavy users of CPU. Processor queue usage would be used when long-running high CPU was used to a level of maxing out the CPUs on the database server. S.H.A.D.E was given the ability to use the process kill command from the operating system level to kill processes if they were within a defined list of processes. From the graph, the data shows that during early stages S.H.A.D.E was unable to kill the process that was consuming the most because it was not defined as a process it could kill. After adding more “safe” processes, the S.H.A.D.E engine was able to improve and fix the situation. Later, S.H.A.D.E was configured to just alert the problem, but this example displayed the engine could be configured to make the right decisions without causing risk to the overall system’s health. The monitored-only system continues to have high amounts of run queue consumption for all CPUs; with no healing enabled, the system alerted the administrator.

Watch #20: Database process virtual memory usage

This query collects data from the database system regarding the level of virtual memory bytes used by the Oracle database process.

Graph 20.0: Watch #20: Virtual bytes usage for database process between database servers.



Depending on the server memory configuration – whether it is configured for normal or extended memory usage - the ceiling for virtual memory usage varies, but there is a ceiling nonetheless. Consuming above this point will flag user errors and prevent further database sessions from logging onto the database. In normal systems, where the level is not monitored, the system will flag an error and the end-user will know before the administrator. In all, this is a less than suitable scenario. S.H.A.D.E monitors the level and flags a warning when it is close to reaching the 1.6 GB ceiling or 2.6 GB for extended memory assigned servers. S.H.A.D.E was initially configured to reduce the database SGA [37]memory usage on discovering an issue, but this later caused the system to fight itself. Reducing the cache level to avoid virtual memory error dropped the cache hit rate, which prompted S.H.A.D.E to increase it again and then reduce and increase. S.H.A.D.E was thus configured to alert the administrator only, prompting the human element

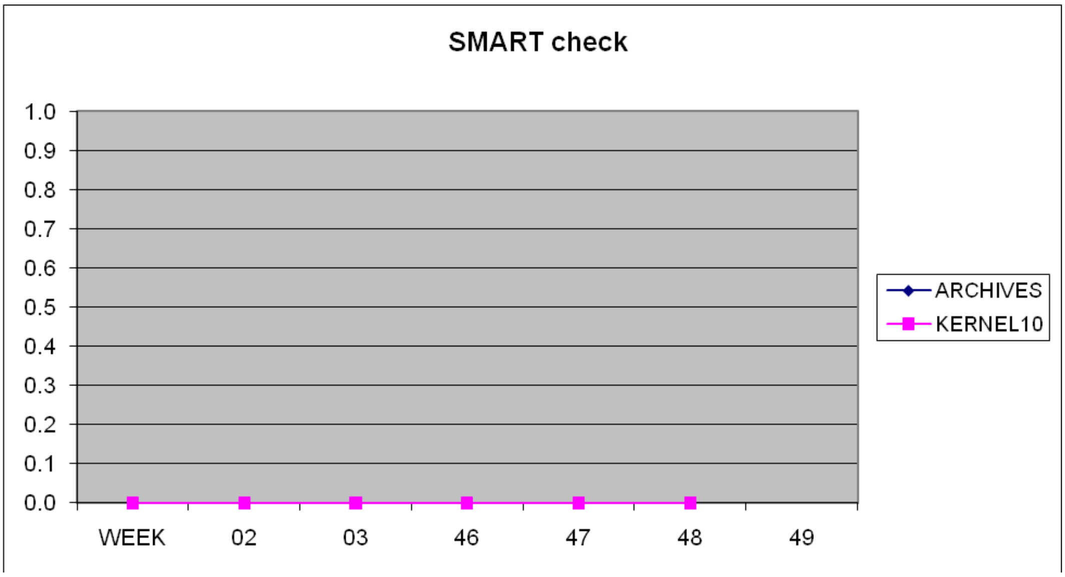
[37] The SGA is a chunk of memory that is allocated by an Oracle Instance and is shared among Oracle processes and used for database operations.

of the system to make the best call in reducing database memory usage, or per session memory usage. From the graph, S.H.A.D.E was able to deal with the alert, but after the “heal” was set to alert only; the problem remained an ongoing issue. The alerts were less frequent on the monitored-only system; this could be caused by smaller session counts on this system, again prompting for a fault-injection engine to ensure the same issues arise on both systems at the same time.

Watch #21: Database server SMART alerts

This query collects data from the database system regarding the level of SMART alerts on the database servers.

Graph 21.0: Watch #21: SMART alert levels between database servers.

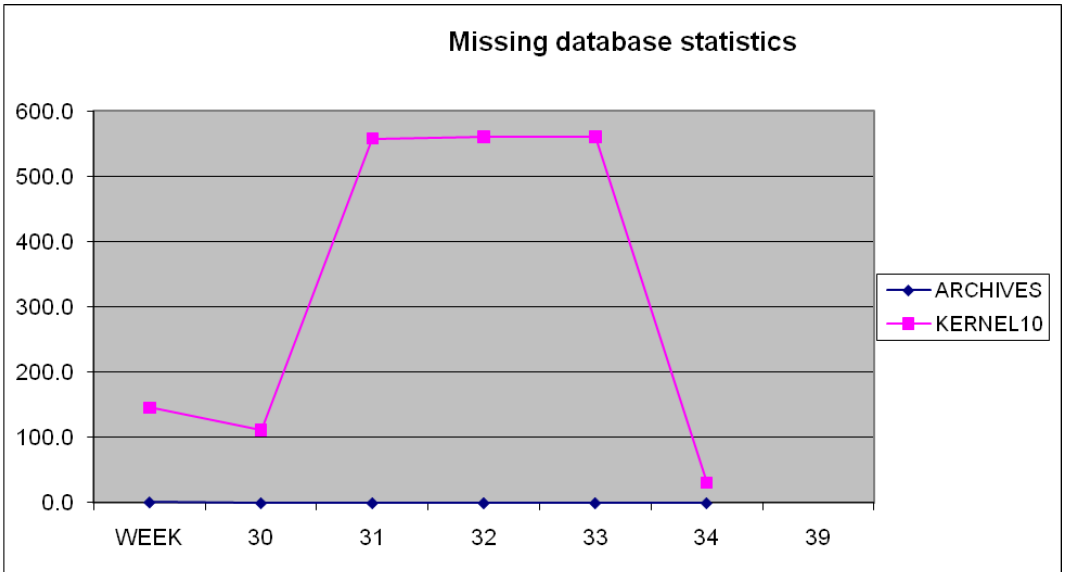


SMART alerts were included as a proof of concept for early disk warnings and alerts if, and potentially when, a disk may fail. To gain full use from this type of software detection, it would really need to fully integrate with the hardware, flagging the disk that has degraded and alerting that it needs replacing. Even rebuilding it from a pool of spare disks and alerting later could be an option. These types of disk degrading flaggings are used in most modern SAN environments such as the system offered by EMC and HP. The administrator receives an alert that a disk could potentially fail in the future and acts on it then, rather than actually waiting for the disk to fail. Here S.H.A.D.E does the same, but was unable to retrieve SMART information from the server disks as the option was not available on the monitored servers. The option was tested during earlier stages of the S.H.A.D.E development and the option worked and remained in place as a proof of concept that would be developed to support server side as well as desktop side technology for disk monitoring.

Watch #22: Database objects missing statistics

This query collects data from the database data dictionary regarding the level of objects with missing database statistics within each of the databases.

Graph 22.0: Watch #22: Objects with missing statistics levels between database servers.

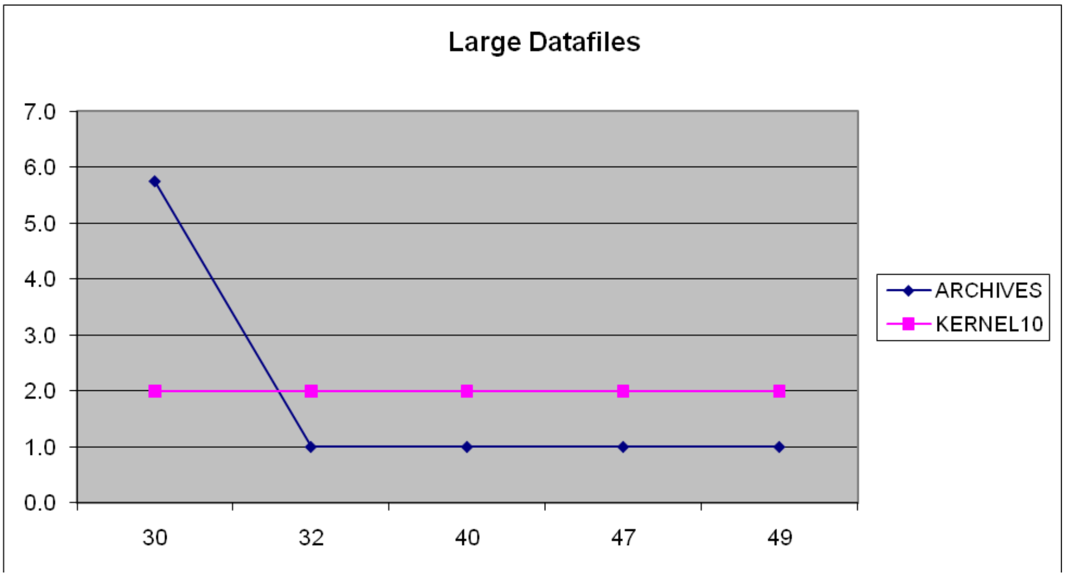


Database statistics on an Oracle database are often overlooked as a common cause of poor performance. If tables/indexes grow or new objects are introduced, the database needs to be made aware of their existence. Oracle has automated the statistics gathering with database residing jobs in 10g onwards. But administrators have learned that, sometimes, larger growing objects benefit from the old-fashioned way of gathering using computer objects'. Doing so simply informs the database how many rows are in the table and which indexes/locations are in use. Oracle will monitor objects and only gather statistics for objects that it sees having changed enough to gather statistics on. During the design of S.H.A.D.E., experience prompted the monitoring and gathering of statistics for objects with no statistics gathered for a week. If S.H.A.D.E finds objects older with still no statistics or out-of-date statistics, it will gather them immediately. From the graph, it can be seen that S.H.A.D.E took a small amount of time to "get on top" of the problem as so many objects continued to have old or no statistics gathered (even though the option for automatic statistics was left operational in the 10g database), but managed to gather older and new objects' statistics to the level that it remained on top of the problem. The monitored-only system only reported minor issues; again, this was because a more natural change was happening in the healing S.H.A.D.E system, because the same faults didn't exist on both systems at the same time. If automatic statistics weren't gathered, it would show very similar amounts of errors and repairs each time.

<u>Watch #32: Number of large database datafiles detected</u>

This query collects data from the database data dictionary regarding the number of large database datafiles that are so large they may hit a high water mark for the present database configuration and fail to expand.

Graph 23.0: Watch #23: Large database datafiles levels between database servers

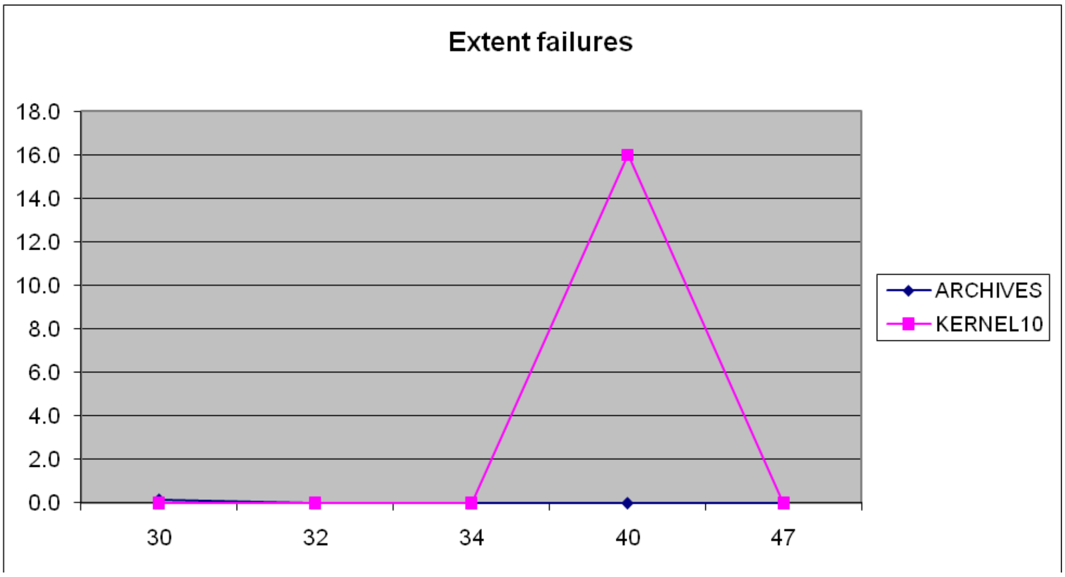


Depending on the database block size, in this case 8k, the limit to any file is 32 GB max. If this file is set to expand and no other file has been created for the tablespace (as Oracle "large file" option not in use i.e. one file per tablespace), the database will error out. In this instance, S.H.A.D.E will detect a file that is close to the limit, alert it as a future "potential" issue and add a datafile, as needed, to the flagged tablespace. This will allow the database to continue to grow into the new file without hitting a space limitation. From the graph, it is visible that S.H.A.D.E was able to detect and fire a "heal" by adding a new file, but still continued to experience a single file alert after the repair. The engine was unable to fix this issue as it was an oversight in the design of the healing process for this watch. Even though the engine was able to fix the issue, it still flagged the same large file as a problem and attempted to fix it again and again. Eventually, it parked the "heal" as failed, as the issue could be amended, even though it was fixed. The engine would need to ignore the condition if it already reacted to and fixed it. In this case, it would need to run a verify script and flag the file in question to be ignored in the future. This would need to be built into later builds/versions of S.H.A.D.E as it would require further code and parameter changes, as well as testing and verification.

Watch #24: Number of "possible" future extent failures on the database.

This query collects data from the database data dictionary regarding the number of "potential" database extents that may fail if the database needs to expand an object (table or index).

Graph 24.0: Watch #24: Possible extent failures levels between database servers

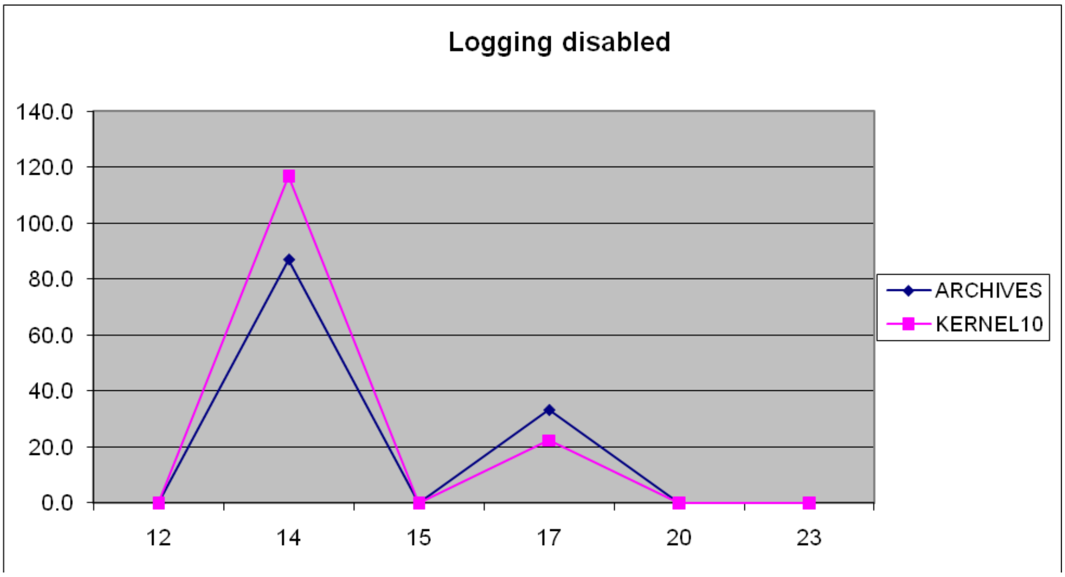


Because the files are set within the database as locally managed, the overall number of extents per object wasn't monitored for issues i.e. any more than 10 extents being flagged as a potential performance issue with the object flagged for future rebuild as a result. This watch was concerned with objects that could, and eventually would, fail due to space limitations within the tablespace in which they reside. If the healing component of S.H.A.D.E is set to fire, it automatically resides in the tablespace/datafile in which the detected object resides. From the graph, S.H.A.D.E detects a selection of "potential" objects that will fail in the future if no more space is assigned (the file could be set to auto-expand to a limited size), S.H.A.D.E heals the situation by re-sizing the file to bigger than it is (and below the 32 gb limit) and the issue is no longer flagged because sufficient space is now in place with the file. With the monitored-only system, space was not an issue which again prompts for use of a fault-injection engine to ensure all problems happen on both systems. But since these systems are in "real-world" use, faults are detected because they would be issues through standard operations on a daily basis.

<u>Watch #25: Number of objects with logging turned off on the database.</u>

This query collects data from the database data dictionary regarding the number of database-residing tables and indexes that have logging turned off.

Graph 25.0: Watch #25: Number of database objects with logging turned off, compared on both database servers

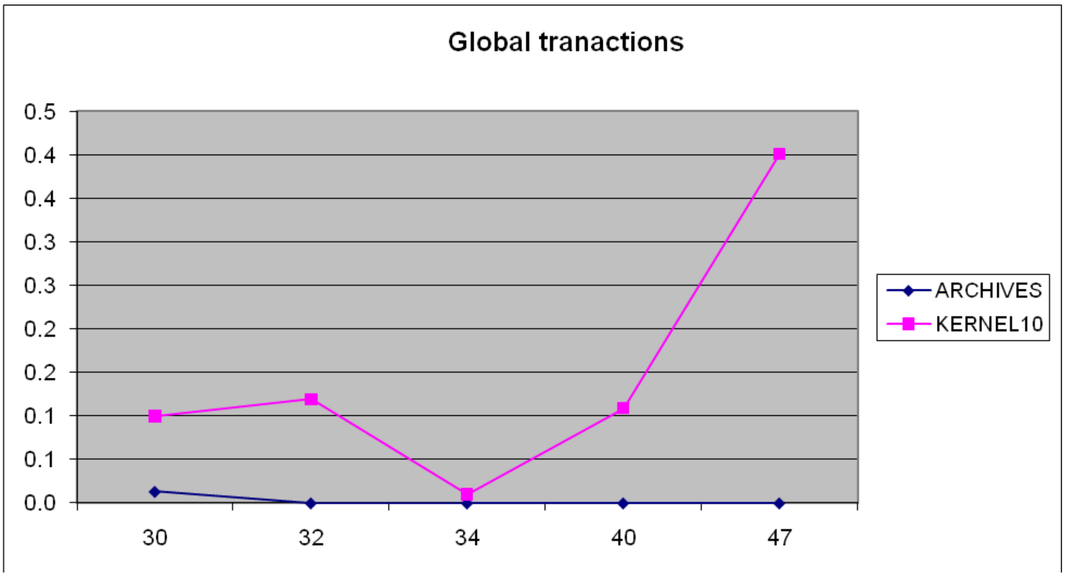


Sometimes, objects can be created or altered to turn logging off. With logging turned off, the object will not stored re-do data in the database's archive logs and thus the administrator will not be able to recover these said objects if the database needs to be recovered and rolled forward to a point in time. S.H.A.D.E monitors the database for any objects that have logging turned off; when it discovers any objects, it loops through turning logging on. When the next point in time, hot or cold back-up is taken; these said objects can be recovered as all changes to these said objects after that back-up will now be logged. From the graph, S.H.A.D.E effectively detected issues with old and obviously new objects, as the issue arose again after the first heal, after which the problem went away. Interestingly, S.H.A.D.E appeared to detect issues in the monitored system but the issues didn't remain. After further investigation, it was discovered S.H.A.D.E was set to heal both systems by mistake, the heal was removed from the monitoring system and the issue didn't arise again until new objects were created.

Watch #26: Overall level of global transaction that remains active

This query collects data from the database data dictionary regarding the number of active global transactions [38]or transactions that query from another "linked" database.

