Bulletin of the Technical Committee on

Data Engineering

September 2006 Vol. 29 No. 3

IEEE Computer Society

Letters

Special Issue on Self-Managing Database Systems

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

There are two Data Engineering Bulletin web sites: http://www.research.microsoft.com/research/db/debull and http://sites.computer.org/debull/.
The TC on Data Engineering web page is http://www.ipsi.fraunhofer.de/tcde/.

Letter from the Editor-in-Chief

The IEEE Technical Committee on Data Engineering

The Technical Committee on Data Engineering (TCDE) is the sponsoring technical committee within the IEEE Computer Society both for the Data Engineering Bulletin that you are currently reading and for the International Conference on Data Engineering (ICDE). The TCDE has an elected chair who then appoints an executive committee. Being chair of the TCDE is a responsible and important position. Hence, voting in the election of a new TCDE chair is an important duty of TCDE members. So I would urge you to vote. Information from the nominating committee and on the candidate that they have nominated is on the following page. You will find the ballot form on page 3. I urge you to vote.

The Current Issue

Users have eagerly bought the wonderful technology provided by database vendors while simultaneously struggling to exploit it effectively. So making systems, and particularly database systems, easier to use has been on the hot list of research topics for many years. It is very difficult to make systems simple to use. And the early industrial efforts were not very successful, with systems performing badly ”out of the box” and with users required to provide insightful values for dozens of performance parameters to improve on this. Things began to change in the 1990’s when Surajit Chaudhuri’s insights on automatic index selection led to an explosion of new industrial interest in this area.

Over time, this renewed industrial and research interest in automatic management of database systems led to the formation of the Working Group on Self Managing Database Systems, chaired by Sam Lightstone. Sam, in representing the working group, intended to organize issues of the Bulletin devoted to this topic. By happy circumstance, Natassa Ailamaki, the current issue editor also had an interest in self-managing systems and decided to proceed with an issue on this topic. This was a bit earlier than I had expected, but surely timely considering the importance of the topic. The issue contains articles both from researchers and from technical folks working for the three dominant database vendors. This is the kind of issue that is the strength of the Bulletin, at the intersection of research and industrial practice. I want to thank Natassa Ailamaki for initiating the effort that led to the issue and for her hard work in bringing it to fruition. Readers will be well rewarded by the technical papers presented here.

David Lomet
Microsoft Corporation

TC on Data Engineering: Election of Chair for 2007-2008

TC on Data Engineering: Election of Chair for 2004-2005

The Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) is elected for a two-year period. The mandate of the current Chair, Erich Neuhold, terminates at the end of 2006. Hence is time to elect a Chair for the period January 2007 to December 2009. Please vote before December 1, 2006 using the ballot on the next page.

The Nominating Committee, consisting of Betty Salzberg (chair), Erich Neuhold, and David Lomet is nominating Paul Larson as Chair of TCDE. Paul's position statement and a short biography are included below. The Committee invited nominations from members of the TCDE but received no other nominations.

Betty Salzberg, David Lomet, Erich Neuhold
Nominating Committee

Position Statement and Biography

Biography

Per-Åke (Paul) Larson is a Principal Researcher at Microsoft Research. His primary research area is query optimization and query processing in database systems. Prior to joining Microsoft Research, he was a Professor in the Department of Computer Science at the University of Waterloo, Canada, serving as department chair for three years. He received his Ph.D. from bo Akademi University in Finland where he also served as an Assistant Professor. He is a Fellow of the ACM.

Position statement

The International Conference on Data Engineering (ICDE) is the main conference sponsored by the Technical Committee on Data Engineering (TCDE). Both its quality and attendance have improved greatly over the last few years. If elected I will continue working to further increase its visibility and quality. I will work closely with the ICDE Steering Committee to achieve this goal and actively seek input and suggestions from the database community.

Paul Larson
Microsoft Research

ELECTION BALLOT

ELECTION BALLOT

IEEE COMPUTER SOCIETY

TECHNICAL COMMITTEE ON
DATA ENGINEERING

The Technical Committee on Data Engineering (TCDE) is holding an election for Chair. The term of the current chair, Erich Neuhold, expires at the end of 2006. Please email, mail or fax in your vote.

BALLOT FOR ELECTION OF CHAIR
Term: (January, 2007 - December, 2008)

Please vote for one candidate.

○ Paul Larson

○ ____________
(write in)

Your Signature:____

Your Name:____

IEEE CS Membership No.:________________________________
(Note: You must provide your member number. Only TCDE members who are Computer Society members are eligible to vote.)

Please email, mail or fax the ballot to arrive by December 1, 2006 to:

s.wagner@computer.org
Fax: +1-202-728-0884

IEEE Computer Society
Attn: Stacy Wagner
1730 Massachusetts Avenue, NW
Washington, DC 20036-1992

RETURN BY December 1, 2006

TC ON DATA ENGINEERING
IEEE Computer Society

Letter from Chair of the Working Group on Self-Managing Database Systems

"Civilization advances by extending the number of important operations which we can perform without thinking about them." – Alfred Whitehead

The topic of administration cost is growing more important every year and the problems that require our research and development are changing. Why is that? The rise of the Internet, online transaction processing, online banking, and the ability to connect heterogeneous systems have all contributed to the massive growth in data volumes over the past 15 years. Terabyte sized databases have become commonplace. Concurrent with this data growth, have come dramatic increases in CPU performance, spurred by Moores Law, and improvements in disk fabrication which have improved data density limits for persistent disk storage. At the same time there has been a large shift in the nature of the data we manage. Over the past decade data has made a dramatic shift from being predominantly well structured, to being increasingly unstructured (such as image, audio data streams, and video) and semi-structured data (such as XML). Users naturally expect the same benefits that relational databases achieved with structured data to apply to new data types as well – concurrency control, performance, data manipulation, rich language support, availability and recovery, security and authority to name a few. The increasing volumes of data, and the changing kinds of data we manage also spur "creeping featurism" as clever scientists and engineers find new ways to provide function over volume, scale and content. Creeping featurism adds administrative complexity to systems. We end up with more data, more features, more options, more things to think about and a lot more complexity. End users end up spending more on the staff to manage the complexity of information technology than on the products they purchased. Because featurism will not cease to expand our paradigm must change.

The urgent need to reduce administrative complexity is not unique to database systems and is pervasive throughout the IT industry. I'd like to share with you a very simple and hopefully entertaining analysis: In a brief examination of 3 popular middleware products produced by different companies, we observed the number of configuration/registry parameters ranged from 384 to 1200. These parameters were used to specify everything from memory configuration to connection and process limits. Many of the parameters have a dynamic range of potential values (for example, configuring memory allocations to the database caching areas can have a wide range of values). The most extreme simplification of the configuration space considers each configuration parameter as having a binary setting (ON/OFF or TRUE/FALSE etc). Using this gross oversimplification, and completely excluding the complexity of logical and physical design, the possible configurations for a product with 384 parameters is 10^{115} while a product having 1200 binary parameters would have 10^{361} configurations. These numbers are monstrous, far beyond the ability of human beings to assess. Astrophysicists estimate the upper bound on the number of atoms in the universe[1] to be 10^{81}, many orders of magnitude less than the number of ways one can configure the three middleware products we studied, even with dramatic oversimplifying assumptions. It matters because the alternatives are bleak: poorly administered high cost systems that run at fractions of their potential. The answer to this is self-managing autonomic systems that just work without fuss and bother.