Graph 26.0: Watch #26: Global transactions levels between both database servers



The number of global transactions in a database is set to a defined limit. If this limit is reached, further global transactions will fail and raise an error. Often the default parameter is set and Oracle continues until the limit is reached; when an error is met, then and only then, does the administrator increase the parameter and the cycle continues until the issue is met again – if global transaction levels continue to increase with code/application changes. S.H.A.D.E monitors the level in use against the level set within the Oracle database (parameter) and flags an alert to the administrator when the database is getting close to the defined limit. From the graph, it is clear the healing instance experiences issues and continues to flag the administrator who fails to act upon them by increasing the database parameter. This reaction is typical, but in this case S.H.A.D.E hasn't been given the ability to alter sessions and it was deemed too risky to simply kill these

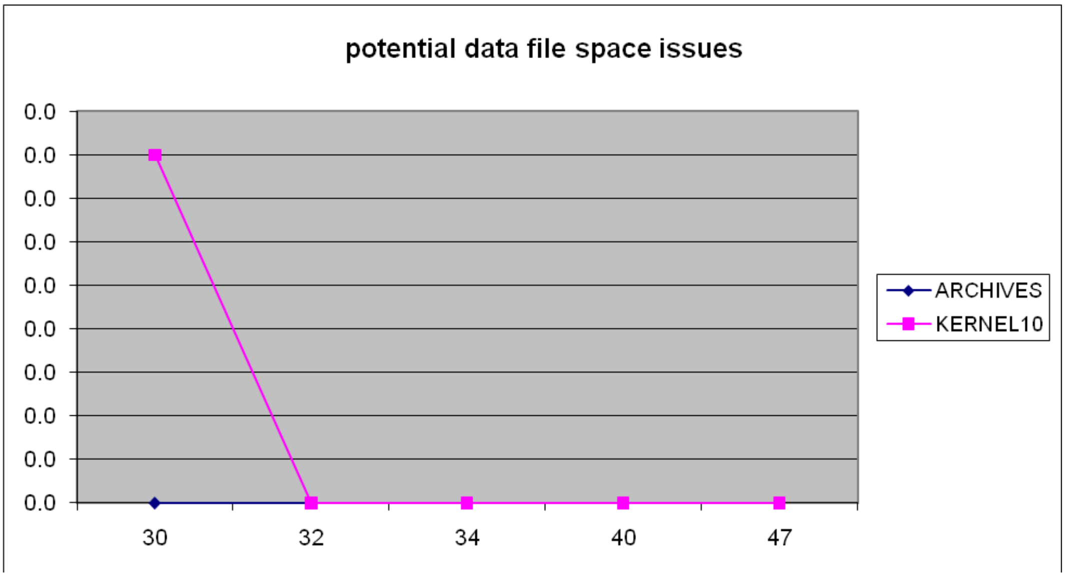
[38] A global transaction, or distributed transaction, is a set of two or more related transactions that must be managed in a co-ordinated way. Typically, the transactions are in different databases and often in different locations.

sessions without gathering more information on what they are doing, if anything at all. In future builds of S.H.A.D.E, the engine could set a timer and trace the sessions after they were detected as being active too long, dissecting what they are actually doing on the database and killing the sessions if they are marked as being “ghost” – dormant sessions or sessions not needed. The monitored-only system experienced some, but not many, global transactions, as there was little interaction within the database to other databases.

<u>Watch #27: Potential datafile space issues.</u>

This query collects data from the database data dictionary regarding the number of potential datafile space issues where by a file needs space and is not set to auto expand.

Graph 27.0: Watch #27: Potential datafile space issue levels between both database servers.

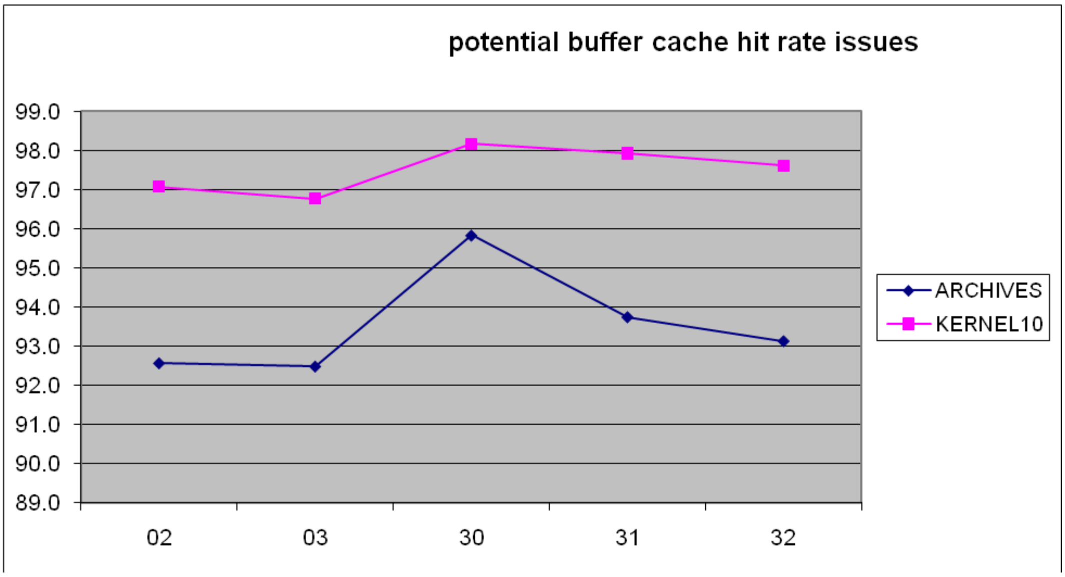


A quick way to set Oracle files to self-manage (with limitation) is to set a datafile to auto-expand by a realistic value of space i.e. 500 megabytes. This will prevent the database from hitting a storage wall and faulting, although this is a lazy approach, as it would be more efficient to monitor and adjust files with growth and time. Adopting a combination of both is the wiser choice. If a tablespace has more than one file, set the last file to auto-expand and move/create another as this file grows beyond a desired level (or restriction). S.H.A.D.E adjusts itself by using a combination of watches/heals to deal with these problems. In this watch, if the file is not too large in size, it will be adjusted to auto-expand. If there is another or a number of files in the tablespace, the last file in the set is the only file to be set to auto-expand, so the files spill over from one to the other, as needed. If the file is too large, the heal associated with watches 23 and 24 (as discussed earlier) will adjust the size or create new files, as needed, with the last file expanding and monitored, as needed. From the graph, heal detects and fixes the issue with no further problem highlighted, because any files with potential future issues are already set and remain as such (unless a human administrator changes, in which case it will automatically change back as S.H.A.D.E feels fit). The monitored-only system reported no issues, as space didn’t become an issue or, in this case, potential issue.

Watch #28: Potential buffer cache hit ratio problem on database

This query collects data from the database data dictionary regarding the hit ratio (health) of buffer cache hit ratio on the database.

Graph 28.0: Watch #28: Potential buffer cache hit ratio issues between both database servers.

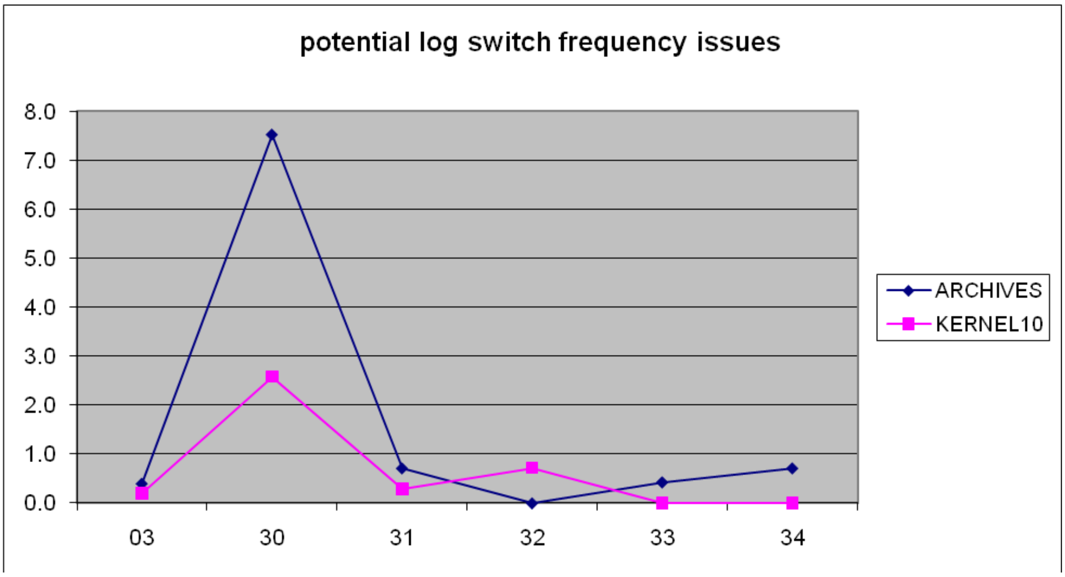


The buffer cache hit rate is a SGA/Data dictionary statistic within the database, used to gauge how effectively sized and used that the cache area of memory used by the database is. Often, it is either over- or under-sized, both of which affect performance of the database and waste resources. Oracle 10g can dynamically adjust the memory allocated from an assigned pool of memory. Moving memory as it sees fit between the databases, various memory resources and needs. But this setting doesn't work with extended memory settings on a Windows server with the database set to use both direct and indirect buffers. In this case, if the database is set to use standard buffer cache settings, the dynamic controls or self-adjustment settings of Oracle are disabled. S.H.A.D.E monitors for continuous low cache hit rates and increases the buffer cache in increments, with further monitoring until the database hit rate hits an acceptable, parameterised high enough level, i.e. over 99.8%. If the first change failed to improve the hit rate, S.H.A.D.E will increase it further and repeat this action with a limitation. From the graph, S.H.A.D.E increase the buffer hit detected, but still reports it as an issue. In the illustrated case, S.H.A.D.E parked the “heal” and alerted the administrator. The lower hit rate was, in fact, caused by system re-starts for patching, which means the system didn’t have enough time to read data into cache. The watch heal was re-activated with an increase in the timing for S.H.A.D.E to re-check as it takes times for the buffer cache to improve if increased in size; the system also required to be up longer to all, as much parsing of data/sql as required to emulate normal system running. The hit rate on the monitored-only system continued to remain poor, as no adjustment was made by either the engine or the administrator (although the issue was flagged and alerted by email).

Watch #29: Potential log switch frequency issues on database

This query collects data from the database data dictionary regarding the amount or level of log switches (too many) on the live database server.

Graph 29.0: Watch #29: Potential log switch frequency issues between both database servers.



Frequent log switches can be a quick indication the system is handling a lot of redo within its redo logs39. If too much switching occurs, it can flood the database and affect its ability to handle redo and archive logs for redundancy, as a lot of redo switching will occur which requires system resources (from operating system and database point of view). When the database switches above a set limit, S.H.A.D.E heals the issue by increasing the size of the redo log files, increasing the time it will take them to fill up and then switch to the next file in the set. S.H.A.D.E also alerts the administrator whoi needs to be made aware from a space/back-up point of view that this change occurred. Redo switches can increase during busier operating, like batch processing times (as illustrated in the graph where the monitored-only system experienced a high switch rate for a period and then continued to have the problem, but it levelled out on its own). Hence, it would be better if S.H.A.D.E analysed its own history for the event and reacted on an average over a desired time period (which it was configured to do), allowing it

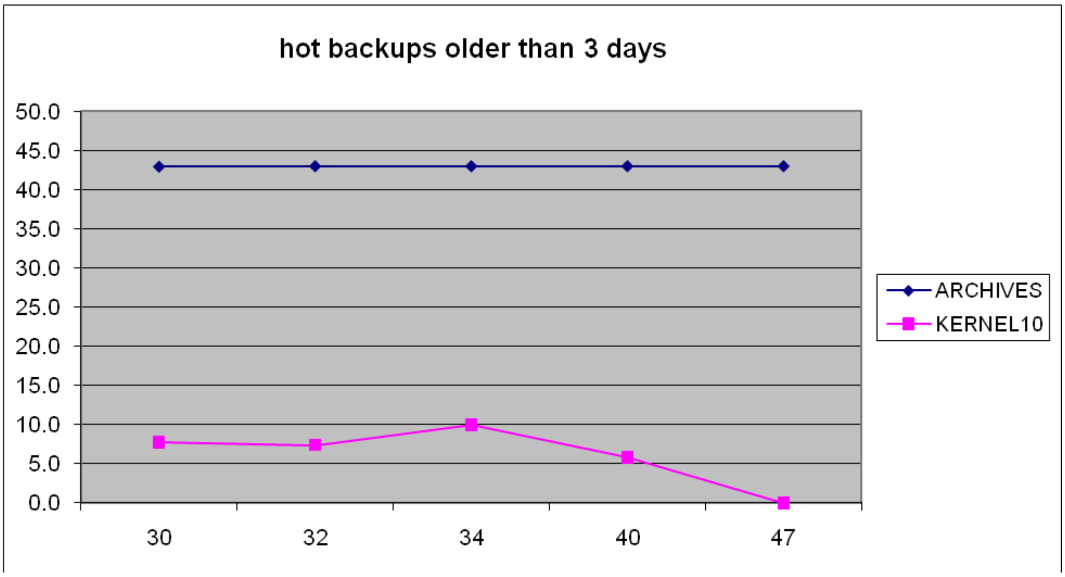
[39] Before Oracle changes data in datafiles, it writes these changes to the redo log.

to take more time and database operations into account before making a change that will impact the system. Reacting incorrectly won’t affect performance, but setting the files too high when not needed could potentially affect the database’s ability to recover to a point in time if needed, if other log timings’ parameters are not in use. From the graph, S.H.A.D.E appears to handle the alert/error by increasing the log files until they no longer are an issue.

Watch #30: Potential issue with hot backup executions.

This query collects data from the database data dictionary regarding the amount of times the database has been backed up using the hot back-up function, if enabled. In this case, S.H.A.D.E flags a potential issue when any or all of the database files haven't been put in hot back-up mode for a defined time.

Graph 30.0: Watch #30: Hot back-up frequency issues between both database servers.

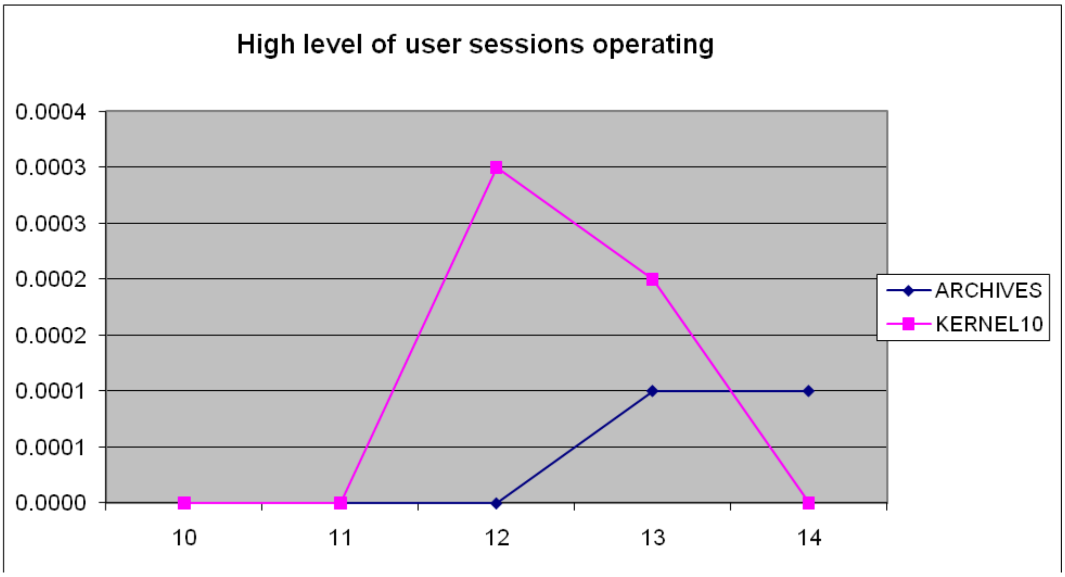


Hot back-ups are "usually" conducted on a daily basis, but are dependent on a number of factors. Amount of change on the database, size of the database, are incremental back-ups in use (where only change is backed up). All of these are different factors and affect how often each and every block of the database is backed up to disk/tape. In this case, RMAN is not used and standard hot back-up from disk to disk is conducted on a daily basis using scheduled tasks/jobs on the database server. If these jobs fail to execute or if the database is altered without the back-up script being altered, S.H.A.D.E intervenes and fixes the issue, but making sure a hot back-up is conducted. S.H.A.D.E only alerts on the issue when a three-day window has passed and it decides (running under its own defined parameters) that this condition is an error/fault and that it should repair the problem. From the graph, S.H.A.D.E reacts and fixes the issue where possible. In the illustrated example, a back-up script was failing and new files were later added. Eventually, S.H.A.D.E was on top of the problem and prevented a potential issue. If a new file is added to a database and the back-up script is not altered, it will prevent the administrator from recovering the database completely or at all. Older back-ups also put the database at risk from a recovery standpoint as the administrator must have each and every archive log since the back-up to allow him/her to roll forward through all changes. If one file is missing from tape or disk that will be as far as the recovery can go. Taking frequent “fresh” back-ups will help reduce the risk. The monitored-only system is flagged constantly, even though hot back-ups are not possible as the system is not in archive log mode. S.H.A.D.E is correct in flagging it as a potential issue as no checkpoints have been issued on the system, so the administrator is alerted and reminded that a cold back-up must be carried out on the said system, otherwise recovery will not be possible if needed.

Watch #31: High level of user sessions on the database.

This query collects data from the database data dictionary regarding the amount of user sessions currently logged into the database.

Graph 31.0: Watch #31: High level of user sessions connected issues between both database servers.



Each and every user session takes up space and resources on the database and database server. Each session consumes server side memory as well as memory within the database SGA. If some or all of these sessions are "ghost"[40] sessions or sessions no longer connected but still consuming resources, it can be an early warning of a future problem, or will be consuming resources that could be better used elsewhere. A network issue on one or more clients can cause these types of issues, whereby the sessions are not freed and resources remain consumed. S.H.A.D.E was enabled to alert the administrator as it was deemed too high a risk to kill user sessions without analysing fully what is the cause of the issue. In early S.H.A.D.E testing, the system was set to kill older user sessions and keep only the new session, looping through sessions and killing the ones not needed. But these failed, as some nodes required more than one session and sometimes the older sessions were the ones which were live and needed. Accurate SQL tracing on each and every session to identify which ones are real and which ones are not or

[40] A ghost session occurs when a user gets disconnected and the server does not register it; they remain logged in on the server, but are in fact disconnected.

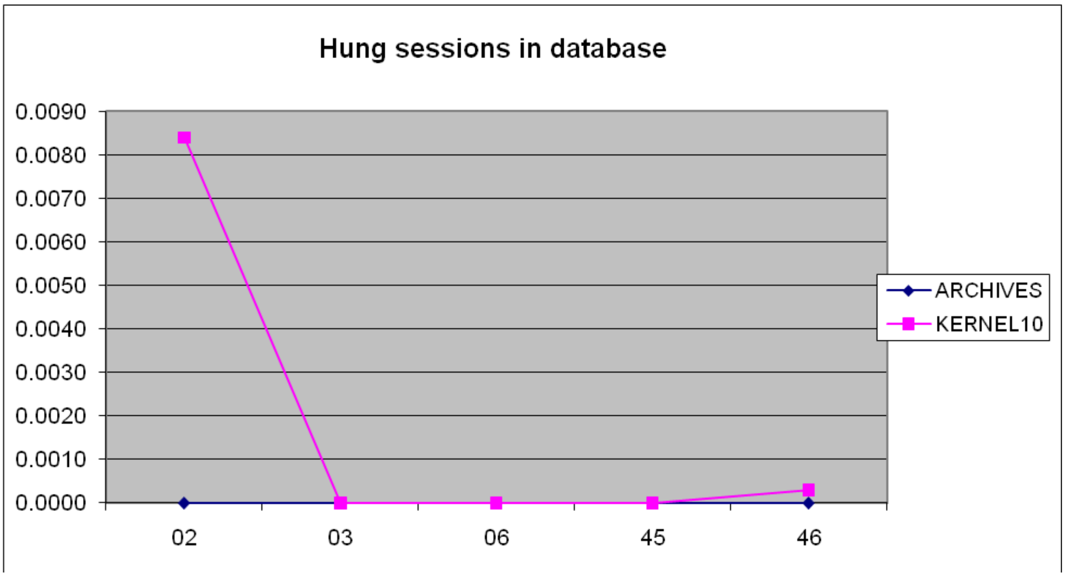
identify the client in question and forcing a re-boot which will clear all assigned user sessions and allow SQLnet [41] on the server to clean up the user sessions, was decided to be a more accurate and safe measure. Later versions of S.H.A.D.E could be set to remotely alter the client PCs in question in examples where, say, one node holds 30 + sessions because of a client network issue. Comparison between the two systems clearly displays the presence of the problem, with human intervention being the solution in these cases, with either client re-boots, kills or database server bounces/re-boots.

[41] SQLnet is Oracle's client/server middleware product that offers connection from client to the database.

Watch #32: Hung user sessions in the database

This query collects data from the database data dictionary regarding the amount of hung user sessions in the database potentially holding resources or consuming resources.

Graph 32.0: Watch #32: Hung user sessions issues between both database servers.



Hung user sessions typically occur after a user session is marked for killing or losing connection from the database server. They are killed, marked for kill and lose sessions, but the sessions remain on the database server. These sessions can cause deadlocks [42] (especially if killed to release objects already locked). In an ideal world, deadlocks woul dbe automatically cleaned up by SQLnet and the database when clients re-boot or time-out occurs. But sometimes they require intervention from an adminstrator to fix the issue and free the locks/resources. Oracle provides a server side tool that can be used to kill hung sessions called"orakill". Orakill must use the process spid (process address) and database's side reference to mark the user session to be killed at a process level within the Oracle process space. S.H.A.D.E queries the data dictionary for sessions that are flagged as hung, queries the revelnet information needed to execute an Orakill and

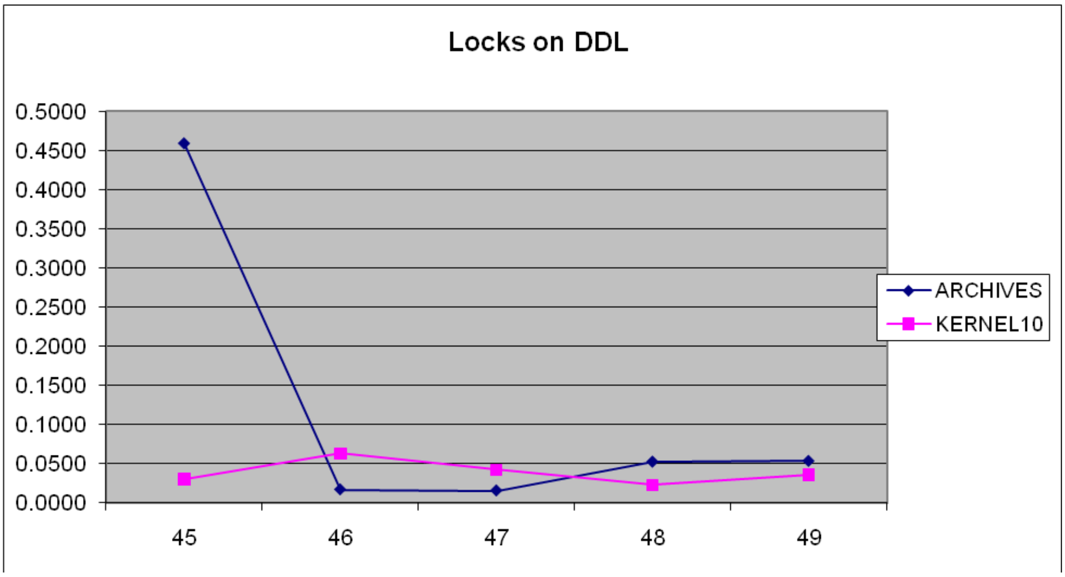
[42] A deadlock occurs when two or more threads are blocked, each waiting on a resource held by the other. When this happens, there is no possibility of the threads ever making progress unless some outside agent takes action to free the deadlock/resource(s).

runs the said request from the server, parsing in the correct process number and database reference. Executing Orakill with the wrong process could potentially kill a live “good” session or worse kill an Oracle process used to manage the database; hence the use of Orakill is a risk but is a powerful tool to free resources without re-starting the database to flush all sessions and executions to do so. From the graph, S.H.A.D.E managed to heal the issues as they arose, alerting the administrator as needed. The monitored system didn’t experience any hung session issues, again prompting for the need to inject faults with a fault-injection engine to all systems, to be compared more accurately in how they react to issues as they arise at the same time.

Watch #33: DLL[43] locks in the database

This query collects data from the database data dictionary regarding the amount of DLL locks that could prevent database residing code from being updated.

Graph 33.0: Watch #33: DDL lock issues between both database servers.



DDL locks are normal on a database; DDL deadlocks can cause issues as they lock objects and prevent other user sessions from using this object. DDL deadlocks can be particularly troublesome when a database object needs to be updated. An example would be putting a new database view or function live and not be able to, as the chance hangs and queues behind another session that won't free the resource without manual intervention. The DDL lock can be handled in much the same way as table deadlocks, as mentioned earlier, but DDL locks are detected differently. S.H.A.D.E monitors and alerts when the level rises above a set "healthy" defined level for the database in use (again this prompts knowledge of the instance the engine is running against and suggests a future feature of S.H.A.D.E, whereby the engine is able to build a history of optimal parameters and judge future failings based on this "health check" – basically setting down its baseline from knowledge it has built of the database, rather than manually

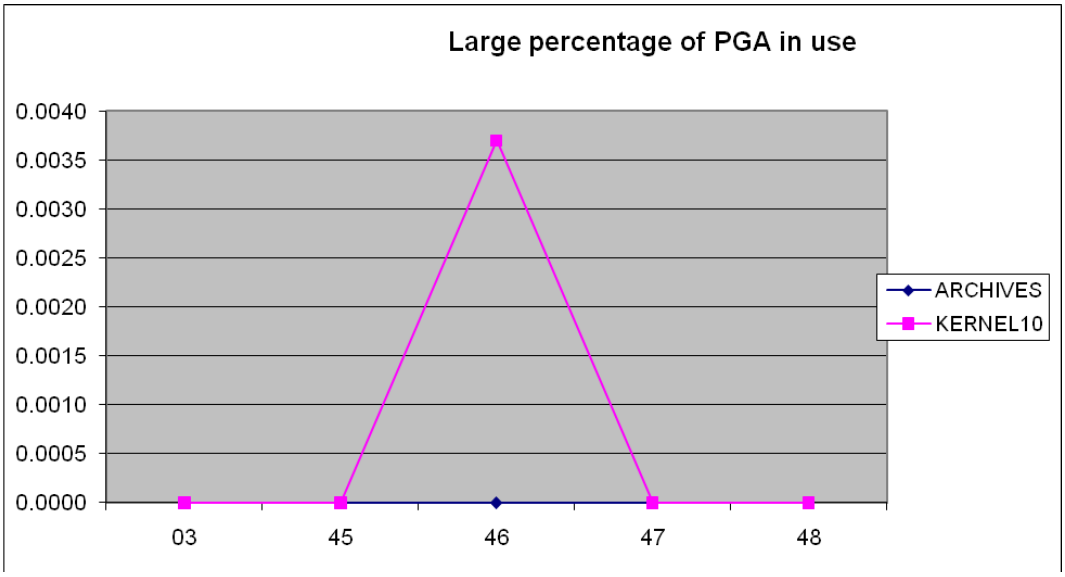
[43] Data Definition Language: computer language for defining data structures.

configuring the engine parameters over time). From the graph, it is illustrated that both systems experienced DDL locking issues, and both alerted the administrator of the errors, but DDL locks were only altered through human intervention, as full session analysis would be required to identify which DDL locks could be killed and which shouldn’t to allow DDL locks to be released.

Watch #34: Database PGA [44]usage

This query collects data from the database data dictionary regarding the amount of PGA memory assigned to and consumed by Oracle sessions.

Graph 34.0: Watch #34: Hung user sessions issues between both database servers.



The database's PGA is set at the database limit as an area of defined server/database memory that can be used and shared with connected users. If sessions require more space, they can take memory from the PGA pool for sorting data. If the PGA is set too small, users can get space within the database "on disk" temp files for sorting as needed but, as expected, this is a slower operation. If the PGA is set too large, space on the server could be better used for other processes or operations. Ideally, it should have sufficient space for all user operations without the need to constantly expand and contract to meet needs, while avoiding (where possible for normal transaction operations) the use of the slower temp disk files. From the graph, S.H.A.D.E was able to quickly repair the issue and re-size the PGA to better handle user transactions while avoiding execution failures due to space restrictions or slowing down operations because of disk rather than memory usage. The issue didn't arise again and there were no instances of it on the monitored-only database, as there were less user transactions and thus use of

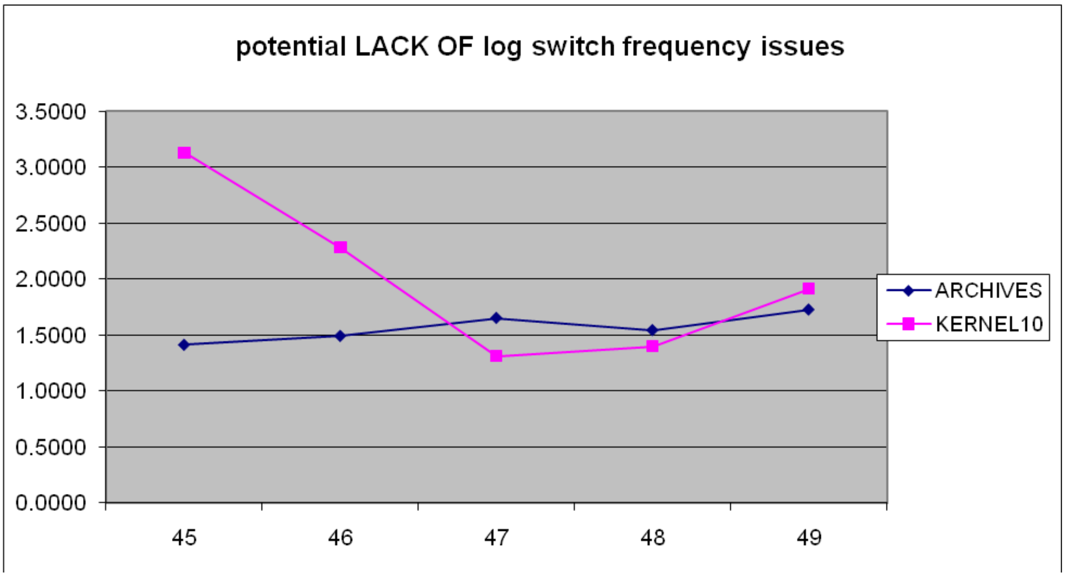
[44] Oracle session Private Ram Area. Individual PGA memory area is allocated each time a new user connects to the database.

the PGA area of memory. It would have been simulated by reducing the PGA size, but that would be similar to using a fault-injection procedure and not part of this experiment.

Watch #35: Potential lack of log switches on database.

This query collects data from the database data dictionary regarding the amount of log switches – if it is too low, it could affect the database's ability to recover.

Graph 35.0: Watch #35: Lack of log switches compared between both databases servers.

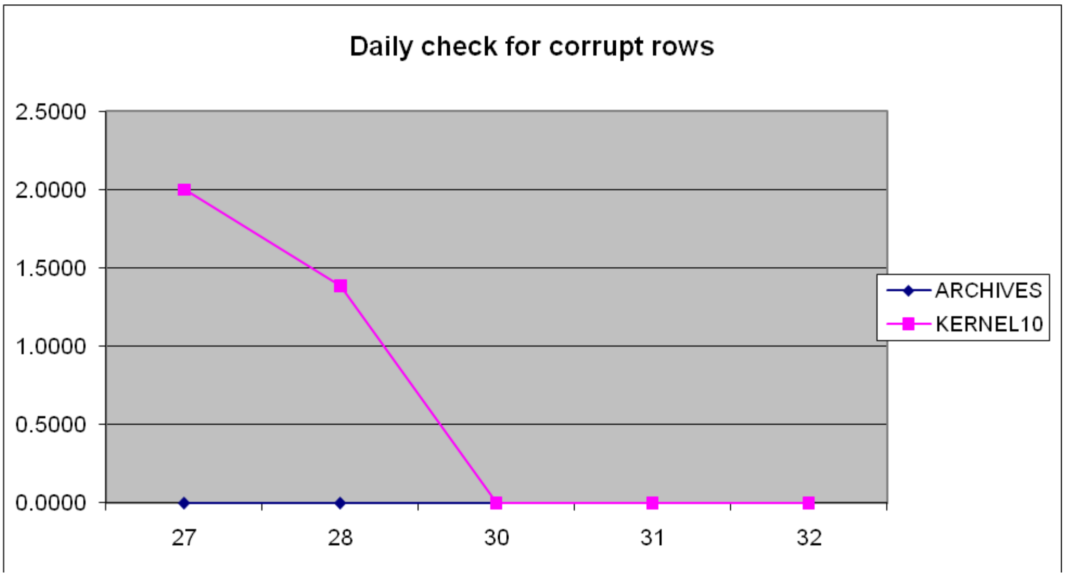


In much the same way as the watch that monitors and adjusts for too many log switches, this watch monitors and adjusts for too few. In this case, the database is not being hit from a performance point of view; on the contrary, performance would be improved with less switches and archive log arcs. But where this could cause an issue is if the system crashes; the bigger the log files, the more redo or data that could be lost from the log if lost. Ensuring more frequent switches means more frequent archive logs are produced and "hopefully" backed up to disk/tape. Again, redo switches can vary during the course of a database's operations, with more demand during heavy inserts/updates or long-running batch processes. S.H.A.D.E looks over a course of time and selects an average amount of switches. If this average is too low, it forces a switch and thus archives off a log for recovery. From the graph, it is illustrated how S.H.A.D.E managed to reduce the problem by forcing switches but, depending on operations, it was still flagged as an issue. After this, S.H.A.D.E continued to heal the problem and minimise the risk, but unless the redo's were reduced in size or other database parameters were altered to block changes or time to forces switches, the database would continue to report a “potential” issue - S.H.A.D.E. however, reduces the risk by forcing a switch. The monitored-only system continued to alert on switching issues but, to a different degree and with no human intervention, the alert level increased and decreased along with database update/insert levels.

<u>Watch #36: Check for corrupt database blocks in the database.</u>

This query collects data from the database data dictionary regarding the amount of corrupt database blocks in the storage files of the database.

Graph 36.0: Watch #36: Corrupt database block issues between both database servers.

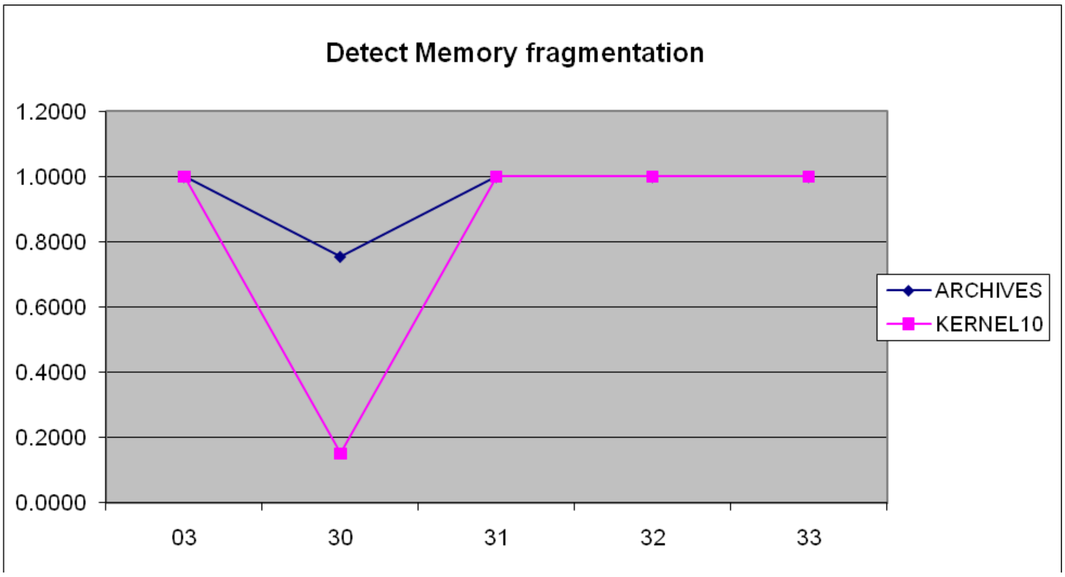


Corruptions can occur at the storage level of the database at anytime. These corruptions can be either hard or easy to fix, but all corruptions should be monitored and alerted on. These corruptions can be logical and thus only data issues, in which case the object can be re-built (index) or exported, dropped and re-built if a table. This would not be regarded as a "game-stopping" corruption and typically the user/administrator may only notice such a problem when the database reads the "infected" block or blocks of data and flags an issue with an error. Once identified, the administrator can manual repair the fault without too much downtime or effects on the user (depending on the size of the object, of course). If the corruption was caused by a hardware fault, then the entire system will need recovery or movement to an uncontaminated piece of hardware. The administrator would need to identify how bad an issue or fault had occurred and if the same hardware could be used. Often the mistake is to use the same hardware after a refresh without checking the root cause of the initial fault. S.H.A.D.E alerts the administrator of the possible corruption and immediately kicks off an export of the scheme's data to another disk. This means if the fault gets worse and the system totally crashes, or the error is only logical and limited to one object, the administrator has a head-start on the issue and S.H.A.D.E has not just alerted the problem, but has also taken the first steps in resolving the issue while protecting critical data. Within the graph, it is illustrated that S.H.A.D.E did, in fact, detect corruptions on the system (these were only simulated) and the engine successfully exported and alerted the administrator. But the data dictionary statistic was flushed when the system was re-started and the alert went away (not healed by S.H.A.D.E). The monitored-only system didn’t have any reported corrupt block issues and a simulation was required to test, as no heal operations were executing against this database.

Watch #37: Possible memory fragmentation on the database

This query collects data from the database data dictionary regarding the level of memory fragmentation with the database's SGA.

Graph 37.0: Watch #37: Memory fragmentation between both database servers.



S.H.A.D.E monitors the shared-pool[45] area of the database's SGA for memory fragmentation. If the shared-pool is under-sized or becomes fragmented, it can cause the user session to fail, as they will be unable to get enough database residing memory allocated to them to allow them to complete SQL execution on the database. When this happens, the client application will fail with a typical Oracle error and continue to fail until a) the shared-pool is increased or b) the shared-pool is flushed. Even flushing may not prevent the issue from occurring for a time period and the administrator will have to analyse the database and re-size it as required. Over-sizing it can help prevent the issue, but this wastes memory resources and can affect system performance overall as well as start-up and shut-down operations. S.H.A.D.E was given the ability to first attempt a memory flush of the shared-pool to effectively coalesce the memory space and attempt to get an area of free, defragmented space to all SQL to be parsed without error. If the issue continues, it increases the shared-pool and continues to monitor. If this still fails, the watch will attempt one more flush before parking the healing operation,

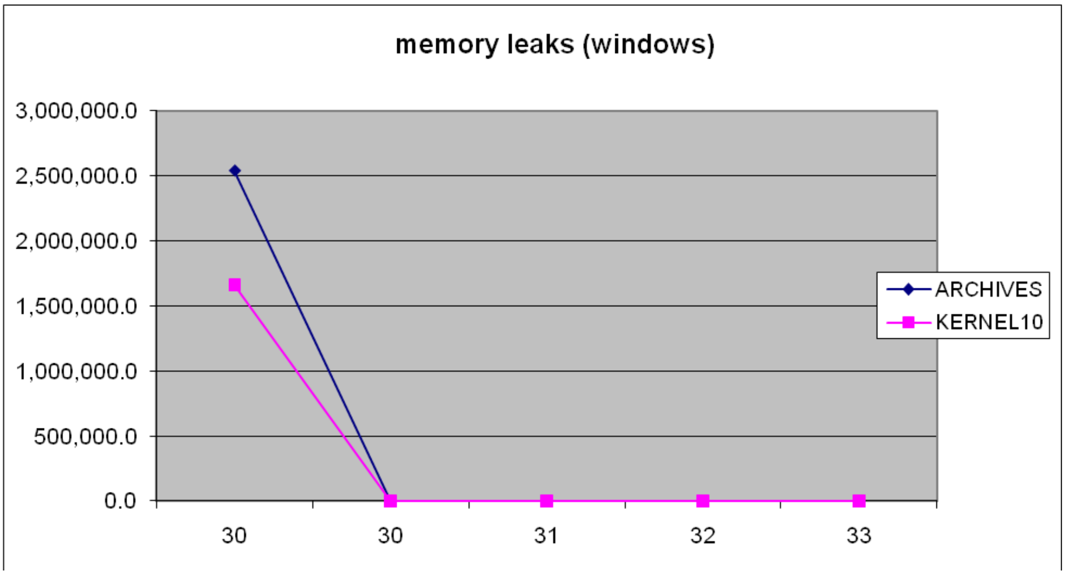
[45] Shared-pool is the cache of parsed and commonly-used SQL statements, and also the data-dictionary cache.

otherwise the continuous assignment of memory to the SGA without investigating the possible cause (in this case a badly written piece of SQL with looping memory consuming or leak) could take down the entire system. From the graph, it is illustrated that S.H.A.D.E initially reacted well and healed the issue, but the problem arose again and S.H.A.D.E failed in its healing operations. A poorly-tuned piece of SQL was to blame and the issue was repaired. S.H.A.D.E tried the first administrator actions and did alert the human element of the system to trigger further investigation. The monitored-only system experienced a continuously steady level of fragmentation during the period, not causing a fault, but any further consumption could have caused a failure or session fault.