The following topics generally frame the domain for self-managing information management systems and are the areas where continued research and development are required for our industry to be successful in significantly reducing administrative cost.

1. *Self-Configuring*: Information management systems that are trivial to set up.
2. *Self-Healing*: Information systems that know their own problems.
3. *Self-Optimizing*: Information systems that maximize their own performance/efficiency.

[1]Current estimates are in the range of 10^{63} to 10^{81} atoms. I have used the upper bound (worst case) to strengthen the argument.

4. *Trust in self-managing systems*: How is trust in automation fostered when, especially when the financial stakes are high?

5. *Benchmarking of self-managing information systems*: How are self-managing information systems best evaluated?

6. *Self-Protecting*: Databases that protect themselves from security attacks.

7. *System-wide self-management*: Systems that are self-managing as an integrated whole, not only by components.

To advance research and development in self-managing database and information management systems, the IEEE Data Engineering Workgroup on Self Managing Database Systems (which was formally announced a few months ago) will sponsor workshops in conjunction with the ICDE, foster publications, and maintain an online collection of links to key resources. I am pleased that our initial executive committee includes some of the leading advocates of self-managing database technology from across geographies in both industry and academia. Dr. Guy Lohman is organizing a workshop on Self Managing Database Systems at ICDE 2007. We invite readers to follow the news and announcements about the workgroup's activities at our website: http://db.uwaterloo.ca/tcde-smdb/.

I'm delighted to see the this special edition of the bulletin in print, and extend my thanks and appreciation to Prof. Anastassia Ailamaki of Carnegie Mellon University, and David Lomet of Microsoft Research for their efforts. I look forward to welcoming you to the ICDE 2007 Workshop on Self Managing Database Systems in Istanbul.

Sam Lightstone
DB2 Universal Database, IBM Canada LTD
Toronto, Canada

Letter from the Special Issue Editor

This special edition of the IEEE Data engineering Bulletin is devoted to Self-Managing Database Systems, an extremely important research topic for managing today's ever-growing datasets. Since the 1999 edition on Self-Tuning Databases[2], automatic information management has become increasingly critical not only for database systems, but also for storage systems as well as middleware and applications. In all these domains there is a common trend: hardware and software depreciates over time, while the human administration costs increase. It is typical for a company today to spend $20 on human administration and maintenance for every $1 spent on hardware and software. There is currently a vast research community working on the subject of automating data management tasks, varying from automatic configuration and organization to self-maintenance and healing.

The invited articles in this bulletin demonstrate the tremendous advancement in the field. The first three articles reflect the commercial state-of-the-art: an overview of the automated database management features in three leading commercial products (Microsoft's SQL Server, IBM's DB2, and Oracle). Research teams from these companies describe tools integrated in their products that automate physical database design as well as other daunting tuning tasks. The next article by Pautasso et.al. describes the design and development of an autonomic workflow engine, which can be used to compose large-scale system services. Moving on to system resource management, Narayanan et.al. present the design of a resource advisor which answers questions such as "how would the performance of my OLTP application change if doubled the main memory?". The last two articles reflect efforts on automating storage management: Qiao et.al. describe PULSTORE, an analytic framework to transparently alter storage configuration to satisfy a time-varying I/O workload while maintaining QoS guarantees. Finally, a large team from the Parallel Data Lab at Carnegie Mellon University report on the design and implementation of a self-* storage system, which can manage itself using performance predictions, thereby greatly simplifying tuning automation.

I would like to thank Sam Lightstone, Chair of the IEEE Data Engineering Workgroup on Self-Managing Database Systems, who kindly agreed to foreword this issue with a letter summarizing the current trends in the field and the workgroup's activities. I also cordially thank David Lomet and all the authors who graciously contributed their time and effort to make this special edition a reality. I hope that you will enjoy reading it.

Anastassia Ailamaki
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

[2]IEEE Data Engineering Bulletin Volume 22, Number 2, June 1999 Special Issue on Self-Tuning Databases and Application Tuning

AutoAdmin: Self-Tuning Database Systems Technology

Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, Vivek Narasayya
Microsoft Research

1 Introduction

The AutoAdmin research project was launched in the Fall of 1996 in Microsoft Research with the goal of making database systems significantly more self-tuning. Initially, we focused on automating the physical design for relational databases. Our research effort led to successful incorporation of our tuning technology in Microsoft SQL Server and was subsequently also followed by similar functionality in other relational DBMS products. In 1998, we developed the concept of self-tuning histograms, which remains an active research topic. We also attempted to deepen our understanding of monitoring infrastructure in the relational DBMS context as this is one of the core foundations of the "monitor-diagnose-tune" paradigm needed for making relational DBMS self-tuning. This article gives an overview of our progress in the above three areas – physical database design, self-tuning histograms and monitoring infrastructure.

2 Physical Database Design Tuning

One great virtue of the relational data model is its data independence, which allows the application developer to program without worrying about the underlying access paths. However, the performance of a database system crucially depends on the physical database design. Yet, unlike widespread use of commercial tools for logical design, there was little support for physical database design for relational databases when we started the AutoAdmin project. Our first challenge was to understand the physical design problem precisely. There were several key architectural elements in our approach.

Use of Workload: The choice of the physical design depends on the usage profile of the server, so we use a representative workload (defined as a set of SQL DML statements such as SELECT, INSERT, DELETE and UPDATE statements), as a key input to the physical design selection problem. Optionally, a weight is associated with each statement in the workload. A typical way to obtain such a workload is to use the monitoring capabilities of today's DBMSs that allow capture of SQL statements, which execute on the server over a representative time period to a trace file (see Section 4). In some cases, a representative workload can be derived from an appropriate organization or industry specific benchmark.

Optimizer "in the loop": When presented with a query, the database query optimizer is responsible for deciding which available physical design structures (e.g. indexes or materialized views) to use for answering the query. Therefore, it is crucial to ensure that the recommendations of an automated physical design selection

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

tool are in-sync with the decisions made by the optimizer. This requires that we evaluate the goodness of a given physical design for a database using the same metric as optimizer uses to evaluate alternative execution plans, i.e., the optimizer's cost model. The alternative approach of building an external, stand-alone simulator for physical design selection does not work well for both accuracy and software engineering reasons (see [13]). This approach was first adopted in DBDSGN [19], an experimental physical design tool for System R.

Creating "what-if" physical structures: The naïve approach of physically materializing each explored physical design alternative and evaluating its goodness clearly does not scale well, and is disruptive to normal database operations. A "what-if" architecture [14] enables a physical design tool to construct a configuration consisting of existing as well as *hypothetical* physical design structures and request the query optimizer to return the best plan (with an associated cost) if the database were to have that configuration. This architecture is possible because the query optimizer does not require the presence of a fully materialized physical design structure (e.g., an index) in order to be able to generate plans that use such structure. Instead, the optimizer only requires *metadata* entries in the system catalog and the relevant statistics for each hypothetical design structure, often gathered via sampling. The server extensions that support such a "what-if" API are shown in Figure 1.