Watch #38: Windows (operating system) memory leaks

This query collects data from the server operating system performance levels to detect any occurrences of operating system memory leaks that could potentially halt or crash the system.

Graph 38.0: Watch #38: Windows memory leak levels between both database servers.



From the graph, it is illustrated that both systems only experienced one occurrence of memory leaks (simulated) during week 30. This was a test to see the effectiveness of the S.H.A.D.E monitoring only. A possible heal with a windows memory leak would be to kill possible leaking processes detected within the resource kit "poolmon". S.H.A.D.E only monitored both systems and reported the issues; the problem went away as it was only simulated and didn't occur for real after the simulated executions.

4.7 Improve phase

Improvements are only visible with a percentage of heal operations presented here. The events in which the engine failed to succeed are clearly defined and explained within each watch element. The elements that are listed as failing to improve or healing sufficiently would require further development in later builds of the engine. More time and refinement could increase their rate of success. If they continue to have negative success rates, they would need to be removed completely or set to monitor-only, as there is no benefit having code firing and failing with limited chance of success.

4.8 Control phase

During the initial design of the engine, improvements were necessary to not just see how the faults were detected, but also how the system reacted to such detections. Initial fixes were tweaked and enhanced during the control phase, to ensure the engine operated with the smallest footprint as well as with the greatest chance of success. Each and every watch was custom-defined and altered to execute at pre-defined polling times and with custom fixes (where possible). Any heals that were deemed as potential risks through either insufficient abilities or risk of change, were constructed as alert triggers only. This meant all defined watches made it into the final build of the engine, but those that caused issues or fired with a high level of risk, were altered within the engines parameters to just monitor and not make changes.

4.9 Summary

This chapter presented the results of the experimental activities of the S.H.A.D.E engine. The defined failings of the engine have resulted in defining possible improvements in future builds.

These possible future improvements are:

a) Improved and more advanced integration with hardware elements of the system. The healing system needs to be able to not just alter the database and operating elements of the system, but must also be able to alter hardware components as well as alert on potential issues. If the system could proactively move data from drives it has flagged as being at risk, rather than alert, the value of such options would been greatly increased.

b) A more advanced selection of fix options to hand. If the system had an increased number of fixes, it could fire for each watch, it could reduce the human interaction element of the system, as there could potentially be less fixes parked through errors and less alerts for the administrator to react to (as well as fixes to carry out when S.H.A.D.E is unable to fix issues).

c) More advanced error logging checks. To inject the system with a collection of experiences in what error logs are critical. Certain alerts and corruptions would require a DBA to shut down and inspect a system immediately. S.H.A.D.E could be given more knowledge and fix options for such events and could also alert when detected by executing the first steps a human would undertake before the human element would get a chance to interact and diagnose.

d) No hard-coded parameters for disk drives but the ability to detect connected volumes and what the operational tasks are within the system. It would include the ability to check volume sizes, free space, fragmentation levels and what is running on each volume along with disk IO levels. It would also include more "disk intelligence" rather than the limited disk options in the present build. A large amount of system performance issues arise purely because of ill-balanced disk IO. S.H.A.D.E could have the ability to move files in much the same way as SAN hardware can move data blocks to lesser-used volumes from volumes that are over-utilised.

e) Reduce platform limitations that are present in current build. It would be advantageous to include knowledge and options for other operating systems such as UNIX and Linux to allow S.H.A.D.E to be used not just on the Oracle/Windows platform but to include options for other operating systems as well as “possible” hardware elements (Clusters, SAN, hot spares etc) and to allow these options to be active and present or de-activated through the user interface easily without constant parameters changes within the database repository tables.

f) Constant monitoring and alerting of poorly-written and slow SQL on the database. A very common cause of poor performance and thus poor health of a database is the SQL code running against the database. Many 3rd party monitoring applications have options to capture and tune SQL as needed. S.H.A.D.E should, therefore, have the ability to constantly detect poor SQL if and when it “ever” executes, so even if a poor piece of SQL rarely runs, its time, condition and all other details would be detected, captured and presented to the administrator.

Chapter 5: Discussion.

5.1 Introduction

The success of self-healing within future system designs can aid both in the overall manageability of systems as well as reducing cost and simplifying the task of administration altogether (Chan-Bin Ling, 2004). Both of these elements are high on the objectives of modern IT managers' challenges for 2007 and beyond (BMC software, 2006). The real challenge, however, is how to provide an effective level of service while operating under an environment of continued cost reduction and shrinking budgets (Ryan et al., 2008). Budgets are shrinking even greater as we enter unchartered waters in 2009 and beyond, with world credit problems and a global recession, effectively meaning IT managers are now expected to deliver greater automation and maximise efficiency while operating under tighter budgets. The IT customers have no knowledge of these restrictions, but still expect their systems to operate 24/7 (Armstrong, 2005). How to achieve this without adding to overall costs requires a different system management approach. One approach to reduce costs is to reduce human intervention while allowing the human to effectively manage more by reducing complexity. This is not just a simple suggestion to reduce jobs, by removing one person and shifting the workload onto someone else. It helps the human operator by allowing them to do more with less, by reducing the amount of problems and thus complexity they need to deal with (Ryan et al., 2008).

This challenge becomes even more prevalent in the next few years and beyond, as operational costs are being cut to maximise capital and reduce spending during harsh economic times. Never since the introduction of the computer system into industry and the subsequent birth of the IT manager has the cost of managing computer systems been as scrutinised, as companies are challenged to cut costs. "IT delivery organisations are still tasked with meeting acceptable service levels while dealing with massive storage growth, increasing complexity in the data centre, and shrinking budgets—and potentially reduced staff." (EMC, 2009). The challenge to adapt and find a better way is something all administrators and managers must adopt in order to continue moving forward, while delivering effective levels of service to the business.

5.2 Overheads of self-healing designs

It would be naive to assume any changes to a system could be conducted without any risk to its operational state, that a self-healing engine could operate without any overheads. Overheads, however, are not just limited to its effect on system performance, but the overhead of risk must also be factored into any system change, even a new system engine such as the self-healing one defined in this paper. Any new element within a system, whether automated or human/manual, introduces a risk of overhead. Any change implemented (if not monitored) runs the risk of affecting the system state in either a positive or negative fashion. The key is to reduce this overhead by monitoring the effects of any change. If human administrators made changes without monitoring the effects of the changes they made, not only are they running the risk of affecting the system's performance, but also are not gauging their success if any, in reacting to what they felt needed changing in the first place.

Also, overheads of executing the self-healing engine may also vary from system to system. It could run on one server with adequate resources and have minimal or any effect on system operations in a negative manner. Running on a lower spec machine could greatly affect system operations. Hence, this is why the design of the S.H.A.D.E engine was to be made to the lower system footprint (within reason) and the selected heals/alerts were chosen as those with greater level of success versus greatest level of risk.

5.3 Design needs

The only way a self-healing or true autonomic system can be effectively constructed is when each component in a system can act and communicate as a whole. Each component must contribute and aid other components, whether said components are hardware or software. It is not sufficient for an operating system to attempt a healing operation that causes it to take resources from the system that can affect other elements of the overall system. Each component requires system side communication and awareness (Ryan, et al., 2008), not only of its own actions and effects but the actions of other components on the entire system (IBM, 2001). The only way for this to be achieved is through standards and system design co-operation, essentially to have system designers working with common goals, not just concerned with new features that may sell their own products. Unfortunately, the larger computer vendors such as Oracle and Microsoft work under business models that rely on growth and sales, with sales relying on new versions and updates to older products (as well as support that is required because of potential issues after release).

5.4 Where the S.H.A.D.E engine failed

As discussed, the only way a self-healing or true autonomic system can be effectively constructed is when each component in a system can act and communicate as a whole. S.H.A.D.E chiefly dealt with elements of the RDMS (database) system along with components of the operating system. Incorporating hardware elements such S.M.A.R.T., although incorporated into the engine, only acted with alerting capabilities and was not developed to carry out more effective tasks such as moving data from “potentially” future failing or contaminated disks. S.H.A.D.E would need to be more aware of all system components to be truly autonomic in its actions and capabilities. As stated in the previous chapter, there was insufficient time to develop all of the watch elements to have healing abilities because some were deemed too risky in execution, potentially risking system health on their own. Thus S.H.A.D.E failed in being truly autonomic but did succeed in actually healing key identified system issues when provided with the “abilities” to do so. Chapter 4, and in particular section 4.9, clearly outlines a list of future improvements which were compiled from the successes and failures of the S.H.A.D.E engine and these improvements would aid in making the engine more autonomic in future builds, with more abilities, options and development time.

5.5 Where the S.H.A.D.E engine succeeded

Although S.H.A.D.E didn’t succeed in being truly autonomic, this thesis did effectively identify the need for such, and how autonomic abilities could be introduced into the engine during future builds and work. S.H.A.D.E did succeed in changing the systems running state and efficiency by eliminating and repairing key issues that arose naturally through normal “everyday” usage. As discussed, a fault-injection engine would have allowed more effective testing and compare the S.H.A.D.E-enabled system against a system running without the engine. However, “any” positive change to a system’s running state is a step towards true automation and the evolution of a true autonomic system and with that in mind, S.H.A.D.E has proven the theory outlined in the literary review chapter. It has proven that a system can be designed with self-healing in mind and we could one day, possibly be operating less complex and more efficient systems in the future.

5.6 Summary

As discussed, the challenges of the modern IT manager and system administrator are a constantly changing field. IT, by its nature, requires funding and upgrades to allow it to evolve and progress, but recent modern times have increased not only the demand on system “up-time”, but also decreased the ability to move forward with ever-shrinking budgets and the need to cut costs. The cloud appears to offer one level of reducing cost, but which company do you hire to hold your mission critical data and how effective are they are keeping systems on-line and operational?

Autonomic and self-healing systems of the future could only help to enhance the cloud model or any system model for that matter. If true autonomic computing became a reality, data centres, whether in the cloud or in the computer room, could operate with less human intervention which will in itself reduce costs as well as potentially increasing their effectiveness and efficiency in both “up-time” and operational performance. S.H.A.D.E has demonstrated, as part of this thesis, how self-healing and self-management is a step in the right direction. Now we must rely on the system builders such as Microsoft, Google, IBM and Oracle (amongst others) to work together to define standards for a true autonomic age of computing.

Chapter 6: Conclusions and future work.

6.1 Introduction

“Self-healing systems are expected to respond to problems that arise in their environments with minimal human intervention” (Griffith et al., 2007). This statement covers a lot in very few words regarding what is expected of a self-healing system and also what factor may deem it a success or not. This factor is the reduction of human intervention. The self-healing system is measured on the reduction of the human element in the system’s operations. This reduction of the human interaction was a clear goal of the S.H.A.D.E engine and design, where a selection of “everyday” possible issues were identified and designed with subsequent healing abilities (where possible).

Automation at any level can enhance the operations of a system, so long as automation does not increase risk to the system or reduce the quality of its operations. Reducing the human administrators’ manual operations will obviously free up their time and thus decrease cost. But also automating repairs will help a system repair itself quicker and reduce the risk of the system performance degrading or even crashing. Freeing up the human element can help people archive more with less and free up their time to deal with more systems.

This could also open up another argument regarding loss of skill. If the human administrators are not exposed to issues more regularly, they may run the risk of losing the knowledge on how to deal with a problem once the automation of self-healing fails. But this same issue could also occur with newer system versions, whereby the vendor has reduced complexity and filtered features away from the administrators, in much the same way as Oracle has done in its 10g RDBMS (Kumar, 2006).

The cost of doing business has never been so high on companies’ agendas. Although the use of IT systems can effectively help business to reduce cost through automation and a reduced workforce, the cost of IT is also not without its own challenges. Self-healing designs will aid systems to become more self-sufficient. Through the introduction of standards and mature initiatives such as

IBM’s autonomic computing, computer systems could one day be completely self-sufficient.

6.2 Future Work

If the human operator was removed completely from the equation, the problem focus would shift to how the human element might be replaced with a computer solution. This is a two-fold problem; on one hand, there is a problem of finding a solution to the issue of complexity, by either helping or replacing the human operator. On the other hand, humans offer elements that today’s A.I. [46]/expert systems have no hope of matching; “humans can handle unknown situations and learn from their experiences” (Hermannet al., 2005). Humans can adapt and change their approach based on knowledge and results. As shown, the design of a self-healing engine requires significant knowledge of the platform it will operate under. The developer needs to be aware of numerous elements within the system and have a clear, tested understanding regarding the success level of the self-healing elements of the engine. Otherwise, the application runs the risk of causing more issues than it could fix by making changes that would not succeed and affecting components of the system.

Clear and concise standards developed by the key players in the industry are needed if self-healing and thus self-managing systems are ever to become a reality. The autonomic computer initiative’s current definitions are sufficiently well defined.. The four functional areas represent a useful categorisation, but the differences between them are not clear. As mentioned in this paper, distinguishing between a faulty system and a system that's not operational optimally isn’t easy. While a faulty system would be subject to self-healing, a sub-optimal system should self-optimise. It really depends on how one classes a fault in the same respect as how one classes as a “heal”. What distinguishes self-healing from self-optimisation is as much of a problem that requires definition as does what is a

[46] A.I.: Artificial Intelligence, a method whereby computer software attempts to simulate and replicate human intelligence.

fault? These questions are the building blocks for standards that will define self-managing systems. Other concepts that exist are self-adaptation and self-organisation. Although we might intuitively regard them as being more general than the concept of self-management, finding a clear definition of these terms is extremely difficult, if not impossible, because the concepts behind them are still only partially understood (Hermann, et al., 2005).

One obvious future development for the engine would need to be the ability to react and diagnose "unknown" situations and errors in much the same manner as a human engineer who is presented with a problem he has never seen before. The S.H.A.D.E engine would thus not just need the ability to diagnose and identify the issue but would also have to be able to draw on previous experiences and attempt to change the system (improving its condition). It would be building real artificial intelligence into the engine and drawing on experience in much the same way as an automatic expert system. Ideally, the experience could be extracted from companies' help desk systems (if standardized), drawing from an already potentially massive amount of data concerned with possible errors and repairs. This is where standards need to be designed and adopted so true autonomic systems are not just moving forward in their daily operations but are (like their human counterparts) as able to draw from a wealth of previous experience, whether the same system has experienced it themselves. This could be compared to a human engineer who finds himself without the knowledge or experience and calling a colleague or searching the internet/knowledge boards for answers. Since help desk calls are stored in databases, if they were stored in standard formats and the systems were open to interrogation, then the data would be of value to everyone and not just the companies housing the help desk information.

6.3 Conclusions

As stated, an IBM objective for its Autonomic Computing project is to free people from having to think how computers continue to operate. As operators and end-users, our time would be better spent using computers as tools, rather than needing to know how they work, or how to fix them once they stop working.

The term autonomic computing introduces a new interesting and promising research field. However, this area is not well explored in depth, not yet delineated, and final targets are not well identified. Researchers have different and contradictory opinions and ideas about autonomic computing and terminology is often used in a wrong manner (Tosi, 2004). Self-healing can be regarded a single piece of the overall puzzle, which is in fact the overall goal of self-management or autonomic computing.

To have a fully operational self-healing system, each and every component within the system would need to work in full co-operation. This would require hardware and software to be aware of each other's existence and configurations and be able to control each other's actions. The software would need to determine if and when hardware may be at fault, identify that faulted unit and have sufficient resources to continue without it. Server farms of today can manage this on a simplified level, whereby the transaction's load is moved and balanced onto healthy systems when a system faults, picking up the slack and even bringing passive systems in active operations to step in for the failed unit. This principle would need to work in a similar fashion within a self-healing system. A disk fails and a spare disk automatically becomes active, while the old disk is removed and replaced with a passive spare disk. A CPU or memory chip faults, a passive chip comes into operation and the old chip is flagged as faulted. Until advanced robotics and A.I. are commonplace, however, we will still need the human engineer to remove and replace the contaminated part.

For self-healing and autonomic computing to progress, it will need the support of the major hardware and software developers. As suggested, they need to work together in developing standards and not just using these types of elements as unique features that can be used to sell their own products. Future studies in the field of self-healing could work at defining a baseline for these standards and carrying out some of the foundation work for moving towards the design of self-healing systems or systems with self-healing features.

If systems continue to become more complex and rely more on the human element to intervene when problems arise, it will only continue to add to the cost of ownership and increase the dependency on the human element and their skill-sets.

Chapter 7: Appendices.

7.1 Bibliography

1. Adya, Atul. Paramvir Bahl, Ranveer Chandra & Lili Qiu.Architecture and Techniques for Diagnosing Faults in IEEE 802.11. Infrastructure Networks. Tenth Annual International Conference on Mobile Computing and Networking MobiCom 2004.

2. Ariolic Software, Active Smart. S.M.A.R.T. Technology. 2007 http://www.ariolic.com/activesmart/smart-technology.html Accessed 22nd August 2007.

3. Armstrong, P. 2005, Swing into Business Service Management: Seven strategies for enabling IT to activate the business. BMC Software. p. 5-7.

4. Ballmer S. 2006, "A New Era of Business Productivity and Innovation. Microsoft Corporation. http://1010wins.com/pages/136856.php?contentType=4&contentId=25047 Accessed 16th August 2007.

5. Baum, David. Step Into Windows. "Oracle technology on the Windows platform pumps up performance and reliability". Oracle Magazine May/June 2007.

6. Beeler, Diane. Ignacio Rodriguez Database Performance Made Easy: BMC software 2002.

7. BeyondThrust, A Process-based Approach to Protecting Privileged Accounts & Meeting Regulatory Compliance. BeyondThrust. 2009.

8. Bloor, Robin. The Extraordinary Failure of Anti-Virus Technology**:** Whitelisting Succeeds Where AV Has Failed. Hurwitz & Associates. 2007.

9. BMC software 2006, A Market Analysis: Performance Management: New "Hybrids" Combine Agent and Agent-less Technology, Deliver Best of Both Worlds From a Survey of Technology Decision-Makers. Ziff David Media U.S. p.7-9.

10. Borning, Alan, “Computer System Reliability and Nuclear War”. Association For Computer Machinery. 1987.

11. Brown Aaron B, Charlie Redlin, “Measuring the Effectiveness of Self-Healing Autonomic Systems”. Second International Conference on Autonomic Computing (ICAC'05) By Publication Date: June 2005.

12. Brown, A.B., J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, M. Peterson Yost 2004, Benchmarking Autonomic Capabilities: Promises and Pitfalls. IBM Laborites U.S. p. 2.

13. Brzezinski, Jerzy, Michal Szychowiak. Poznan SELF-STABILIZATION IN DISTRIBUTED SYSTEMS – A Short Survey University of Technology 2000.

14. Burleson, Donald K. Cost Control: Inside the Oracle Optimizer. 2001 http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/burleson_cbo_pt1.html
Accessed 10th February 2008.