These three foundational concepts allow us to precisely define physical design selection as a *search* problem of finding the best set of physical structures (or *configuration*) that fits within a provided storage bound and minimizes the optimizer-estimated cost of the given workload.

Challenges in Search for a Physical Design

We now outline the key challenges for solving the physical design selection problem as defined above.

Multiple physical design features: Modern DBMSs support a variety of indexing strategies (e.g., clustered and non-clustered indexes on multiple columns, materialized views). Furthermore, indexes and materialized views can be horizontally partitioned (e.g., range, hash partitioning). The choices of physical design features strongly *interact* with each other [4]. Thus, an integrated approach that considers all physical design features is needed, which makes the search space very large.

Interactions due to updates and storage: The physical design recommendation for a query may have a significant impact on the performance of other queries and updates on the database system. For example, an index recommended to speed up an expensive query may cause update statements to become much more expensive or may reduce the benefit of another index for a different query in the workload. Thus, good quality recommendations need to balance benefits of physical design structures and their respective storage and update overheads [15].

Scaling Challenges: Representative workloads generated by enterprise applications are often large and consist of complex queries (e.g., in the order of hundreds of thousands of statements). Furthermore, the database schema and the table sizes themselves can be large. Thus, to be usable in an enterprise setting, physical design tuning tools must be designed to scale to problem instances of this magnitude.

Key Ideas for Efficient Search

Table Group and Column Group pruning using frequent itemsets: The space of physical design structures that needs to be considered by a physical design tool grows exponentially with the number of tables (and columns) that are referenced in any query. Therefore it is imperative to prune the search space early on, without compromising the recommendation quality. Often, a large number of tables (and columns) are referenced infrequently in the workload. For many of them, any indexes or materialized views on such tables (and columns) would not have a significant impact on the cost of the workload. Therefore, we use a variation of frequent itemset techniques to identify such tables (and columns) very efficiently and subsequently eliminate these from consideration [2, 4].

Workload compression: Typical database workloads consist of several instances of parameterized queries. Recognizing this, and appropriately compressing the input workload [10] can considerably reduce the work done during candidate set generation and enumeration.

Candidate set generation: A physical structure is considered a candidate if it belongs to an optimal (or close to optimal) configuration for at least one statement in the input workload. The approach described in [13] generates this candidate set efficiently. A more recent work [5] discusses how to instrument the optimizer itself to efficiently generate the candidate set.

Merge and Reduce: The initial candidate set results in an optimal (or close-to-optimal) configuration for queries in the workload, but generally is either too large to fit in the available storage, or causes the updates to slow down significantly. Given an initial set of candidates for the workload, the *Merge* and *Reduce* primitives [15, 2, 5, 6] augment the set with additional indexes and materialized views that have lower storage and update overhead while sacrificing very little of the querying advantages. For example, if the optimal index for query Q_1 is (A, B) and the optimal index for Q_2 is (A, C), a single "merged" index (A, B, C), while sub-optimal for each Q_1 and Q_2 can be optimal for the workload if there is insufficient storage to build both indexes.

Enumeration: The goal of enumeration is to find the best configuration for a workload from the set of candidates. As described earlier, the choice of various structures interacts strongly with each other and this makes the enumeration problem hard. We have explored two alternative search strategies: top-down [5] and bottom-up [13, 2] enumeration, each of which has relative merits. The top-down approach can be efficient in cases where the storage bound is large or the workload has few updates. In contrast, the bottom-up search can be efficient in storage constrained or update intensive settings.

Customizing Physical Design Selection

Exploratory "what-if" analysis: Experienced DBA's often want the ability to propose different hypothetical physical design configurations and explore the impact of the proposed configuration for a given workload (which statements speeded up or slowed down, and by how much etc.) [14, 3].

Incremental refinement of physical design: Changes in data statistics or usage patterns can introduce redundancies and may make a very well tuned physical design inappropriate over time. However, physical design changes can have a significant overhead (e.g., existing query plans can get invalidated, which is not be desirable on production servers). In such cases, one would like to compact the existing physical design in an incremental manner, without significantly sacrificing performance. Reference [6] describes such a technique that starts from the initial configuration, and progressively refines it using the *merge* and *reduce* primitives until some property is satisfied (e.g., the configuration size or its performance degradation meets a pre-specified threshold).

Constrained physical design selection: DBAs often need the ability to specify a variety of constraints on the physical design (e.g, which tables to tune, or which existing indexes to keep) [3]. An important constraint that impacts manageability is that indexes are partitioned identically as the underlying table (i.e. *aligned*).

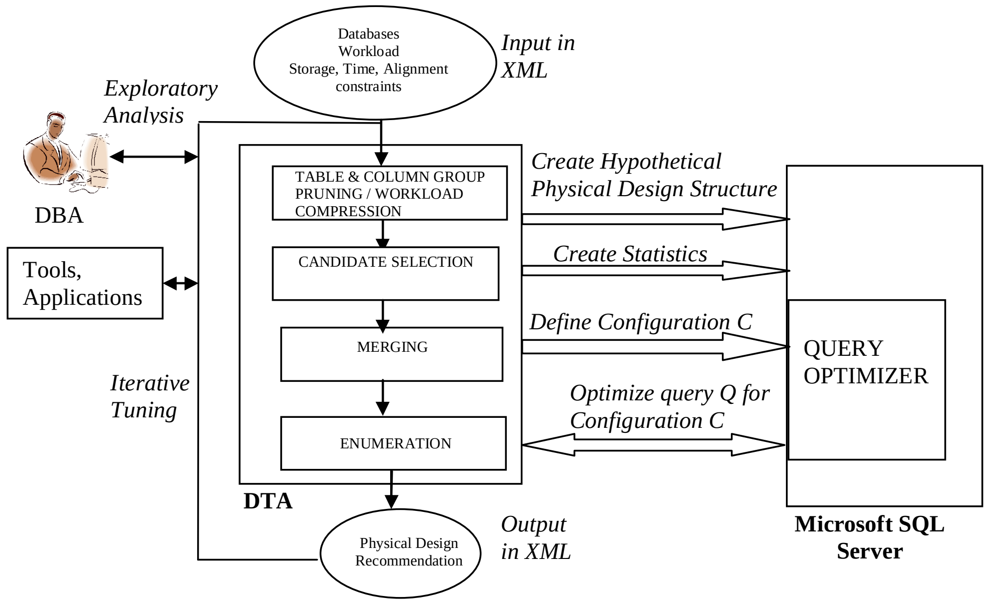


Figure 1: Overview of the Database Engine Tuning Advisor (DTA).

In this case, common operations like per-partition backup/restore and data load/remove become much easier. However as shown in [4], database system performance can be significantly impacted by the choice of a specific partitioning strategy. Therefore, a DBA may still need to find the best physical design where all the indexes are required to be aligned.

When to Invoke Physical Design Tuning?