15. Burniece, Tom. DISK AEROBICS™ and MAID (Massive Array of Idle Disks) A Match Made in Heaven. Copan systems. 2005.

16. Chan-Bin Ling, B. 2004, Self-Healing. http://www.usenix.org/events/nsdi04/tech/full_papers/ling/ling_html/node17.html
Accessed 10th August 2007.

17. Cruz, Acacio 2009. Official Gmail Blog. http://gmailblog.blogspot.com/
Accessed 25th February 2009.

18. Czap, H., R. Unland, C. Branki, Self-Organization and Autonomic Informatics, IOS Press, 2005 pp 267.

19. Daly, Mark A. Task Load And Automation Use In An Uncertain Environment. Department Of The Air Force Air University Thesis, Captain, USAF. 2002.

20. Dashofy, E.M., A. Van der Hoek, R. N. Taylor 2002, Towards architecture-based self-healing systems. University of California. p 2.

21. Diao, Yixin, Research staff member, Research Lab: Watson Research Center (Hawthorne).
http://domino.research.ibm.com/comm/research_people.nsf/pages/diao.index.html 2006
Accessed 5th January 2008.

22. Diao, Yixin, Joseph L. Hellerstein, Gail Kaiser, Sujay Parekh, Dan Phung "IBM Research Report Self-Managing Systems: A Control Theory Foundation" RC23374 (W0410-080), 2004 pp 8.

23. Dijkstra, E.W., "Self-Stabilizing Systems in Spite of Distributed Control," Comm. ACM, vol. 17, no. 11, 1974, pp. 643-644.

24. Diskeeper Corporation. Background multitasking: A technology report for IT professionals. 2006.

25. Diskeeper Corporation. Hands-on with diskeeper Whitepaper 2007.

26. Dolev, Shlomi and Reuven Yagel. Ben-Gurion SOSP'05 \ SIGOPS. Doctoral Workshop, University of the Negev, Beer-Sheva, Israel. Oct 2005.

27. Dunne, S., 2007.Microsoft updates windows without users consent.
http://windowssecrets.com/2007/09/13/01-Microsoft-updates-Windows-without-users-consent
Accessed 2nd January 2009.

28. Executive Software International. Is daily fragmenting needed in today's environment? Whitepaper. 2005.

29. Foley, M.J. 2000 Bugfest! Win2000 has 63,000 'defects'.
http://news.zdnet.co.uk/software/0,1000000121,2076967,00.htm
Accessed 1st August 2007.

30. Food and Drug Administration, 2002. General Principles of Software Validation; Final Guidance for Industry and FDA Staff. U.S. Department Of Health and Human Services p 8-9.

31. Fontana, John. Microsoft reissues patch, encourages XP SP2, SP3 re-installs., Network World. http://www.networkworld.com/news/2008/061908-microsoft-reissues-patch.html?t51hb&netht=mr_062308&nladname=062308dailynewsamal Accessed June 16th 2008.

32. Frankenhaeuser, Marianne. To Err is Human: Nuclear War by Mistake?*. Stockholm Sweden, Psychology Division, Karolinska Institute, Stockholm (1997).

33. Ganek. A.G. and T.A. Corbi, "The Dawning of the Autonomic Computing Era," IBM Systems J., vol. 42, no. 1, 2003, pp. 5-18.

34. Ganesh, Amit. Sushil Kumar The Self-Managing Database: Proactive Space & Schema Object Management with Oracle Database 10g Release 2. Oracle Corporation. 2005.

35. Gao, J., G. Kar, and P. Kermani. Approaches to building self-healing systems using dependency analysis. IEEE/IFIP Network Operations and Management Symposium 2004.

36. George, Selvin, David Evans, Steven Marchette, A Biological Programming Model for Self-Healing. University of Virginia. 2003.

37. Gillen, Al, Randy Perry, Bob O'Donnell and Brett Waldman. Analysis of the Business Value of Windows Vista. IDC 2006.

38. Gongloor, Prabhaker, Cecilia Gervasio, & Sushil Kumar Oracle Database 10g: Intelligent Self-Management Infrastructure. Oracle Corporation. 2006.

39. Google, 2009. Google Apps – Gmail Incident Report February 24, 2009. P 1.

40. Griffith, R., R. Virmani and G. Kaiser 2007, The Role of Reliability, Availability and Serviceability (RAS) Models in the Design and Evaluation of Self-Healing Systems. p. 1-3. Columbia University U.S. p. 1-3.

41. Griffith, Rean, Gail Kaiser. "A runtime adaptation framework for native C and bytecode applications." IEEE 2006.

42. Griffith, Rean, Gail Kaiser. "Adding Self-healing capabilities to the Common Language Runtime" Columbia University 2006.

43. Herrmann, K., G. Muhl and K. Geihs. 2005, Self Management: The solution to complexity or just another problem? IEEE distributed systems online p. 5-6.

44. Hewlett-Packard .HP openview Self-healing Services Data sheet. 2005.

45. Hunt, Preston. Dylan Larson.Addressing IT Challenges with Self-Healing Technology. Intel Corporation 2003.

46. International Business Machines Corporation, Autonomic Computing Concepts, IBM press 2001.

47. International Business Machines Corporation (2006a), Autonomic Computing White Paper An architectural blueprint for autonomic computing fourth edition. International Business Machines U.S. p. 3, 6-7, 18-20.

48. International Business Machines Corporation 2001, "Autonomic Computing: IBM's Perspective on the state of Information Technology". International Business Machines U.S. p. 1-2, 21-23, 25-27,30-32.

49. International Business Machines research. Deep Blue Technology:
http://www.research.ibm.com/know/blue.html
Accessed March 22nd 2008.

50. International Business Machines Corporation (2008). When the going gets tough, the tough need insight. Performance management in the weak economy. P 8.

51. International Business Machines: IBM Internet Security systems. X-Force 2008 Trend and Risk report. P. 6, 30.

52. Jonssonm, Håkan: Algorithm Lecture with course "Object-Oriented Design" Luleå University of Technology: 2006.

53. Jupiter Research, RETAIL WEB SITE PERFORMANCE: Consumer Reaction to a Poor Online Shopping Experience: 2006.

54. Kandogan, Eser, Christopher S. Campbell, Peter Khooshabeh, John Bailey, and Paul P. Maglio, Policy-based Management of an E-commerce Business Simulation: An Experimental Study. University of California. 2006.

55. Kelton research 2007, "65 Percent of Americans Spend More Time with Their Computer than Their Spouse". Support Soft. Research survey.

56. Kephart, J. O. and D. M. Chess 2003, The vision of autonomic computing. IBM Thomas J.Watson Research Center p. 9-10.

57. Kephart, J. O. "Research Challenges of Autonomic Computing" IBM Thomas J. Watson Research Center 2005.

58. Kessler, Michael. Maintaining Windows 2000 Peak Performance Through Defragmentation. Microsoft Corporation. http://www.microsoft.com/technet/prodtechnol/windows2000serv/maintain/optimize/w2kexec.mspx
Accessed 9th September 2008.

59. Kumar, Sushil. Oracle Database 10g Release 2: The Self-Managing Database. Oracle Corporation. 2006.

60. Lahiri, Tirthankar, Arvind Nithrkashyap The Self-Managing Database: Automatic Shared Memory Management with Oracle Database 10g Release 2. Oracle Corporation. 2005.

61. Lamport, Leslie, “My Writings/ Solved Problems, Unsolved Problems and NonProblems in Concurrency”, 1984 section 57.
http://research.microsoft.com/users/lamport/
Accessed 10th September 2008.

62. Laliberte, Bob. Emc, 2009. An Innovative Approach to increasing Operational efficiencies. P3.

63. Ling, Chan-Bin Self-Healing. Benjamin. 2004.
http://www.usenix.org/events/nsdi04/tech/full_papers/ling/ling_html/node17.html
Accessed 30th September 2008.

64. Locasto, Michael E. Micro-speculation, Micro-sandboxing, and Self-Correcting Assertions: Support for Self-Healing Software and Application Communities. Department of Computer Science Columbia University. 2005.

65. Marti, Sergio, T.J. Giuli, Kevin Lai, and Mary Baker.Mitigating Routing Misbehavior in Mobile Ad Hoc Networks. Stanford University. 2000.

McKendrick, Joe. Research Analyst The Rise of the Renaissance Data Professional – 2007 and Beyond. By. Conducted by Unisphere Research for the Independent Oracle Users Group (IOUG). 2007.

67. Mazzotta, Mary Y. *Nutrition and wound healing*. Journal of the American PodiatricMedical Association. Volume 84, Number 9, p. 456–62. September 1994.

68. Middlemiss, Jim. Fragmentation: A $50 Billion A Year Problem (Citibank and Wells Fargo achieve performance gains and cost savings through defragmentation.). Miller Freeman Inc 2000.

69. Microsoft Vista Techcenter Reducing Support Costs with Windows Vista. 2006.
http://technet.microsoft.com/en-us/windowsvista/aa906019.aspx
Accessed 3rd February 2008.

70. Microsoft. Windows Vista customer solution case study. 2007.

71. Mushkatin, Victor. Application Health Monitoring and Modeling, AVIcode, Intercept Studio, 2006.

72. Murray, S. A. 1997. "Effects of Operator Alertness on Human-Machine Interaction and Supervisory Control Performance." Doctoral Dissertation, University of Wisconsin-Madison.

73. Opsware, Selecting a Flexible, Custom Platform to Automate Your Data Center Management. 2006.

74. Olofson, Carl W. Oracle Database 10g Standard Edition One: Meeting the Needs of Small and Medium-Sized Businesses. IDC 2005.

75. Oracle Corporation, Oracle Database 10g, Enterprise Edition, Oracle Datasheet 2006. pp 5.

76. Oracle corporation Oracle Enterprise Manager 10g: Oracle Diagnostic pack. 2005.

77. Pinheiro, Eduardo, Wolf-Dietrich Weber and Luiz Andr´e Barroso Failure Trends in a Large Disk Drive Population. Google Inc. Feb 2007.

78. Randall, D. “Mystery virus hits 15 million PCs around the world”. 2009.
http://www.independent.co.uk/life-style/gadgets-and-tech/news/mystery-virus-hits-15-million-pcs-around-the-world-1515314.html
Accessed 25th January 2009.

79. Ryan, S, Quinn-Whelton, N, McCarthy, M “SELF-HEALING COMPUTER SYSTEMS: THEIR ROLE IN FUTURE SYSTEM DESIGNS”. IMC 25, 2008.

80. Ryan, Sean & Ryan, Lisa. P.O.P.A: Pocket Oracle PDA admin. BSc project W.I.T. 2005.

81. Samsung, S.M.A.R.T. Introduction Self-Monitoring Analysis and Reporting Technology, Samsung. 2007. http://www.samsung.com/Products/HardDiskDrive/whitepapers/WhitePaper_07.htm
Accessed 15th April 2009.

82. Sarter, N.B. , D. D. Woods, and C.E. Billings, "AUTOMATION SURPRISES**",** Cognitive Systems Engineering Laboratory, The Ohio State University, 1997.

83. Scalability Experts, Inc. Microsoft® SQL Server 2005: Changing the Paradigm (SQL Server 2005 Public Beta Edition). Scalability Experts, Inc.

84. Seagate Technology "Get S.M.A.R.T. for Reliability." 1999.

85. Shaw, Mary. Sufficient Correctness and Homeostasis in Open Resource Coalitions: How Much Can You Trust Your Software System? 2000.

86. Shaw, M. 2002, "Self-Healing": Softening Precision to Avoid Brittleness Position paper for WOSS '02: Workshop on Self-Healing Systems. Institute for Software Research, International School of Computer Science. Carnegie Mellon University. p. 2-3.

87. Stanek, W.R., Introducing Microsoft Windows Vista. p 206 Microsoft press U.S. 2006.

88. Stauffer, Andr´e, Daniel Mange, Gianluca Tempesti, and Christof Teuscher A Self-Repairing and Self-Healing Electronic Watch: The BioWatch. Logic Systems Laboratory, Swiss Federal Institute of Technology, 2001.

89. Sterritt, R. M. Parashar, H. Tianfield, R. Unland 2005, A concise introduction to autonomic computing. Engineering Informatics. p 5.

90. Stork, Dr. David G. The end of an era, the beginning of another? HAL, Deep Blue and Kasparov : Chief Scientist at Ricoh Silicon Valley 1998. http://www.research.ibm.com/deepblue/learn/html/e.8.1.html
Accessed 10th December 2008.

91. Stojanovic, L. et.al. The role of ontologies in autonomic computing systems. IBM Systems Journal, 43(3), 2004.

92. Sudhir, B. Kumar Reddy, MTech(IT) Kanwal Rekhi School of Information Technology. Control and Coordination of Software Adaptation for Automization. Indian Institute of Technology – Bombay. 2004.

93. Sun Microsystems, Predictive Self-healing in the Solaris 10 Operating system: Delivering relentless availability. 2004.

94. Sweeny, Tim. The next Mainstream Programming language: A game Developers perspective. Epic Games. 2006.

95. Tesauro, G., D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart and S. R. White 2004 A Multi-Agent Systems Approach to Autonomic Computing. IBM T.J. Watson Research Center. U.S. p. 1-3.

96. Tewari, Vijay. Milan Milenkovic Standards for Autonomic Computing. Intel Technology journal. Volume 10. Issue 04. 2006. Vijay Tewari. Milan Milenkovic.

97. Tosi, Bicocca Davide: Research Perspectives in Self-Healing Systems Università degli Studi di Milano 2004.

98. Vanden Eynden, K. 2007, How can you possibly test modern software fully?
http://www.regdeveloper.co.uk/2007/07/15/all_pairs_testing/
Accessed 28th July 2007.

99. Vilja, John O. Operationally Efficient Propulsion System Study (OEPSS) Data Book.Volume III. Kennedy Space Center (NASA). 1990. P 25-27.

100. Wellbrink, Joerg. Modeling reduced human performance as a complex adapative system. Naval Postgraduate School. California. 2003.

101. Williams, H. R. , R. S. Trask, A. C. Knights, E. R. Williams and I. P. Bond Biomimetic reliability strategies for self-healing vascular networks in engineering materials. Department of Aerospace Engineering, University of Bristol, 2007.

102. Wiseth, Kelli. Oracle Database 10g: The world's first Self managing, Grid-ready Database arrives. Oracle Magazine October 2003.

103. Wood, G. and K. Hailey, 2006, The Self-Managing Database: Automatic Performance Diagnosis with Oracle Database 10g Release 2. Oracle Corporation. U.S. p. 3-4, 16.

104. Zhang, Zheng, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, Chao Jin BitVault: a Highly Reliable Distributed Data Retention Platform. Microsoft Research Asia. 2005.

7.2. Other referenced material during agent design.

1. A History of the Personal Computer: The People and the Technology By Roy A. Allan.

2. Manufacturing Information and Data Systems: Analysis, Design and Practice By Brian (EDT) Griffiths, Franjo Cecelja.

3. Encyclopaedia of Computer Science and Technology: Volume 19 - Supplement 4: Access Technology: Inc. By Allen Kent, Kent Kent.

4. Computer & Information Systems By Cambridge Communications Corporation, Inc Cambridge Scientific Abstracts.

5. Computer Security - Esorics 2004: 2004 Esorics By Pierangela (EDT) Samarati.

6. Organic and Pervasive Computing -- Arcs 2004 By C Müller-Schloer, Theo Ungerer, Bernhard Bauer.

7. The Grid: Core Technologies by Maozhen Li, Mark Baker.

8. A course in probability theory: Third addition. Stanford University. Academic press 2001. Kai Lai Chung.

9. A course in modern analysis and its applications. Cambridge University Press 2003. Graeme L.Cohen.

10. Concepts in Programming Languages. Cambridge University Press 2003. John C. Mitchell.

11. Administrator's Guide to SQL Server 2005. Addison Wesley Professional 2006. Buck Woody.

12. ASP.NET 2.0 Illustrated. Addison Wesley Professional. 2006. Alex Homer, Dave Sussman.

13. Building High Availability Windows Server™ 2003 Solutions. Addison Wesley Professional. 2004. Jeffrey R. Shapiro, Marcin Policht.

14. Designing Effective Database Systems. Addison Wesley Professional. 2005. Rebecca M. Riordan.

15. Essential ASP.NET with Examples in Visual Basic .NET. Addison Wesley. 2003. Fritz Onion.

16. A First Look at ASP.NET v. 2.0. Addison Wesley. 2003. Alex Homer, Dave Sussman, Rob Howard.

17. How To Run Successful Projects III: The Silver Bullet. Addison Wesley. 2001. Fergus O'Connell.

18. Inside SQL Server 2005 Tools. Addison Wesley Professional. 2006. Michael Raheem, Dima Sonkin, Thierry D'Hers, Kami LeMonds.

19. JavaServer Pages™, Second Edition. Addison Wesley. 2003. Larne Pekowsky.

20. .NET Web Services: Architecture and Implementation. Addison Wesley. 2003. Keith Ballinger.

21. SQL Server 2005 Distilled. Addison Wesley Professional. 2006. Eric L. Brown.

22. The Java™ Programming Language, Fourth Edition. Addison Wesley Professional. 2005. Ken Arnold, James Gosling, David Holmes.

23. Windows Forensics and Incident Recovery. Addison Wesley. 2004. Harlan Carvey.

24. Ajax patterns and best practices. Apress. 2006. Christian Gross.

25. An introduction to probability theory. 2004. Christel Geiss and Stefan Geiss.

26. Applied Statistics and Probability for Engineers Third Edition. John Wiley & Sons, Inc. 2003 Douglas C. Montgomery, George C. Runger.

27. Windows administration at the command line. For windows 2003, Windows XP and Windows 2000. Sybex. 2006. John Paul Mueller.

28. Ajax on Rails. Scott Raymond. O’Reilly 2006.

7.3 APPENDICES

7.3.1 Database repository for the S.H.A.D.E engine.

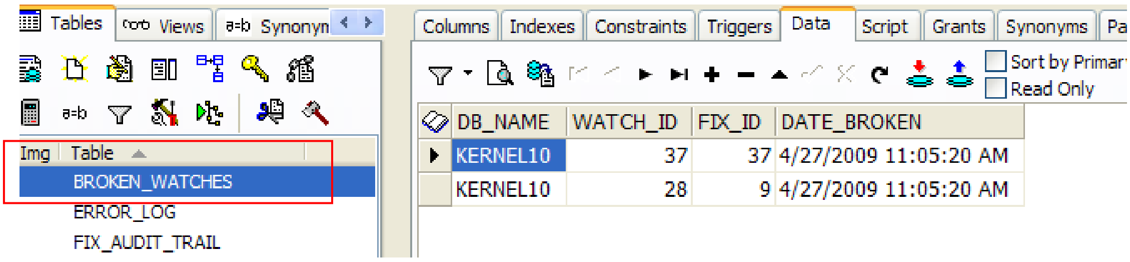
Definitions and descriptions of the S.H.A.D.E database repository concerned with log files and enfine configurations.

Note: Final structure illustrated.

Broken Watches:

This table holds data regarding the “heals” that have failed and needed to be parked because they are not suiting defined failure parameters. Any watches listed in this table will not fire again until removed. The engine displays the context of this table in the user interface and inserts jobs into the log when they have failed more times than allowed. S.H.A.D.E also references this table to verify if heals should be fired.

Illustrated example shows watch 28 (database cache heal) has failed and is thus parked.