Data and workload characteristics can change over time. The usage modes described thus far assume that the DBA knows when to invoke a physical design tuning tool. Since physical design tuning has an associated overhead (and impacts the underlying database engine performance), it is useful to identify a priori whether or not physical design tuning on the database can significantly improve performance. We built a lightweight tool, called *Alerter*, that identifies when the current physical design has the opportunity to be improved. It does not make any additional optimizer calls; rather it piggybacks on the optimizer when the latter generates the query plan (see [7] for details).

Product Impact

The AutoAdmin research work on physical database design tuning resulted in a tool called the Index Tuning Wizard that shipped with Microsoft SQL Server 7.0 in 1998. The tool was the first of its kind among commercial relational database systems. It used many of the key building blocks described above, including the “what-if” architecture (which required extending the SQL Server optimizer), candidate selection, merging and bottom-up enumeration. Microsoft added support for materialized (or indexed) views in the SQL Server 2000 release. Our work on table group pruning and view merging in Index Tuning Wizard enabled us to provide efficient, integrated recommendation for indexes and materialized views. In SQL Server 2005, our work resulted in a tool, called the Database Engine Tuning Advisor (DTA) that replaced and significantly expanded the scope and usability of Index Tuning Wizard. DTA can provide integrated recommendations for range partitioning in addition to indexes and materialized views. Furthermore, it incorporates some of the new usage modes described above such as: (a) partitioning “alignment” constraint (b) exploratory “what-if” analysis. An overview of the architecture of DTA is shown in Figure 1 (see [3] for more details).

3 Self-Tuning Histograms: Exploiting Execution Feedback

Query optimization in DBMSs has traditionally relied on single column histograms to estimate selectivity values. Despite the fact that several proposals for multi-dimensional histograms have been put forward [21, 22, 20] to address obvious inaccuracies in estimating multiple selection conditions on different columns using single-column histograms, none is presently supported among leading relational database products. This is partly explained by the fact that like single-column histograms, multi-dimensional histograms implicitly assume that all multi-dimensional queries are equally likely. This is rarely true in practice and this incorrect assumption has much more adverse impact on multi-dimensional histograms than single column histograms.

The above observation led the AutoAdmin team to propose the notion of a *self-tuning histogram*. The key intuition in self-tuning histogram is to use the query workload as a key driver in defining the structure of the multi-dimensional histogram. Self-tuning histograms are incrementally built up by exploiting workload information and query execution feedback. Intuitively, we exploit query workloads to zoom in and spend more resources in heavily accessed areas while allowing some inaccuracy in the rest. We exploit query execution feedback in a truly multidimensional way to: (i) identify promising areas to enclose in histogram buckets, (ii) detect buckets that do not have uniform density and need to be "split" into smaller and more accurate buckets, and (iii) collapse adjacent buckets that are too similar thus recuperating space for more critical regions. Self-tuning histograms can gracefully adapt to changes in the data distribution they approximate, without the need to periodically rebuild them from scratch. Additionally, some data sources might only expose their values through queries (e.g., web-services), and thus traditional techniques, which require the complete data set to proceed, are of little or no value.

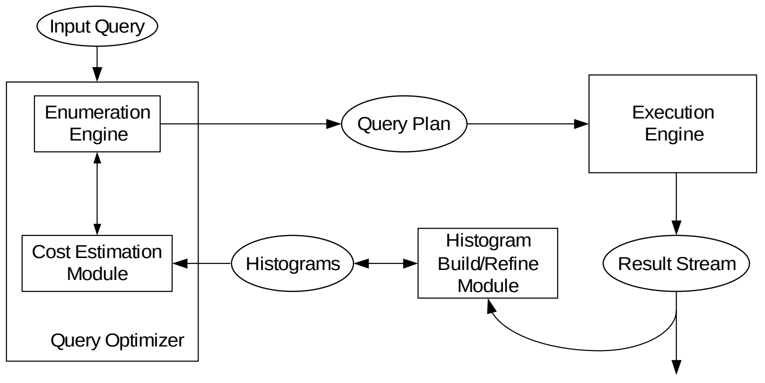


Figure 2: Self-tuning histograms.

Figure 2 shows schematically how to maintain self-tuning histograms. For each incoming query, the optimizer exploits existing histograms and produces an execution plan. The resulting execution plan is then passed to the execution engine, where it is processed. A build/refine histogram module monitors the query execution feedback and diagnoses whether the relevant buckets are accurate enough. If that is not the case, the corresponding histogram buckets are tuned so that the refined histogram becomes more accurate for future similar queries.

To define a self-tuning histogram, we need to address three key issues: the multidimensional structure that holds histogram buckets, the monitoring component that gathers query execution feedback, and the tuning procedures that restructure histogram buckets. Our first attempt at self-tuning histograms was STGrid histograms [1] where we (i) greedily partition the data domain into disjoint buckets that form a grid, (ii) monitor query results by aggregating coarse information that is used to refine bucket frequencies, and (iii) periodically restructure buckets by merging and splitting rows of buckets at a time (to preserve the grid structure). Later, in reference [8] we introduced STHoles histograms, which use a novel partitioning strategy that is better suited to represent multi-dimensional structures. Specifically, some buckets can be completely included inside others. This way, we implicitly relax the requirement of rectangular regions while keeping rectangular bucket structures. STHoles histograms gather query execution feedback on a per-bucket level, and can refine individual buckets for better accuracy.

While STGrid focuses on efficiency by monitoring aggregated coarse information, STHoles focuses on accuracy at the expense of a more heavyweight monitoring. In many cases, STHoles histograms are more accurate for the expected workload than alternative techniques that require multiple scans over the whole data set. Recently, reference [23] introduces ISOMER, a variation of STHoles histograms that balances accuracy and monitoring overhead. By using a lightweight monitoring mechanism and applying the maximum entropy principle to refine buckets, ISOMER provides a good middle-ground between the faster but less accurate STGrid histograms and the relatively slower but more accurate STHoles histograms.

4 Monitoring the Database Server

Our goal for making the database systems self-tuning requires the ability to observe and analyze the “state” of the server often over a period of time. The previous two sections of this article point to the importance of capturing the query workload as well as execution feedback. Despite the clear benefit of monitoring the state of the server, database systems have traditionally been limited in their support for monitoring. Therefore, the AutoAdmin project considers this an important area of further exploration. In this section, we summarize some of our progress in this area, after reviewing the current state of the art for DBMS monitoring infrastructure.

Today’s relational database systems support two basic monitoring mechanisms in addition to those provided by the operating system. The first exposes a *snapshot of counters* that captures current database state. These counters can be obtained at any point in time by polling the server via system defined views/functions. For example, in Microsoft SQL Server 2005, these snapshots can be obtained by Dynamic Management Views or *DMVs* (`www.msdn.microsoft.com`). The second mechanism, which we refer to as *event recording*, allows system counters to be logged to a table/file whenever a pre-specified event occurs. For example, in Microsoft SQL Server 2005, the Profiler provides such event recording functionality. Both these mechanisms form the basis of diagnostic and tuning tasks (e.g., DTA uses Profiler, DMVs can be used to diagnose performance bottlenecks such as excessive blocking). We now highlight two pieces of work on the monitoring infrastructure.

Query Progress Estimation

Consider the problem of *estimating progress* of a currently executing query. An estimate of the percentage completed for a query is useful to DBAs for many reasons (e.g., to decide whether to kill the query to free up resources and for admission control decisions). However, this problem is significantly more challenging than the common problem of measuring progress of a file download. In particular, unlike the file download example, it is not always possible to ensure that *estimated* progress monotonically increases over time without compromising accuracy.

It is important to recognize that since query progress estimation will be used for monitoring, such estimation must be computationally lightweight and yet be able to capture progress at fine granularity (being accurate at only at 0% and 100% is trivial and uninteresting). In solving this monitoring problem, our first challenge is to define a meaningful metric for work done by a query. For example, the count of answer tuples is not a good indicator since there could be one or more blocking operators in the execution plan. Next, we must provide estimators that rely on the model of work done for doing robust, lightweight estimation.

Metric for Work done: The requirements for modeling the work done for progress estimation are different from those of the query optimizer, which uses its cost model to compare alternative execution plans. Estimation of progress requires the ability to incorporate execution feedback and progressively refine the a priori estimation, obtained initially using the optimizer's model. These considerations lead us to a different metric for work, as explained in [16]. We observe that the operators in a query execution plan are typically implemented using a demand-driven iterator model, where each physical operator in the execution plan supports Open(), Close() and GetNext(). We model the work done by the query as the total number of GetNext() calls executed in the plan, which can satisfy the requirements of a progress estimator mentioned above.

Estimators: The above metric for work leads to the natural definition of an idealized measure for progress as $\sum K_i / \sum N_i$, where K_i is the number of GetNext() calls executed by operator Op_i thus far, and N_i is the total number of GetNext() calls that will be executed by operator Op_i when the query execution is complete. While K_i is easily measured as the query is executing, this is not so for N_i. Thus, the key challenge is to estimate N_i as accurately as possible *while the query is executing*. The work in [11] analyzes the characteristics of the progress estimation problem from the perspective of providing robust, worst-case guarantees. Despite the fact that we have to contend with a negative result in the general case, for many common scenarios it is possible to design effective progress estimators with bounded error. For example, if we assume input tuples arrive in random order, then measuring progress at the leaf nodes that "drive" the execution of the pipeline by supplying tuples to the other nodes (e.g, table or index scans), can provide robust estimation of progress for the entire query [16, 11].

SQLCM: A Continuous Monitoring Infrastructure

Beyond ensuring that we have the right plumbing to monitor status of the server (e.g., query progress monitoring), another key challenge is that of tracking and aggregating changes in one or more selected counters over time, aggregating from multiple counters that are being monitored, or a combination of both. For example, consider the task of detecting instances of a stored procedure that are 3 or more times slower than the historical average execution time of the stored procedure. If we use event recording, then a very large volume of monitored data needs to be written out by the server (all stored procedure completion events). On the other hand, if we use the mechanism of repeatedly polling the server using DMVs, we could compromise the accuracy of answers obtained if we do not poll frequently enough (i.e., miss outliers). If instead, we poll too frequently, then we may impose significant load on the server. Thus, neither of the prevalent mechanisms provides adequate support for handling the above task – what we need is a lightweight server-side mechanism to aggregate events generated by the monitored counters (also referred to as *probes* in this section).

These requirements led us to build the SQLCM prototype (Figure 3) [12], with the following characteristics. First, it is implemented inside the database server. Second, monitoring tasks can be specified to SQLCM in a declarative manner using a simple class of Event-Condition-Action (ECA) rules. A rule implicitly defines what conditions need to be monitored (e.g., an instance of a stored procedure executes 3 times slower than the average instance, a statement blocks others for more than 10 seconds) and what actions need to be taken (e.g., report the instance of the stored procedure to a table, cancel execution of the statement). Third, the monitored information can be automatically grouped and aggregated based on the ECA rule specifications. This grouping and aggregation can be done very efficiently using an in-memory data structure called the lightweight aggregation table (LAT). Consequently, the volume of information that needs to be written out by the server is small, thus dramatically reducing the overheads incurred on the server by the monitoring tasks. SQLCM only incurs monitoring overhead that is necessary to implement currently specified rules (see details in [12]).

5 Conclusion

As part of the AutoAdmin research project, we have had the opportunity to address several significant challenges that relate to endowing the relational database system with increased self-tuning capabilities. These self-tuning capabilities rely on a monitoring infrastructure and leverage that to build specialized diagnostic and

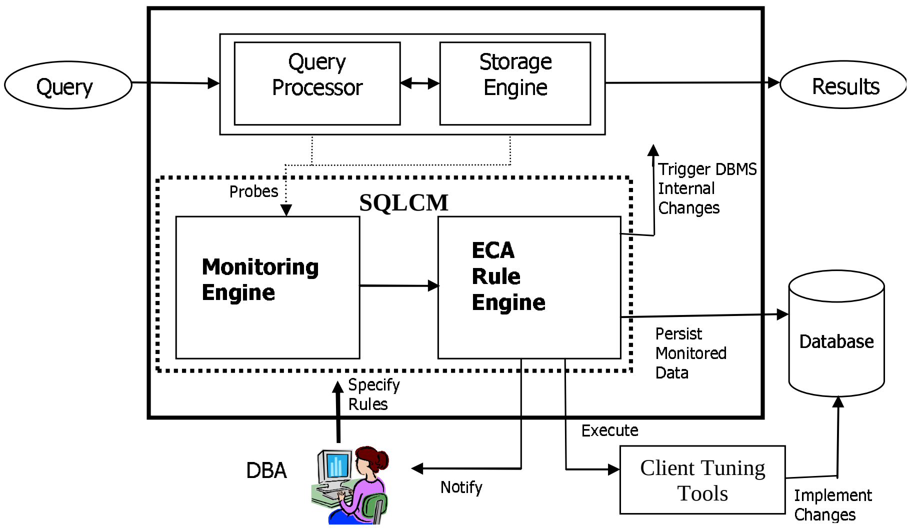


Figure 3: Architecture of SQLCM.

tuning capabilities that are appropriate to the task at hand. Thus, they conceptually share a common *"monitor-diagnose-tune"* pattern. As new queries are executed, the DBMS internally monitors and keeps information about the workload. After a triggering condition happens (e.g., a fixed amount of time, an excessive number of recompilations, significant database updates), the diagnostics component is launched automatically and evaluates the situation quickly. After the lightweight diagnostics, if it is determined that the database needs to change, a tuning component proceeds to recommend/incorporate changes for better performance. The diagnostics and tuning components are typically specific to each task. However, the monitoring component can be shared by multiple "vertical" diagnose-tuning components. Due to lack of space, we have only highlighted a few selected aspects of the AutoAdmin project. Information about other work done in the AutoAdmin project can be found at `research.microsoft.com/dmx/AutoAdmin`.