Img	Table
	BROKEN_WATCHES
	ERROR_LOG
	FIX_AUDIT_TRAIL

DB_NAME	WATCH_ID	FIX_ID	DATE_BROKEN
KERNEL10	37	37	4/27/2009 11:05:20 AM
KERNEL10	28	9	4/27/2009 11:05:20 AM

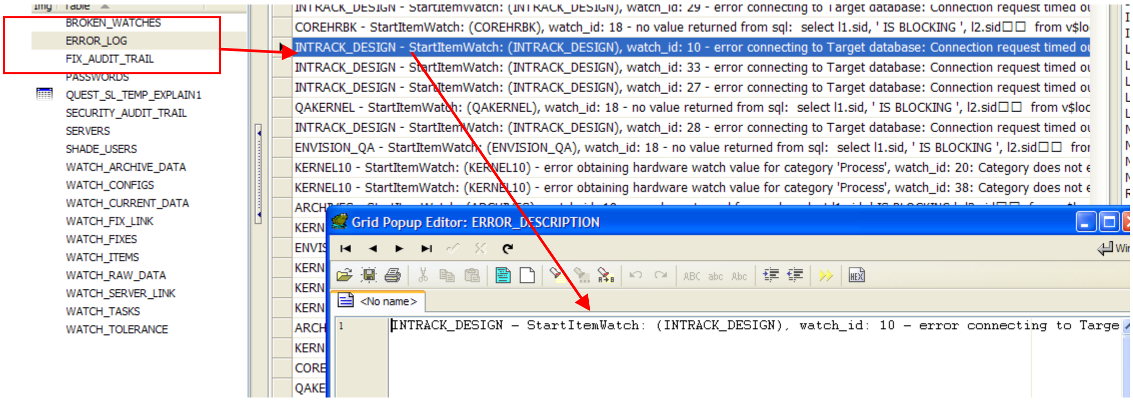
Code:

```
CREATE TABLE S.H.A.D.E..BROKEN_WATCHES
(
  DB_NAME      VARCHAR2(50 BYTE),
  WATCH_ID     NUMBER                          NOT NULL,
  FIX_ID       NUMBER,
  DATE_BROKEN  DATE                            NOT NULL
)
TABLESPACE GENERAL_DATA
```

Error Log:

This table only exists to ease the operations of debugging each and every change to the engine – either to code or database-residing parameters.

Illustrated example shows S.H.A.D.E is not able to connect to database – database server is down



```
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 29 - error connecting to Target database: Connection request timed ou
COREHRBK - StartItemWatch: (COREHRBK), watch_id: 18 - no value returned from sql:  select l1.sid, ' IS BLOCKING ', l2.sid□□  from v$lo
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 10 - error connecting to Target database: Connection request timed ou
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 33 - error connecting to Target database: Connection request timed ou
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 27 - error connecting to Target database: Connection request timed ou
QAKERNEL - StartItemWatch: (QAKERNEL), watch_id: 18 - no value returned from sql:  select l1.sid, ' IS BLOCKING ', l2.sid□□  from v$loc
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 28 - error connecting to Target database: Connection request timed ou
ENVISION_QA - StartItemWatch: (ENVISION_QA), watch_id: 18 - no value returned from sql:  select l1.sid, ' IS BLOCKING ', l2.sid□□  fro
KERNEL10 - StartItemWatch: (KERNEL10) - error obtaining hardware watch value for category 'Process', watch_id: 20: Category does not e
KERNEL10 - StartItemWatch: (KERNEL10) - error obtaining hardware watch value for category 'Process', watch_id: 38: Category does not e
ARCH
KERN
ENVIS
KERN
KERN
KERN
ARCH
KERN
CORE
QAKE
```

```
INTRACK_DESIGN - StartItemWatch: (INTRACK_DESIGN), watch_id: 10 - error connecting to Targe
```

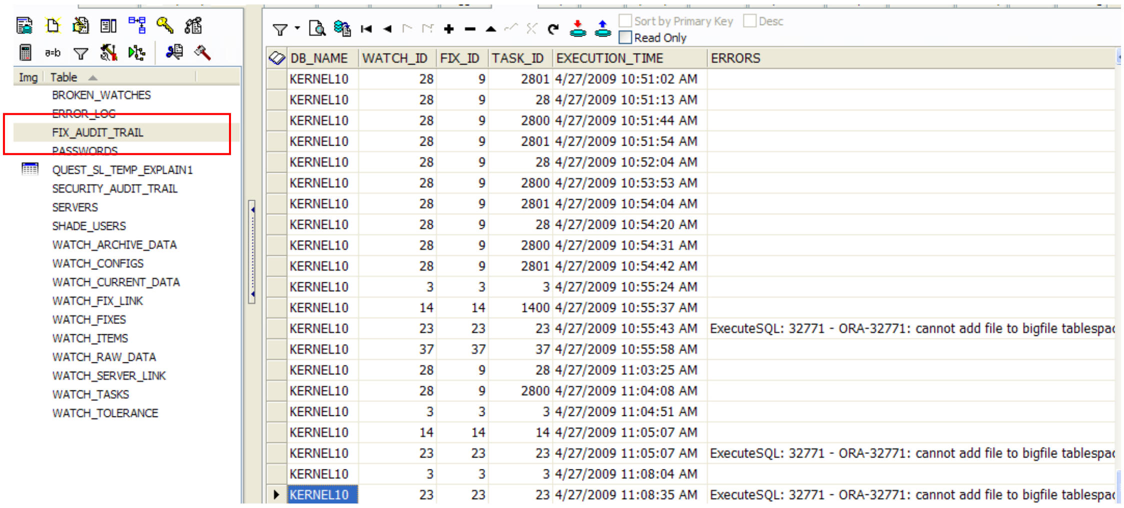
Code:

```
CREATE TABLE S.H.A.D.E..ERROR_LOG
(
  ERROR_SOURCE       VARCHAR2(30 BYTE)          NOT NULL,
  ERROR_DESCRIPTION  VARCHAR2(4000 BYTE)        NOT NULL,
  LOG_DATE           DATE                       NOT NULL
)
TABLESPACE GENERAL_DATA
```

Fix audit trail:

This table contains data especially for S.H.A.D.E fixes that have been fired by the engine. The audit trail is also useful for tracking errors. S.H.A.D.E also references this table to decide which watches are failing and showed to be parked to prevent continuous looping of heals. If the listed heal has been fired and has exhausted re-tries/other options available to it within its designation timescale, it will be set to broken and parked/flagged to the administrator and show in the interface.

Illustrated example shows data on heal/fex 28 that failed and was thus set as broken



DB_NAME	WATCH_ID	FIX_ID	TASK_ID	EXECUTION_TIME	ERRORS
KERNEL10	28	9	2801	4/27/2009 10:51:02 AM	
KERNEL10	28	9	28	4/27/2009 10:51:13 AM	
KERNEL10	28	9	2800	4/27/2009 10:51:44 AM	
KERNEL10	28	9	2801	4/27/2009 10:51:54 AM	
KERNEL10	28	9	28	4/27/2009 10:52:04 AM	
KERNEL10	28	9	2800	4/27/2009 10:53:53 AM	
KERNEL10	28	9	2801	4/27/2009 10:54:04 AM	
KERNEL10	28	9	28	4/27/2009 10:54:20 AM	
KERNEL10	28	9	2800	4/27/2009 10:54:31 AM	
KERNEL10	28	9	2801	4/27/2009 10:54:42 AM	
KERNEL10	3	3	3	4/27/2009 10:55:24 AM	
KERNEL10	14	14	1400	4/27/2009 10:55:37 AM	
KERNEL10	23	23	23	4/27/2009 10:55:43 AM	ExecuteSQL: 32771 - ORA-32771: cannot add file to bigfile tablespac
KERNEL10	37	37	37	4/27/2009 10:55:58 AM	
KERNEL10	28	9	28	4/27/2009 11:03:25 AM	
KERNEL10	28	9	2800	4/27/2009 11:04:08 AM	
KERNEL10	3	3	3	4/27/2009 11:04:51 AM	
KERNEL10	14	14	14	4/27/2009 11:05:07 AM	
KERNEL10	23	23	23	4/27/2009 11:05:07 AM	ExecuteSQL: 32771 - ORA-32771: cannot add file to bigfile tablespac
KERNEL10	3	3	3	4/27/2009 11:08:04 AM	
KERNEL10	23	23	23	4/27/2009 11:08:35 AM	ExecuteSQL: 32771 - ORA-32771: cannot add file to bigfile tablespac

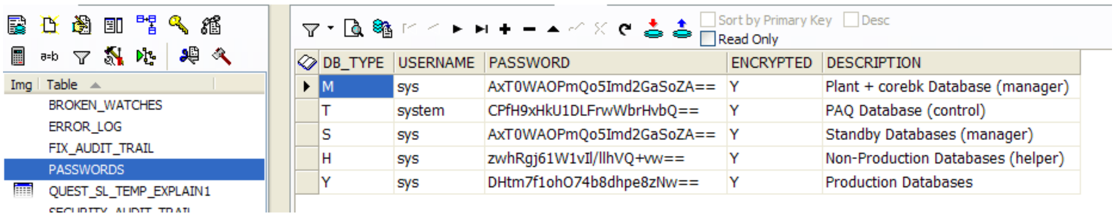
Code:

```
CREATE TABLE S.H.A.D.E..FIX_AUDIT_TRAIL
(
  DB_NAME           VARCHAR2(50 BYTE),
  WATCH_ID          NUMBER,
  FIX_ID         NUMBER,
  TASK_ID          NUMBER,
  EXECUTION_TIME    DATE,
  ERRORS           VARCHAR2(4000 BYTE),
  AUTO             VARCHAR2(1 BYTE),
  EXECUTION_OUTPUT  VARCHAR2(4000 BYTE)
)
TABLESPACE GENERAL_DATA
```

Passwords:

This table contains look up data for the engine to allow it to connect to databases outside of its own engine for monitoring purposes.

Illustrated example shows the passwords are also encrypted to make sure sensitive data cannot be read from the engine's repository.



DB_TYPE	USERNAME	PASSWORD	ENCRYPTED	DESCRIPTION
M	sys	AxT0WAOPmQo5Imd2GaSoZA==	Y	Plant + corebk Database (manager)
T	system	CPfH9xHkU1DLFrwWbrHvbQ==	Y	PAQ Database (control)
S	sys	AxT0WAOPmQo5Imd2GaSoZA==	Y	Standby Databases (manager)
H	sys	zwhRgj61W1vIl/llhVQ+vw==	Y	Non-Production Databases (helper)
Y	sys	DHtm7f1ohO74b8dhpe8zNw==	Y	Production Databases

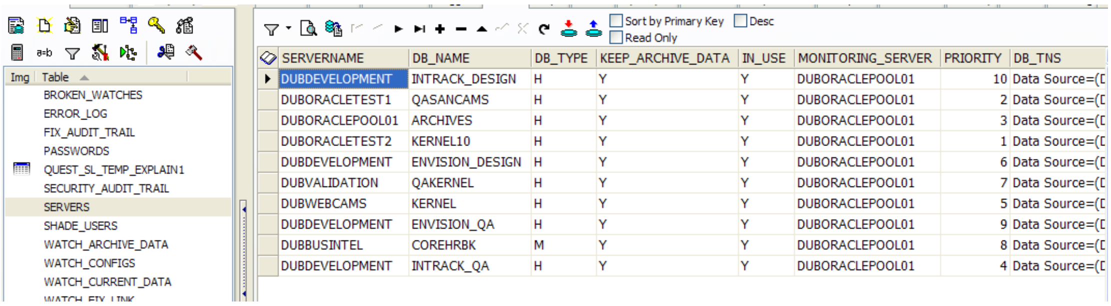
Code:

```
CREATE TABLE S.H.A.D.E..PASSWORDS
(
  DB_TYPE      CHAR(1 BYTE)                NOT NULL,
  USERNAME     VARCHAR2(20 BYTE)              NOT NULL,
  PASSWORD     VARCHAR2(50 BYTE)              NOT NULL,
  ENCRYPTED    CHAR(1 BYTE)                NOT NULL,
  DESCRIPTION  VARCHAR2(50 BYTE)              NOT NULL
)
TABLESPACE GENERAL_DATA
```

Servers:

This table contains look-up data for the engine to allow it to connect to databases and servers outside of its own repository for monitory purposes. The S.H.A.D.E engine uses TNS(less) connectivity, which allows the administrator to change connections by just changing look-up data.

Illustrated example shows a list of servers monitored during S.H.A.D.E development/monitoring.



☐ Sort by Primary Key ☐ Desc
☐ Read Only

Img	Table
	BROKEN_WATCHES
	ERROR_LOG
	FIX_AUDIT_TRAIL
	PASSWORDS
	QUEST_SL_TEMP_EXPLAIN1
	SECURITY_AUDIT_TRAIL
	SERVERS
	SHADE_USERS
	WATCH_ARCHIVE_DATA
	WATCH_CONFIGS
	WATCH_CURRENT_DATA
	WATCH_FIX_LINK

SERVERNAME	DB_NAME	DB_TYPE	KEEP_ARCHIVE_DATA	IN_USE	MONITORING_SERVER	PRIORITY	DB_TNS
DUBDEVELOPMENT	INTRACK_DESIGN	H	Y	Y	DUBORACLEPOOL01	10	Data Source=(D
DUBORACLETEST1	QASANCAMS	H	Y	Y	DUBORACLEPOOL01	2	Data Source=(D
DUBORACLEPOOL01	ARCHIVES	H	Y	Y	DUBORACLEPOOL01	3	Data Source=(D
DUBORACLETEST2	KERNEL10	H	Y	Y	DUBORACLEPOOL01	1	Data Source=(D
DUBDEVELOPMENT	ENVISION_DESIGN	H	Y	Y	DUBORACLEPOOL01	6	Data Source=(D
DUBVALIDATION	QAKERNEL	H	Y	Y	DUBORACLEPOOL01	7	Data Source=(D
DUBWEBCAMS	KERNEL	H	Y	Y	DUBORACLEPOOL01	5	Data Source=(D
DUBDEVELOPMENT	ENVISION_QA	H	Y	Y	DUBORACLEPOOL01	9	Data Source=(D
DUBBUSINTEL	COREHRBK	M	Y	Y	DUBORACLEPOOL01	8	Data Source=(D
DUBDEVELOPMENT	INTRACK_QA	H	Y	Y	DUBORACLEPOOL01	4	Data Source=(D

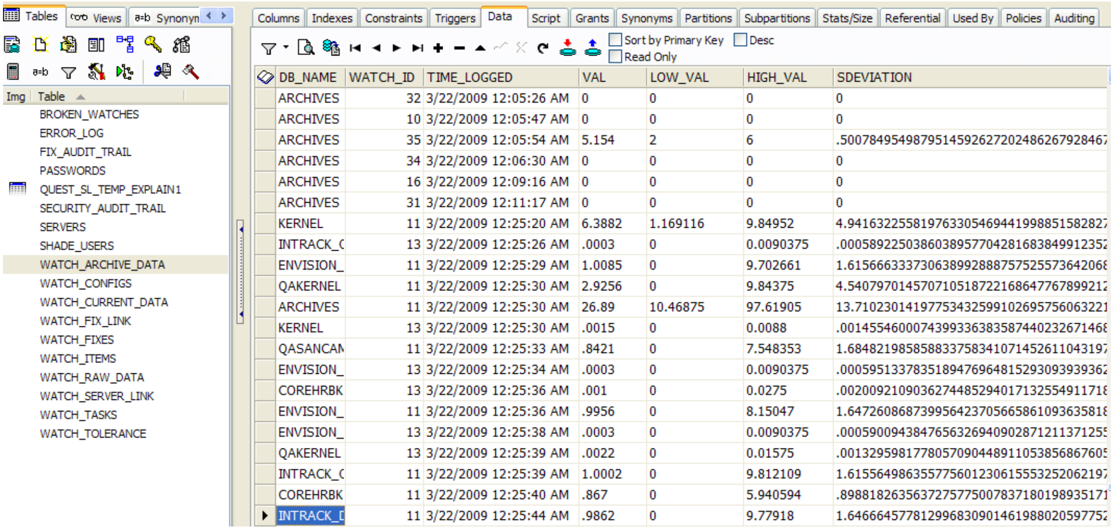
Code:

```
CREATE TABLE S.H.A.D.E..SERVERS
(
  SERVERNAME        VARCHAR2(56 BYTE),
  DB_NAME           VARCHAR2(50 BYTE),
  DB_TYPE           VARCHAR2(1 BYTE)          NOT NULL,
  KEEP_ARCHIVE_DATA  VARCHAR2(1 BYTE),
  IN_USE            CHAR(1 BYTE)              NOT NULL,
  MONITORING_SERVER  VARCHAR2(56 BYTE)          NOT NULL,
  PRIORITY          NUMBER                    NOT NULL,
  DB_TNS            VARCHAR2(255 BYTE)        NOT NULL
)
TABLESPACE GENERAL_DATA
```

Watch archive data:

This table contains the archived live data from the engine on servers monitored and healed. All watches that have been fired will archive under defined parameters and remain permanently in this table for final analysis and compare. All watches are set to archive when a defined count (of rows) is reached (can be individually defined for each watch).

Illustrated example shows data archived on the 22nd March 2009



BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA
WATCH_CONFIGS
WATCH_CURRENT_DATA
WATCH_FIX_LINK
WATCH_FIXES
WATCH_ITEMS
WATCH_RAW_DATA
WATCH_SERVER_LINK
WATCH_TASKS
WATCH_TOLERANCE

DB_NAME	WATCH_ID	TIME_LOGGED	VAL	LOW_VAL	HIGH_VAL	SDEVIATION
ARCHIVES	32	3/22/2009 12:05:26 AM	0	0	0	0
ARCHIVES	10	3/22/2009 12:05:47 AM	0	0	0	0
ARCHIVES	35	3/22/2009 12:05:54 AM	5.154	2	6	.500784954987951459262720248626792846
ARCHIVES	34	3/22/2009 12:06:30 AM	0	0	0	0
ARCHIVES	16	3/22/2009 12:09:16 AM	0	0	0	0
ARCHIVES	31	3/22/2009 12:11:17 AM	0	0	0	0
KERNEL	11	3/22/2009 12:25:20 AM	6.3882	1.169116	9.84952	4.941632255819763305469441998851582827
INTRACK_C	13	3/22/2009 12:25:26 AM	.0003	0	0.0090375	.000589225038603895770428168384991235
ENVISION_	11	3/22/2009 12:25:29 AM	1.0085	0	9.702661	1.615666333730638992888757525573642068
QAKERNEL	11	3/22/2009 12:25:30 AM	2.9256	0	9.84375	4.540797014570710518722168647767899212
ARCHIVES	11	3/22/2009 12:25:30 AM	26.89	10.46875	97.61905	13.71023014197753432599102695756063221
KERNEL	13	3/22/2009 12:25:30 AM	.0015	0	0.0088	.001455460007439933638358744023267146
QASANCAN	11	3/22/2009 12:25:33 AM	.8421	0	7.548353	1.684821985858833758341071452611043197
ENVISION_	13	3/22/2009 12:25:34 AM	.0003	0	0.0090375	.000595133783518947696481529309393936
COREHRBK	13	3/22/2009 12:25:36 AM	.001	0	0.0275	.002009210903627448529401713255491171
ENVISION_	11	3/22/2009 12:25:36 AM	.9956	0	8.15047	1.647260868739956423705665861093635818
ENVISION_	13	3/22/2009 12:25:38 AM	.0003	0	0.0090375	.000590094384765632694090287121137125
QAKERNEL	13	3/22/2009 12:25:39 AM	.0022	0	0.01575	.001329598177805709044891105385686760
INTRACK_C	11	3/22/2009 12:25:39 AM	1.0002	0	9.812109	1.615564986355775601230615553252062197
COREHRBK	11	3/22/2009 12:25:40 AM	.867	0	5.940594	.898818263563727577500783718019893517
INTRACK_D	11	3/22/2009 12:25:44 AM	.9862	0	9.77918	1.646664577812996830901461988020597752