Since launching of the AutoAdmin effort, there has been increased awareness of the need to reduce the total cost of ownership of database systems and several initiatives in other research groups and database vendors have helped contribute to the development of self-tuning technology [9, 18]. Finally, our experience over the last decade has also convinced us that today's relational database architecture may in fact stand in the way of robust self-tuning capability. Specifically, recognizing the trade-off between adding features and providing self-tuning capability requires careful thinking [17].

Acknowledgments

Over the years, the AutoAdmin project has received significant contributions from current and past project members. Manoj Syamala was a key member of the development team of DTA for Microsoft SQL Server 2005. Christian König was a significant contributor to the SQLCM project. Raghav Kaushik and Ravi Ramamurthy led our work on the query progress estimation problem. AutoAdmin has also greatly benefited from the interns and visitors who worked with us diligently. We thank Ashraf Aboulnaga, Ashish K. Gupta and Gerhard Weikum for their contributions to AutoAdmin work referenced in this article.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM International Conference on Management of Data*, 1999.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[6] N. Bruno and S. Chaudhuri. Physical design refinement: The "Merge-Reduce" approach. In *International Conference on Extending Database Technology (EDBT)*, 2006.

[7] N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32th International Conference on Very Large Databases (VLDB)*, 2006.

[8] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2001.

[9] S. Chaudhuri, B. Dageville, and G. M. Lohman. Self-managing technology in database management systems (tutorial). In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[10] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing sql workloads. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[11] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. Can We Trust Progress Estimators For SQL Queries? In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[12] S. Chaudhuri, A. C. König, and V. R. Narasayya. SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2004.

[13] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, 1997.

[14] S. Chaudhuri and V. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD)*, 1998.

[15] S. Chaudhuri and V. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.

[16] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL Queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[17] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[18] S. Chaudhuri and G. Weikum. Foundations of automated database tuning (tutorial). In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[19] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 1988.

[20] D. Gunopulos et al. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[21] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1988.

[22] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, 1997.

[23] U. Srivastava et al. ISOMER: Consistent histogram construction using query feedback. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

Making DB2 Products Self-Managing: Strategies and Experiences

Sam Lightstone[2] Guy M. Lohman[1] Peter J. Haas[1] Volker Markl[1]
Jun Rao[1] Adam Storm[2] Maheswaran Surendra[3] Daniel C. Zilio[2]

[1] IBM Almaden Research Ctr
San Jose, CA, USA
{phaas,lohmang,marklv,junrao}@us.ibm.com

[2] IBM Toronto Lab
Markham, Ontario, Canada
{light,storm,zilio}@ca.ibm.com

[3] IBM T.J. Watson Research Ctr
Hawthorne, NY, USA
suren@us.ibm.com

Abstract

This paper evaluates the impact of the DB2 Autonomic Computing project at the IBM Toronto Software Lab, Almaden Research Center, and Watson Research Center. It describes the key ideas behind the many self-managing features added to the IBM®DB2®for Linux®, UNIX®, and Windows®products, and evaluates the degree to which these features have been accepted by the DB2 user community. We offer lessons learned from this experience, our conclusions, and future directions for self-managing databases.

1 Introduction

Over the last three decades, database research and development has achieved remarkable improvements in functionality and performance, aided both by the emergence of standards for the SQL language and by the TPC family of benchmarks, which fueled competition. However, these features and performance have come at the price of skyrocketing complexity, particularly the complexity of database administration. Researchers focused on languages such as SQL to provide a simple, declarative interface for application developers, but administrative interfaces received considerably less attention until quite recently. Simultaneously, improvements in the density of chips and disk storage have drastically reduced the cost and increased the capacity of hardware, while skilled database administrators (DBAs) have become increasingly rare and expensive. As a result, the total cost of ownership of modern database systems is now dominated by the cost of people, not hardware or software. All of these trends prompted efforts in the last few years to try to make existing database products easier and cheaper to manage, mostly by adding mechanisms to automate previously manual administrative tasks, or at least to provide guidance to DBAs.

This paper evaluates the impact of one such effort, the DB2 Autonomic Computing project. We summarize the key ideas that fueled the many autonomic features that the project contributed to the DB2 products, evaluate the degree to which customers have accepted those features, and relate the lessons learned. This project was initially inspired by the early development of an Index Advisor that first appeared in V6 of DB2 Universal Database™(DB2 UDB) for Linux, UNIX, and Windows [15]. The DB2 Autonomic Computing (DB2 AC) project was subsequently formed in early 2000 as a joint effort between the IBM Almaden Research Center and the IBM Toronto Software Lab, and later the Watson Research Center. Based upon requirements

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

interviews with over 120 customers, an ambitious plan was developed for making DB2 self-configuring, self-healing, self-optimizing, and self-protecting [6, 8, 7]. The resulting autonomic features added to DB2 over several releases have been described in previous papers [2], a complete bibliography of which can be found at http://www.almaden.ibm.com/software/projects/autonomic/references/autonomic_ref.shtml. A good overview of autonomic computing (AC) in DB2 can be found online at the DB2 Magazine site, at http://www.db2mag.com/epub/autonomic.

To understand the context of the DB2 AC project, one must first grasp the constraints under which it operated. The project started with an existing database system that, in response to the competitive environment, had primarily emphasized features and performance, rather than ease of administration. The team didn't have the luxury of building an autonomic system from scratch, but had to retroactively add autonomic functionality. Moreover, the autonomic enhancements needed to be industrial strength and enterprise scale, i.e., we had to develop robust solutions that would work in all environments and would scale to hundred-terabyte databases. For example, moving to one large memory pool would have simplified memory management significantly and reduced the need for configuring individual memory heaps, but a single memory pool would have been vulnerable to a runaway agent over-consuming memory. Finally, we had to support an existing customer set and their expectations, so we had to be very conservative about changing the default behavior. For example, the existing customers were very sensitive to any decreases in performance, and hence we had to be very cautious when adding monitoring overhead. All of these constraints affected our approach and solutions.

The remainder of the paper is organized as follows. The next section summarizes the key ideas underlying the autonomic features that we have added to DB2. Section 3 discusses an evaluation that we performed to determine the extent to which our customers exploited and liked these features. The lessons we have learned from this experience are presented in Section 4, and the final section contains our conclusions and future directions.