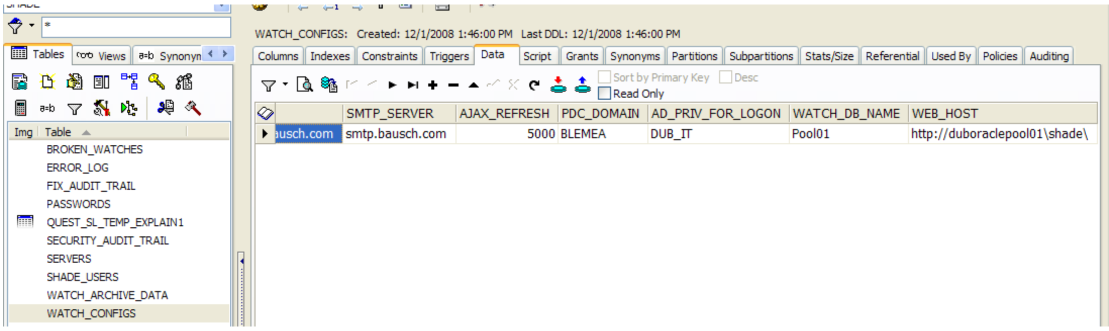
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_ARCHIVE_DATA
(
  DB_NAME       VARCHAR2(56 BYTE)              NOT NULL,
  WATCH_ID      NUMBER                     NOT NULL,
  TIME_LOGGED  DATE                      NOT NULL,
  VAL           VARCHAR2(50 BYTE)              NOT NULL,
  LOW_VAL      VARCHAR2(50 BYTE),
  HIGH_VAL     VARCHAR2(50 BYTE),
  SDEVIATION   VARCHAR2(50 BYTE),
  NUM_ITEMS    NUMBER
)
TABLESPACE GENERAL_DATA
```

Watch configs:

This table defines communications for sending emails as well as location of repository for the S.H.A.D.E engine and Ajax/web server configuration.

Illustrated example shows stmp settings for email alerts



WATCH_CONFIGS: Created: 12/1/2008 1:46:00 PM Last DDL: 12/1/2008 1:46:00 PM

Img	Table
	BROKEN_WATCHES
	ERROR_LOG
	FIX_AUDIT_TRAIL
	PASSWORDS
	QUEST_SL_TEMP_EXPLAIN1
	SECURITY_AUDIT_TRAIL
	SERVERS
	SHADE_USERS
	WATCH_ARCHIVE_DATA
	WATCH_CONFIGS

	SMTP_SERVER	AJAX_REFRESH	PDC_DOMAIN	AD_PRIV_FOR_LOGON	WATCH_DB_NAME	WEB_HOST
ausch.com	smtp.bausch.com	5000	BLEMEA	DUB_IT	Pool01	http://duboraclepool01\shade\

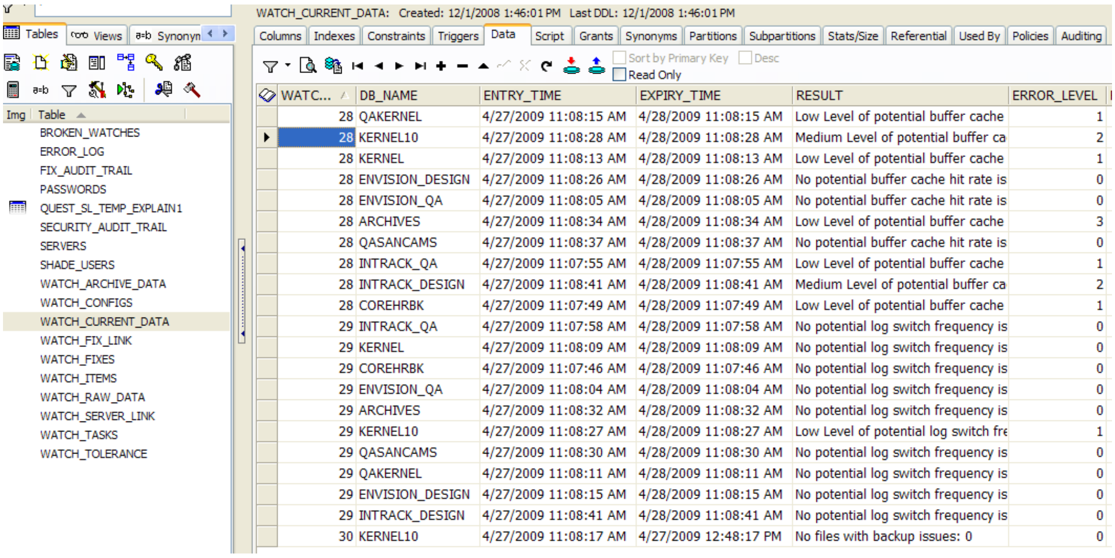
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_CONFIGS
(
  DBA_EMAILS        VARCHAR2(100 BYTE)        NOT NULL,
  SMTP_SERVER       VARCHAR2(56 BYTE)         NOT NULL,
  AJAX_REFRESH      NUMBER                    NOT NULL,
  PDC_DOMAIN        VARCHAR2(30 BYTE)         NOT NULL,
  AD_PRIV_FOR_LOGON  VARCHAR2(30 BYTE)        NOT NULL,
  WATCH_DB_NAME     VARCHAR2(56 BYTE)         NOT NULL,
  WEB_HOST          VARCHAR2(150 BYTE)        NOT NULL
)
TABLESPACE GENERAL_DATA
```

Current data:

This table contains data on heals and items being monitored by the engine, pre-archiving of data. What has been detected and what has been fixed is stored in this table for reference and archiving by the engine when row counts are met.

Illustrated example shows S.H.A.D.E healing buffer cache for monitored database



BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA
WATCH_CONFIGS
WATCH_CURRENT_DATA
WATCH_FIX_LINK
WATCH_FIXES
WATCH_ITEMS
WATCH_RAW_DATA
WATCH_SERVER_LINK
WATCH_TASKS
WATCH_TOLERANCE

WATCH_CURRENT_DATA: Created: 12/1/2008 1:46:01 PM Last DDL: 12/1/2008 1:46:01 PM

WATC...	DB_NAME	ENTRY_TIME	EXPIRY_TIME	RESULT	ERROR_LEVEL
28	QAKERNEL	4/27/2009 11:08:15 AM	4/28/2009 11:08:15 AM	Low Level of potential buffer cache	1
28	KERNEL10	4/27/2009 11:08:28 AM	4/28/2009 11:08:28 AM	Medium Level of potential buffer ca	2
28	KERNEL	4/27/2009 11:08:13 AM	4/28/2009 11:08:13 AM	Low Level of potential buffer cache	1
28	ENVISION_DESIGN	4/27/2009 11:08:26 AM	4/28/2009 11:08:26 AM	No potential buffer cache hit rate is	0
28	ENVISION_QA	4/27/2009 11:08:05 AM	4/28/2009 11:08:05 AM	No potential buffer cache hit rate is	0
28	ARCHIVES	4/27/2009 11:08:34 AM	4/28/2009 11:08:34 AM	Low Level of potential buffer cache	3
28	QASANCAMS	4/27/2009 11:08:37 AM	4/28/2009 11:08:37 AM	No potential buffer cache hit rate is	0
28	INTRACK_QA	4/27/2009 11:07:55 AM	4/28/2009 11:07:55 AM	Low Level of potential buffer cache	1
28	INTRACK_DESIGN	4/27/2009 11:08:41 AM	4/28/2009 11:08:41 AM	Medium Level of potential buffer ca	2
28	COREHRBK	4/27/2009 11:07:49 AM	4/28/2009 11:07:49 AM	Low Level of potential buffer cache	1
29	INTRACK_QA	4/27/2009 11:07:58 AM	4/28/2009 11:07:58 AM	No potential log switch frequency is	0
29	KERNEL	4/27/2009 11:08:09 AM	4/28/2009 11:08:09 AM	No potential log switch frequency is	0
29	COREHRBK	4/27/2009 11:07:46 AM	4/28/2009 11:07:46 AM	No potential log switch frequency is	0
29	ENVISION_QA	4/27/2009 11:08:04 AM	4/28/2009 11:08:04 AM	No potential log switch frequency is	0
29	ARCHIVES	4/27/2009 11:08:32 AM	4/28/2009 11:08:32 AM	No potential log switch frequency is	0
29	KERNEL10	4/27/2009 11:08:27 AM	4/28/2009 11:08:27 AM	Low Level of potential log switch fre	1
29	QASANCAMS	4/27/2009 11:08:30 AM	4/28/2009 11:08:30 AM	No potential log switch frequency is	0
29	QAKERNEL	4/27/2009 11:08:11 AM	4/28/2009 11:08:11 AM	No potential log switch frequency is	0
29	ENVISION_DESIGN	4/27/2009 11:08:15 AM	4/28/2009 11:08:15 AM	No potential log switch frequency is	0
29	INTRACK_DESIGN	4/27/2009 11:08:41 AM	4/28/2009 11:08:41 AM	No potential log switch frequency is	0
30	KERNEL10	4/27/2009 11:08:17 AM	4/27/2009 12:48:17 PM	No files with backup issues: 0	0

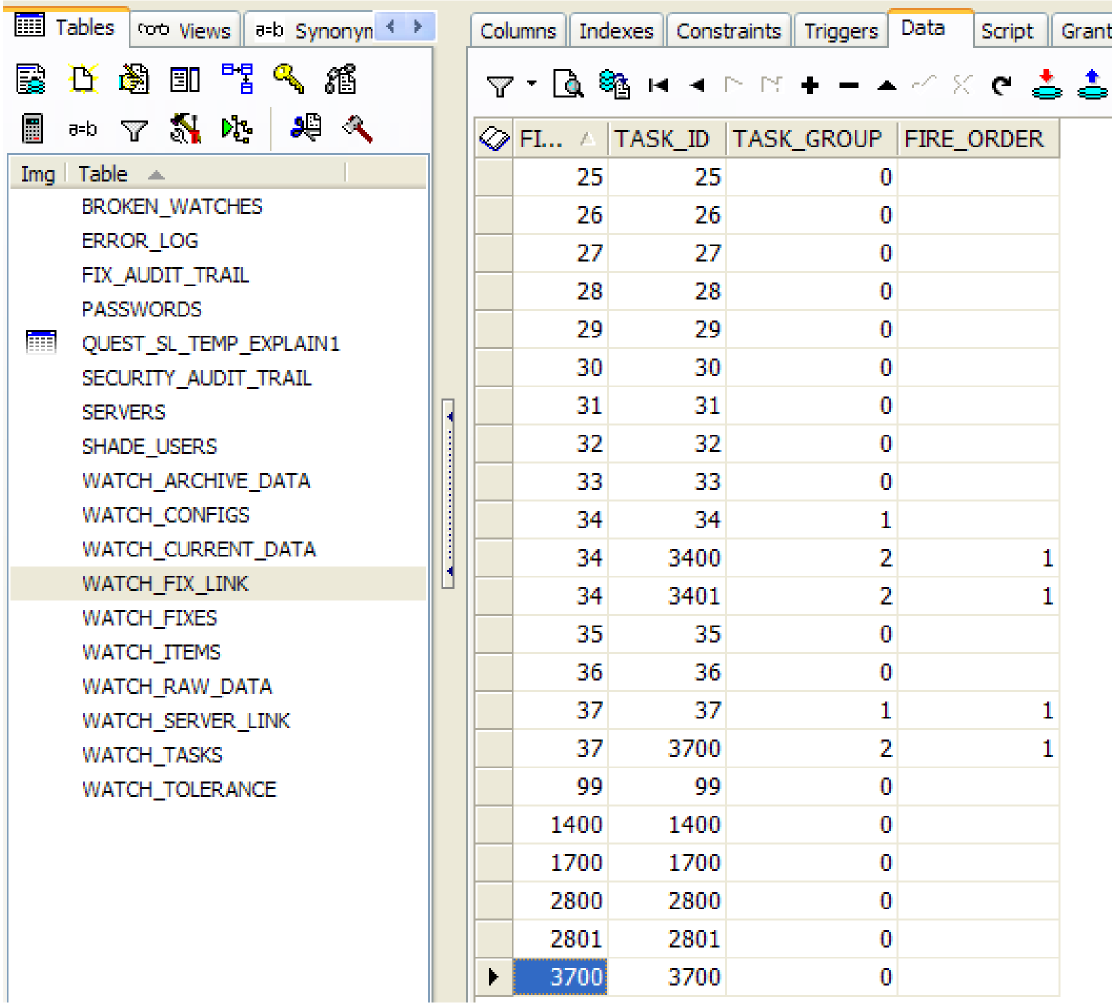
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_CURRENT_DATA
(
  WATCH_ID     NUMBER                        NOT NULL,
  DB_NAME      VARCHAR2(56 BYTE)                 NOT NULL,
  ENTRY_TIME   DATE                        NOT NULL,
  EXPIRY_TIME  DATE                        NOT NULL,
  RESULT       VARCHAR2(100 BYTE)                NOT NULL,
  ERROR_LEVEL  NUMBER                        NOT NULL,
  FIX_ID       NUMBER,
  DESCRIPTION  VARCHAR2(50 BYTE)                 NOT NULL
)
TABLESPACE GENERAL_DATA
```

Fix Link:

Contains configuration data on heals. Which heal is to be fired if first option fails to solve the problem and how many fixes are assigned to each watch. Order is defined in this table, and the engine references which fix should be used if more than one fix is defined.

Illustrated example shows heal options for watch 28:



Img	Table
	BROKEN_WATCHES
	ERROR_LOG
	FIX_AUDIT_TRAIL
	PASSWORDS
	QUEST_SL_TEMP_EXPLAIN1
	SECURITY_AUDIT_TRAIL
	SERVERS
	SHADE_USERS
	WATCH_ARCHIVE_DATA
	WATCH_CONFIGS
	WATCH_CURRENT_DATA
	WATCH_FIX_LINK
	WATCH_FIXES
	WATCH_ITEMS
	WATCH_RAW_DATA
	WATCH_SERVER_LINK
	WATCH_TASKS
	WATCH_TOLERANCE

FI...	TASK_ID	TASK_GROUP	FIRE_ORDER
25	25	0	
26	26	0	
27	27	0	
28	28	0	
29	29	0	
30	30	0	
31	31	0	
32	32	0	
33	33	0	
34	34	1	
34	3400	2	1
34	3401	2	1
35	35	0	
36	36	0	
37	37	1	1
37	3700	2	1
99	99	0	
1400	1400	0	
1700	1700	0	
2800	2800	0	
2801	2801	0	
3700	3700	0	

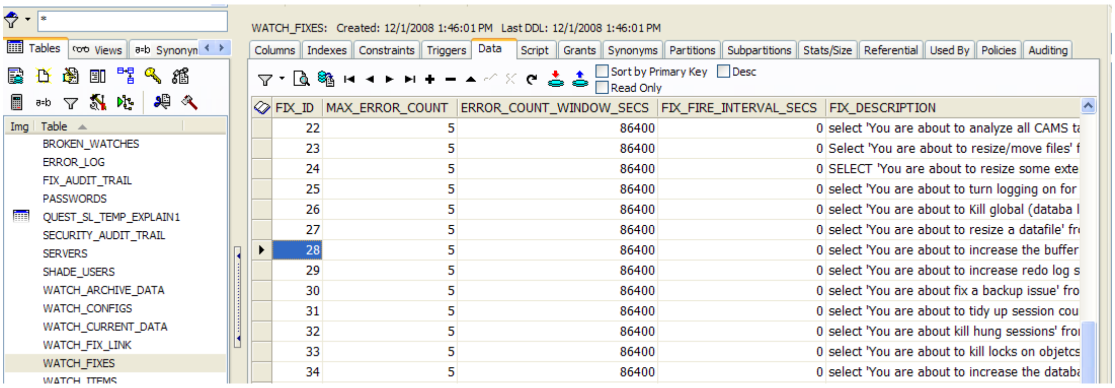
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_FIX_LINK
(
  FIX_ID      NUMBER,
  TASK_ID     NUMBER,
  TASK_GROUP  NUMBER                    DEFAULT 0                 NOT
NULL,
  FIRE_ORDER  NUMBER
)
TABLESPACE GENERAL_DATA
```

Watch fixes:

This table contains parameters that the engine references to define how many times a “heal” can be fired inside a defined time window before it is set to fail. Re-try attempts are also defined.

Illustrated example shows watch 28 can fail only 5 times in 86400 seconds before being flagged as being in an error state.



WATCH_FIXES: Created: 12/1/2008 1:46:01 PM Last DDL: 12/1/2008 1:46:01 PM

FIX_ID	MAX_ERROR_COUNT	ERROR_COUNT_WINDOW_SECS	FIX_FIRE_INTERVAL_SECS	FIX_DESCRIPTION
22	5	86400	0	select 'You are about to analyze all CAMS ta
23	5	86400	0	Select 'You are about to resize/move files' f
24	5	86400	0	SELECT 'You are about to resize some exte
25	5	86400	0	select 'You are about to turn logging on for
26	5	86400	0	select 'You are about to Kill global (databa l
27	5	86400	0	select 'You are about to resize a datafile' fr
28	5	86400	0	select 'You are about to increase the buffer
29	5	86400	0	select 'You are about to increase redo log s
30	5	86400	0	select 'You are about fix a backup issue' fro
31	5	86400	0	select 'You are about to tidy up session cou
32	5	86400	0	select 'You are about kill hung sessions' fro
33	5	86400	0	select 'You are about to kill locks on objetcs
34	5	86400	0	select 'You are about to increase the databa

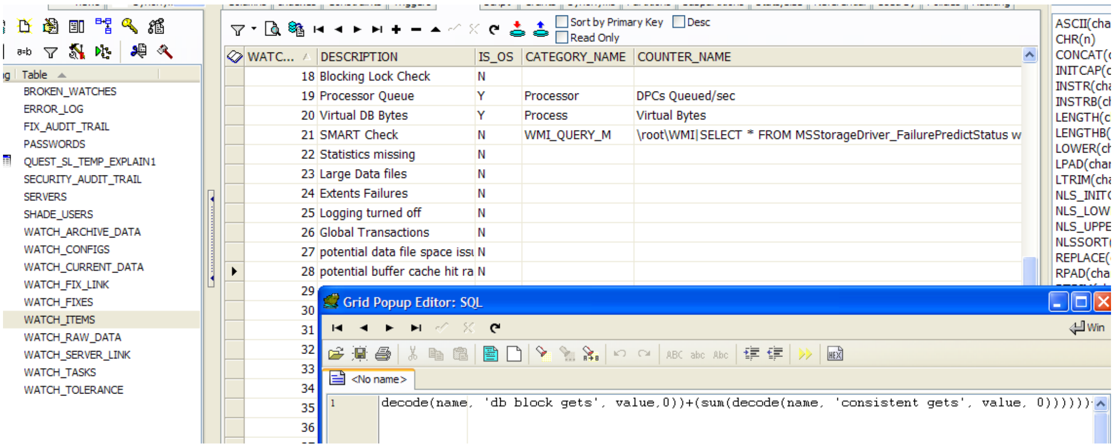
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_FIXES
(
  FIX_ID                    NUMBER,
  MAX_ERROR_COUNT           NUMBER              NOT NULL,
  ERROR_COUNT_WINDOW_SECS  NUMBER              NOT NULL,
  FIX_FIRE_INTERVAL_SECS   NUMBER              DEFAULT 0
NOT NULL,
  FIX_DESCRIPTION          VARCHAR2(1024 BYTE)  NOT NULL
)
TABLESPACE GENERAL_DATA
```

Watch Items:

This is one of the main tables for the repository and configuration of the engine as it defines how the watches query the database, what code they execute and how often the code is fired.

Illustrated example shows code that is fired by watch 28 to query the database



WATC...	DESCRIPTION	IS_OS	CATEGORY_NAME	COUNTER_NAME
18	Blocking Lock Check	N		
19	Processor Queue	Y	Processor	DPCs Queued/sec
20	Virtual DB Bytes	Y	Process	Virtual Bytes
21	SMART Check	N	WMI_QUERY_M	\root\WMI|SELECT * FROM MSStorageDriver_FailurePredictStatus w
22	Statistics missing	N		
23	Large Data files	N		
24	Extents Failures	N		
25	Logging turned off	N		
26	Global Transactions	N		
27	potential data file space issı	N		
28	potential buffer cache hit ra	N		
29				
30				
31				
32				
33				
34				
35				
36				