2 Key Ideas and Themes

Several key ideas and themes were exploited in our changes to make DB2 more autonomic:

Low-impact collection of accurate system data. We developed and exploited two low-impact methods for automatically obtaining database statistics, information on query and system behavior, etc. The resulting up-to-date and accurate information is used to improve the accuracy of the query optimizer's cardinality model, as well as to enable the system to adjust a variety of operational parameters to improve query-processing efficiency. The first method involves opportunistic monitoring of various information sources during query execution; the trick is to focus on measurements that can be collected with very low overhead. For example, DB2 simply counts the actual number of rows processed by each run-time operator during query execution. These cardinality actuals are then compared to the optimizer's estimates, in order to detect significant variations from the optimizer's cardinality model. Such comparisons can be made after the query has completed, as in the LEO LEarning Optimizer [12], or the comparisons can potentially be made dynamically, thereby enabling a Progressive OPtimization (POP) system [10] to decide whether to re-optimize a query plan while the plan is running. A second example is Self-Tuning Memory Manager [13], which collects minimal information on hit ratios (fraction of requested pages that reside in the various buffer pools) to better determine the best allocation of available memory among the competing pools. A third example of opportunistic data collection is the DB2 Health Center, which periodically "takes the pulse" of the system and raises alerts if certain pre-set thresholds are exceeded. The second method for low-impact monitoring is database sampling. For example, we exploit sampling to augment the traditional single-column statistics with multivariate statistics in the DB2 product. Such statistics allow the optimizer to detect statistical correlations between columns and thereby avoid bad estimates due to erroneous independence assumptions. The CORDS (CORrelation Detection by Sampling) system explicitly searches for correlations among all pairs of columns before queries run, by sampling the database [3]. CORDS is less efficient than LEO, because LEO selectively pinpoints only those correlations that cause significant estimation errors in the actual query workload [9]; on the other hand, CORDS complements LEO by providing accurate estimates during LEO's initial learning period, and when LEO is faced with unforeseen queries. The Design Advisor [16, 17] samples the data as needed to confirm or disprove correlations pertinent to the cardinality estimates used for determining which set of materialized views to maintain and which Multi-Dimensional Clustering (MDC) organizations to adopt [5]. The sampling approach provides the Design Advisor with very accurate information on which to base its decisions, which is especially important in data warehousing scenarios.

Feedback. The notion of feedback control loops is not new, but the application to query planning was definitely a novel development. The idea is to use opportunistically-gathered information, as described above, to automatically and dynamically adjust the DB2 engines behavior over time, in repsonse to changes in the data or the operating environment. For example, LEO uses actual and estimated cardinalities to compute correction factors that are used to improve subsequent cardinality estimates, in a perpetually self-correcting loop. Similarly, Self-Tuning Memory Manager uses feedback on hit ratios as described above to dynamically adjust the sizes of the buffer pools. Another form of feedback loop in DB2 is embodied by "throttled" daemons, discussed below.

Re-using Optimizer as a "What if?" tool. Recognition that the query optimizer's model of system execution could be re-used as a "What if?" tool was one of the earliest and most significant "aha!" moments of our project. That is, instead of merely using the optimizer to predict query performance in an existing logical and physical configuration (i.e., existing indexes, materialized views, clustering, partitioning, memory, etc.), the optimizer can be used to evaluate hypothetical, alternative configurations to provide guidance on potentially advantageous reconfigurations. Thus we can create virtual "What if?" objects such as virtual indexes, materialized views, and table partitionings or clusterings, and then track the resulting properties of the query plan as it exploits these virtual objects. This approach has a number of key advantages. First, we can exploit the existing, carefully crafted mechanisms for composing and comparing the cost and properties of plans. Also, we don't have to build and maintain separate cost models, thereby saving much effort. Finally, we avoid the embarrassing situation in which a separate model recommends a change and the optimizer's model, for obscure reasons, disagrees, confusing the customer. If the optimizer as "What if?" tool recommends a plan using a virtual index, then it is very likely that the optimizer as plan selector will also pick the same plan once the index is actually created, because the same model is used in both situations.

Heuristics, new models. Despite the usefulness of the optimizer's cost model, we found that in some cases alternative, novel models or heuristics were needed. The optimizer's model can be too detailed and too focused on picking a plan for a specific query to yield good values for high-level system parameters that interact with each other and affect many queries simultaneously. Thus, we developed new high-level models to choose the 40 or so configuration parameters that most affect performance, including major pools of main memory such as the shared memory used for sorts (sortheap) [4]. These models are less detailed than the optimizer's cost model, but more realistically consider the system-wide interaction of multiple queries and mathematically embodies the real world experience of our performance team gained by running customer and industry-standard benchmarks. A more dynamic and detailed model to deal only with all the memory pools was later developed to holistically make the hard trade-offs between competing needs for memory, while avoiding the dangers of a single memory pool that would permit a single, runaway query to hog system resources to the detriment of others.

"Throttled" daemons. Our early interviews with DBAs revealed that much of their time was spent scheduling and performing batch operations that required large blocks of time, such as performing backups, reorganizations, and database statistics collection. In today's 24×7 world, those blocks of time were being shrunk to zero, so the tasks performed in them had to be executed concurrently but unobtrusively with regular workloads. What was needed was a generic background daemon that would make assured progress on such operations using spare cycles in periods with relatively low (but not nonexistent!) workload demands, and would back off as workload demands increased. The solution we implemented was a generic mechanism for "throttling" processes, using classical control theory to determine the workload-dependent length of time that such a process "sleeps" before it "wakes" and achieves progress on its task [11]. By performing batch processes as a continuous background process, the need for scheduling and reserving large blocks of down time is obviated, unused cycles are efficiently exploited (achieving greater overall system utilization), and the entire process can be automated.

Works out-of-the-box. Reducing the number of decisions necessary for getting started reduces the all-important "time to value," the time between the decision to buy a system and when it begins producing value. Moreover, by automating many of the processes that customers often neglected unless they were experts, we both improve their experience and decrease our service costs. For example, poor optimizer behavior resulting from out-of-date or nonexistent database statistics sometimes stemmed from the fact that new users were unaware of the need to execute the RUNSTATS statistics-collection utility. By automating and throttling RUNSTATS by default [1], the "out-of-the-box" experience of customers has been significantly enhanced. Similarly, the Health Center is pre-configured to collect its health metrics and raise alerts based upon pre-set thresholds. All the installer needs to provide is an address to send the notifications. The DBA could of course subsequently modify the thresholds, but such intervention is not required in order to become operational, and usually isn't needed, because the thresholds are based upon a universal metric — the percent of the resource being consumed.

Progressively more autonomic. Many of the augmentations we made to DB2 required a lot of hard work, understanding the rationale for the existing "knobs", designing an automated scheme to robustly "get it right" (almost) all of the time, and implementing and fully testing the new mechanism, all in the context of regular product release cycles. Frequently the work had to be broken up into smaller pieces that could be released in a timely manner, rather than waiting through multiple releases before the fully automated scheme could emerge full-blown. Take for example the setting of configuration parameters. The first (inglorious but crucial) step was only to make them dynamic, so that changes of those parameters did not require restarting DB2 for the new values to take effect. For parameters such as buffer pools, this change was non-trivial, because shrinking a buffer pool could force out pages prematurely. The next step was our Configuration Advisor, which the DBA had to invoke to set almost 40 detailed configuration parameters, using seven high-level parameters about the system (provided by the DBA) and some equations that summarized the complicated interactions of the 40 configuration parameters. This advisor first appeared in Version 7.2 of DB2 UDB, and was enhanced in Version 8. Finally, in DB2 9 (which was released in late July 2006), we fully automated and dynamically adjusted the settings for many of these configuration parameters that controlled memory heaps and buffer pools with the Self-Tuning Memory Manager. A benefit of this successive roll-out of features was the insight that we were able to develop, based on experience and feedback, about which of these parameters really mattered most to performance.