```
decode(name, 'db block gets', value,0))+(sum(decode(name, 'consistent gets', value, 0))))))
```

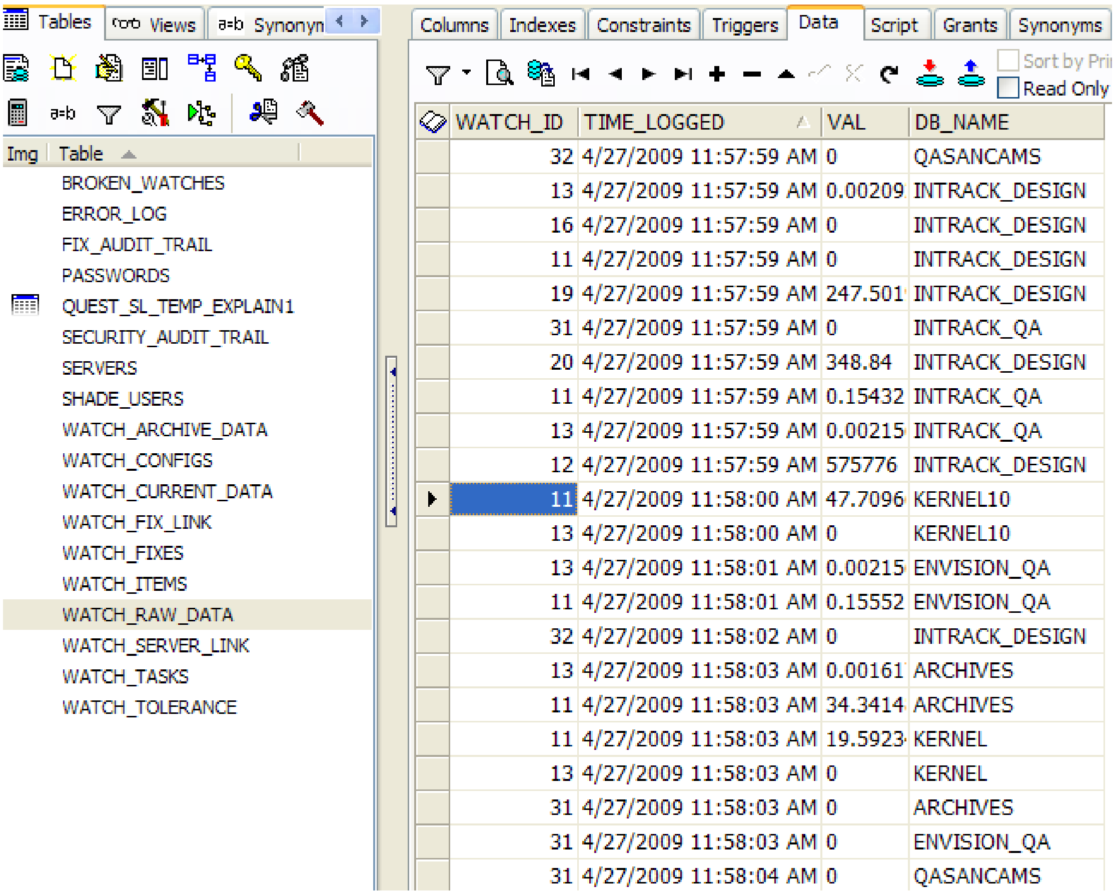
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_ITEMS
(
  WATCH_ID           NUMBER,
  DESCRIPTION        VARCHAR2(255 BYTE)        NOT NULL,
  IS_OS              VARCHAR2(1 BYTE)          NOT NULL,
  CATEGORY_NAME      VARCHAR2(30 BYTE),
  COUNTER_NAME       VARCHAR2(100 BYTE),
  INSTANCE_NAME      VARCHAR2(30 BYTE),
  IN_USE             CHAR(1 BYTE)              NOT NULL,
  REFRESH_INTERVAL   NUMBER,
  SQL                VARCHAR2(1024 BYTE),
  VIEW_IN_PANEL      CHAR(1 BYTE)              NOT NULL,
  ARCHIVE_COUNT      NUMBER                    NOT NULL,
  KEEP_ARCHIVE_DATA  CHAR(1 BYTE)              NOT NULL,
  OK_MESSAGE         VARCHAR2(128 BYTE),
  CONSEC_ERROR_HALT  NUMBER                    NOT NULL,
  ERROR_MSECS_RETRY  NUMBER                    NOT NULL
)
TABLESPACE GENERAL_DATA
```

Raw Data

This table stores live data from the engine on server watches being monitored and alerts being detected. This table is basically the stored results obtained from each watch. Watch id, time, value retrieved and database name are stored here for reference by the engine to decide what is faulted and what needs healing.

Illustrated example shows watch 11 data collection for test system on specific date



BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA
WATCH_CONFIGS
WATCH_CURRENT_DATA
WATCH_FIX_LINK
WATCH_FIXES
WATCH_ITEMS
WATCH_RAW_DATA
WATCH_SERVER_LINK
WATCH_TASKS
WATCH_TOLERANCE

WATCH_ID	TIME_LOGGED	VAL	DB_NAME
32	4/27/2009 11:57:59 AM	0	QASANCAMS
13	4/27/2009 11:57:59 AM	0.00209	INTRACK_DESIGN
16	4/27/2009 11:57:59 AM	0	INTRACK_DESIGN
11	4/27/2009 11:57:59 AM	0	INTRACK_DESIGN
19	4/27/2009 11:57:59 AM	247.501	INTRACK_DESIGN
31	4/27/2009 11:57:59 AM	0	INTRACK_QA
20	4/27/2009 11:57:59 AM	348.84	INTRACK_DESIGN
11	4/27/2009 11:57:59 AM	0.15432	INTRACK_QA
13	4/27/2009 11:57:59 AM	0.00215	INTRACK_QA
12	4/27/2009 11:57:59 AM	575776	INTRACK_DESIGN
11	4/27/2009 11:58:00 AM	47.7096	KERNEL10
13	4/27/2009 11:58:00 AM	0	KERNEL10
13	4/27/2009 11:58:01 AM	0.00215	ENVISION_QA
11	4/27/2009 11:58:01 AM	0.15552	ENVISION_QA
32	4/27/2009 11:58:02 AM	0	INTRACK_DESIGN
13	4/27/2009 11:58:03 AM	0.00161	ARCHIVES
11	4/27/2009 11:58:03 AM	34.3414	ARCHIVES
11	4/27/2009 11:58:03 AM	19.5923	KERNEL
13	4/27/2009 11:58:03 AM	0	KERNEL
31	4/27/2009 11:58:03 AM	0	ARCHIVES
31	4/27/2009 11:58:03 AM	0	ENVISION_QA
31	4/27/2009 11:58:04 AM	0	QASANCAMS

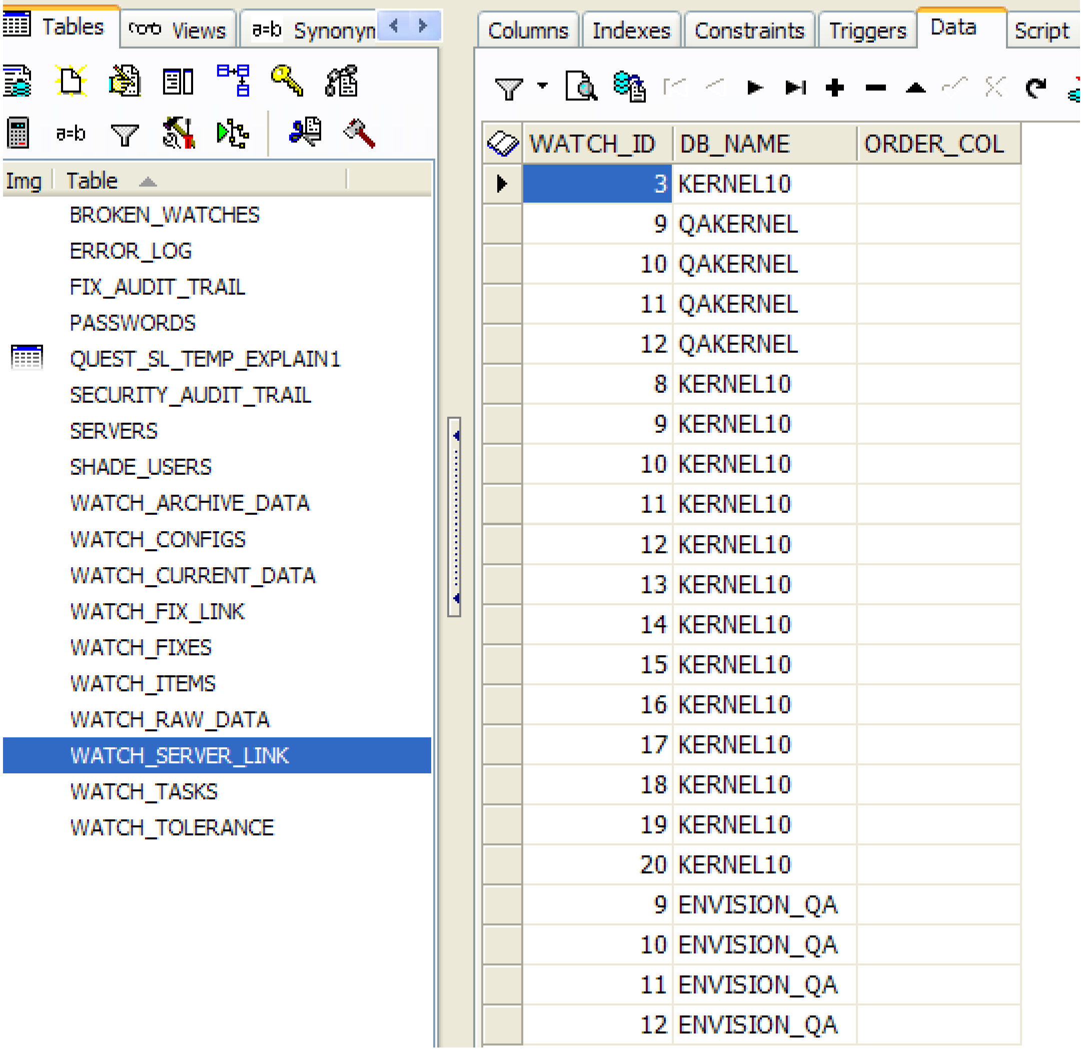
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_RAW_DATA
(
  WATCH_ID     NUMBER                          NOT NULL,
  TIME_LOGGED  DATE                            NOT NULL,
  VAL          VARCHAR2(50 BYTE)               NOT NULL,
  DB_NAME      VARCHAR2(56 BYTE)               NOT NULL
)
TABLESPACE GENERAL_DATA
```

Server Link:

This is a look-up table for the engine that decides which watch is allowed for which system. If the watch is not listed for a specific system here, the engine will not fire the monitoring and thus healing code for that server/database.

Illustrated example shows list of watches for Kernel10 test system



BROKEN_WATCHES
ERROR_LOG
FIX_AUDIT_TRAIL
PASSWORDS
QUEST_SL_TEMP_EXPLAIN1
SECURITY_AUDIT_TRAIL
SERVERS
SHADE_USERS
WATCH_ARCHIVE_DATA
WATCH_CONFIGS
WATCH_CURRENT_DATA
WATCH_FIX_LINK
WATCH_FIXES
WATCH_ITEMS
WATCH_RAW_DATA
WATCH_SERVER_LINK
WATCH_TASKS
WATCH_TOLERANCE

WATCH_ID	DB_NAME	ORDER_COL
3	KERNEL10	
9	QAKERNEL	
10	QAKERNEL	
11	QAKERNEL	
12	QAKERNEL	
8	KERNEL10	
9	KERNEL10	
10	KERNEL10	
11	KERNEL10	
12	KERNEL10	
13	KERNEL10	
14	KERNEL10	
15	KERNEL10	
16	KERNEL10	
17	KERNEL10	
18	KERNEL10	
19	KERNEL10	
20	KERNEL10	
9	ENVISION_QA	
10	ENVISION_QA	
11	ENVISION_QA	
12	ENVISION_QA	

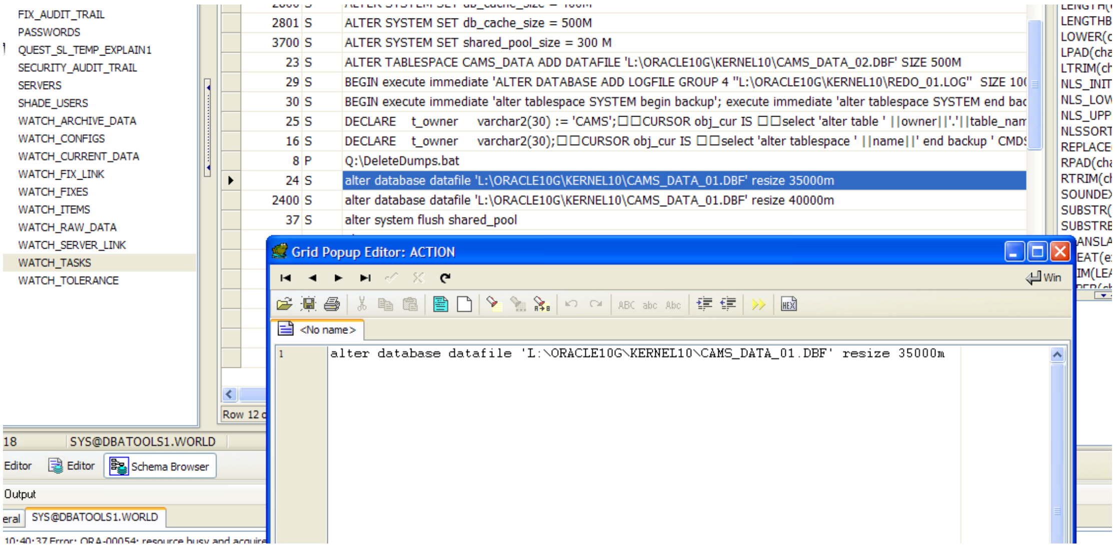
Code

```
CREATE TABLE S.H.A.D.E..WATCH_SERVER_LINK
(
  WATCH_ID   NUMBER                    NOT NULL,
  DB_NAME    VARCHAR2(56 BYTE)               NOT NULL,
  ORDER_COL  NUMBER
)
TABLESPACE GENERAL_DATA
```

Watch tasks:

This table contains parameters and code for the individual heals assigned to task identifiers. If a “heal” exists for a watch, it will be referenced and fired by the engine here. If the task fails and has another option for the task that it can re-try, this task is assigned and defined here also.

Illustrated example shows code to be fired by the engine for task id 24



```
2801 S  ALTER SYSTEM SET db_cache_size = 500M
3700 S  ALTER SYSTEM SET shared_pool_size = 300 M
  23 S  ALTER TABLESPACE CAMS_DATA ADD DATAFILE 'L:\ORACLE10G\KERNEL10\CAMS_DATA_02.DBF' SIZE 500M
  29 S  BEGIN execute immediate 'ALTER DATABASE ADD LOGFILE GROUP 4 "L:\ORACLE10G\KERNEL10\REDO_01.LOG"  SIZE 10(
  30 S  BEGIN execute immediate 'alter tablespace SYSTEM begin backup'; execute immediate 'alter tablespace SYSTEM end bac
  25 S  DECLARE   t_owner    varchar2(30) := 'CAMS';□□CURSOR obj_cur IS □□select 'alter table ' ||owner||'.'||table_nam
  16 S  DECLARE   t_owner    varchar2(30);□□CURSOR obj_cur IS □□select 'alter tablespace ' ||name||' end backup ' CMD
   8 P  Q:\DeleteDumps.bat
  24 S  alter database datafile 'L:\ORACLE10G\KERNEL10\CAMS_DATA_01.DBF' resize 35000m
2400 S  alter database datafile 'L:\ORACLE10G\KERNEL10\CAMS_DATA_01.DBF' resize 40000m
  37 S  alter system flush shared_pool
```

Grid Popup Editor: ACTION

```
alter database datafile 'L:\ORACLE10G\KERNEL10\CAMS_DATA_01.DBF' resize 35000m
```

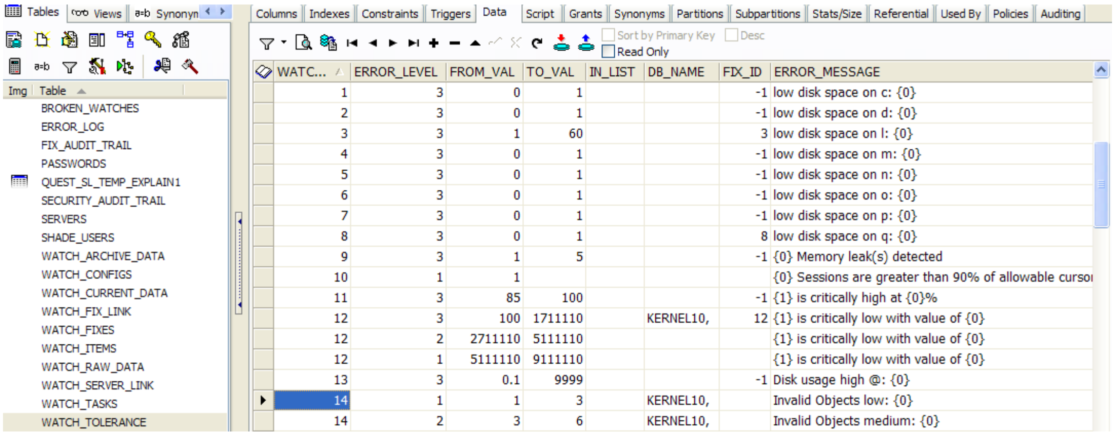
Code:

```
CREATE TABLE S.H.A.D.E..WATCH_TASKS
(
  TASK_ID            NUMBER,
  TYPE               CHAR(1 BYTE),
  ACTION             VARCHAR2(4000 BYTE),
  RETRY_FIX_ID       NUMBER                    DEFAULT NULL,
  AUTO               CHAR(1 BYTE)              NOT NULL,
  NOTIFY_ON_SUCCESS  VARCHAR2(1 BYTE)          DEFAULT 'N'
NOT NULL,
  NOTIFY_ON_FAILURE  VARCHAR2(1 BYTE)          DEFAULT 'N'
NOT NULL
)
TABLESPACE GENERAL_DATA
```

Watch tolerance

This table contains data on parameters that define the error levels for issues detected. Each watch has a level of 1 to 3 where 3 is in high alert and will fire a fix, if one is defined. Anything outside of a 3 is simply flagged as a potential future problem.

Illustrated example shows task 14 and its defined tolerances



WATC...	ERROR_LEVEL	FROM_VAL	TO_VAL	IN_LIST	DB_NAME	FIX_ID	ERROR_MESSAGE
1	3	0	1			-1	low disk space on c: {0}
2	3	0	1			-1	low disk space on d: {0}
3	3	1	60			3	low disk space on l: {0}
4	3	0	1			-1	low disk space on m: {0}
5	3	0	1			-1	low disk space on n: {0}
6	3	0	1			-1	low disk space on o: {0}
7	3	0	1			-1	low disk space on p: {0}
8	3	0	1			8	low disk space on q: {0}
9	3	1	5			-1	{0} Memory leak(s) detected
10	1	1					{0} Sessions are greater than 90% of allowable curso
11	3	85	100			-1	{1} is critically high at {0}%
12	3	100	1711110		KERNEL10,	12	{1} is critically low with value of {0}
12	2	2711110	5111110				{1} is critically low with value of {0}
12	1	5111110	9111110				{1} is critically low with value of {0}
13	3	0.1	9999			-1	Disk usage high @: {0}
14	1	1	3		KERNEL10,		Invalid Objects low: {0}
14	2	3	6		KERNEL10,		Invalid Objects medium: {0}

Code:

```
CREATE TABLE S.H.A.D.E..WATCH_TOLERANCE
(
  WATCH_ID       NUMBER                        NOT NULL,
  ERROR_LEVEL    NUMBER                        NOT NULL,
  FROM_VAL       NUMBER,
  TO_VAL         NUMBER,
  IN_LIST        VARCHAR2(50 BYTE),
  DB_NAME        VARCHAR2(50 BYTE),
  FIX_ID         NUMBER,
  ERROR_MESSAGE  VARCHAR2(128 BYTE),
  NOTIN_LIST     VARCHAR2(50 BYTE)
)
TABLESPACE GENERAL_DATA
```

7.4 Appendix 2 – Commercial Software with Self-Healing Capabilities (present and future versions).

Company	Product	WebSite (for reference only)
Oracle	Oracle 10 + 11g	www.oracle.com
Microsoft	Windows Vista	www.microsoft.com
Microsoft	Sql Server 2005	www.microsoft.com
Diskeeper Corp.	Diskeeper	www.diskeeper.com
Google	Google apps/search	www.google.com
Intel	Support and services	www.intel.com

7.5 Products tested, researched and utilised during engine design.

- Oracle 9i release 2 &10g release 2.
- Quest Iwatch.
- Quest Foglight.
- Quest Spotlight (Windows and Oracle Database versions).
- Quest TOAD.
- Quest Central for Databases.
- Quest Database expert.
- Quest SQL tuning.
- Quest Performance Analysis.
- Oracle Enterprise Manager.
- Oracle Tuning pack.
- Oracle Diagnostic pack.
- Nimbus for Database monitoring and reporting.
- BMC.
- Solaris 10 O/S.
- Windows Vista/2000/XP.
- Microsoft SQL Server 2005.
- DisKeeper 2007 premium.
- BitVault
- Ariolic Software: Active Smart.