3 Evaluation

During the fall of 2005 and winter of 2006, IBM conducted a review of the self-managing features in DB2. The goal was to determine the quality and success of these features as of Version 8.2.2, and to identify any necessary refinements for maximizing their impact. Information was gathered through surveys, discussion, and experimentation. Several hundred people were involved, both within and outside IBM, including customers, consultants, and IBMers involved in sales, pre-sales, services, support, and development. In particular, survey data was collected from over a dozen consultants, called the "Gold Consultants," who each work professionally with multiple DB2 accounts. The autonomic features that were evaluated included[1] Automatic Backup, Automatic Reorganization, Automatic Statistics, Automatic Statistics Profiling, Automatic Storage, Configuration Advisor, Design Advisor, Health Monitor, Self-Tuning BACKUP, Self-tuning LOAD, and Utility Throttling.

A number of interesting trends emerged from this evaluationm as summarized in Figure 1. In the figure, we have masked the feature names, referring to them only as features A, B, C, and so forth. For each feature, we have plotted both the consultants' average perceived usefulness of the feature (on a scale from 1 to 10), as well as the standard deviation of those responses. A standard deviation larger than 2.5 indicates that consultants had significantly different views on the value of a feature.

[1]Note that some features discussed in this paper, such as Self-Tuning Memory Manager, CORDS, and Progressive Optimization, were not yet available in Version 8.2.2, and hence were excluded from the survey.

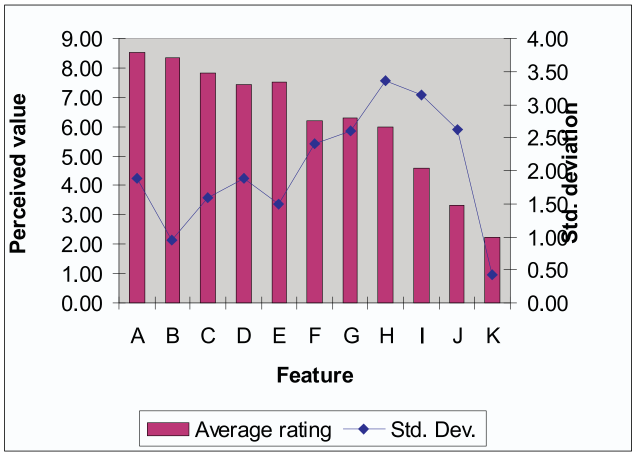


Figure 1: Survey results from DB2 consultants.

Features A, B, and C have the highest average rating for value, as well as relatively low deviation in the opinions. These features were all characterized as being trivially easy to use. Features F, G, H, and I have noticeably large standard deviations in the usefulness ratings. The probable reason for the large variation is that these features were designed predominantly for small to medium-sized business (SMB) markets, where DBA skills are most scarce. Whereas consultants working with small-scale engagements found significant value in the features, consultants who worked primarily with large-enterprise users found these features of little value. In contrast, Feature J was designed for high-end decision support systems. This feature generated very positive feedback for its functionality and potential usefulness, but also engendered frustration over its interface, usability, and platform support requirements. These mixed emotions resulted in the high standard deviation displayed in the figure. Finally, Feature K stands out as having both a low score for perceived value as well as a low standard deviation, implying a lack of success so far. This user reluctance is attributable to the complexity of the feature's interface and its negative impact on data availability.

The survey verified a number of factors related to awareness, adoption, and trust. Anecdotally, our discussions with the Gold Consultants revealed that the majority of our autonomic features were known to most of them, and all were using some of the features on a regular basis. In contrast, fewer features were used by non-consultants, and features enabled by default enjoyed dramatically higher adoption. As of Version 8.2, most of these AC features must be manually enabled, and some features (e.g., Design Advisor) require human expertise. This situation appears to severely hinder adoption by users. In general, the survey showed that

- A feature that needs to be invoked will be used 20 times less often than one enabled by default.
- The *average* user will generally not be aware of any of the system's advanced features.
- The *power* user needs self-managing technology the least, and will therefore benefit the most from those AC features that automatically handle frequent, continuous administrative chores that expert humans are hard-pressed to do themselves; one-time automation and advisors are less valuable to this community.

Trust, not surprisingly, was found to be a significant factor in feature adoption [14]. Several customers strongly requested both better monitoring of DB2 autonomic activity and better insight into the specific decisions that the autonomic components were making in the course of their operation. Furthermore, customers expressed more trust in adaptive technologies than in heuristic-based configuration and tuning. As a result, features such as Automatic Utility Throttling and Self-Tuning Memory Manager are likely to be trusted more than features such as the Configuration Advisor.

4 Lessons Learned

We have learned many lessons in the course of planning, researching, and developing autonomic capabilities in DB2 products. These can be summarized in the following seven guiding principles for making existing systems more self-managing. Keep in mind that autonomic computing is largely a software-oriented discipline that inherits the design goals and requirements of all good software (encapsulation, reliability, reuse, etc.), so most of the following seven principles are, not surprisingly, applicable to software development in general.

Build what users need, not what's cool. Perhaps surprisingly, one of the major challenges to development teams that build AC technology is that designing such systems is too much fun. Although this assertion may seem ridiculous at first glance, the fact is that almost everyone who works on autonomic systems continually desires the excitement and challenge of making the system just a little more adaptive and intelligent. In many cases, the added sophistication is not needed, and only increases the complexity of the code. Indeed, there are many instances in the world of industrial software development in which full-blown complex features have been implemented, when a few simple heuristics would have sufficed.

Always give the user an "out:" features providing system automation must have an OFF switch. Even the best autonomic technology will not work perfectly in all situations. Poor automated decisions can occur either because of an imperfect underlying model of system behavior or because of software defects. Either way, when an AC feature fails, the user must have an option to disable that feature. This is particularly true for mission-critical systems: many DBA managers will actively avoid purchasing autonomic technology that cannot be disabled if necessary. More generally, providing the ability to disable autonomic components helps engender trust in autonomic technology by lowering the risks inherent in its adoption. Such trust is important because, without trust, regardless of how good the technology is, it will not be used.

Features must be on by default in order for the majority of users to exploit them. The vast majority of customers are unaware of the existence of autonomic features (indeed, of most advanced features), and discover such features on an as-needed basis. Ironically, these are the customers who most desperately need autonomic technology. The small group of power users, while most aware of AC features, are the least likely to need them. Enabling AC features by default allows the average user to reap the benefits of the technology with no effort required, while still giving the power user a choice to use or bypass the technology. Otherwise, autonomic technology might suffer the same fate as automatic transmissions in cars (which did not dominate the market until roughly 15 years after their introduction in 1939, and which are still resisted by some customers today): the novices don't know about it and the enthusiasts don't want it.

Never force the user to make a choice that your developers couldn't make. All too often in the software world, a development team, unable to determine a reasonable setting for a parameter that is crucial to system performance, opts to require the user to set the parameter's value. This happens frequently when the correct parameter setting is "it depends." While development schedules may temporarily preclude an autonomic solution to the problem of user configurable parameters, eliminating such parameters must be a key objective for autonomic systems over the long run. The reason is simple: if the development team that designed and coded the system didn't know how to set the parameter, it is almost certain that the vast majority of end users won't, either. Foisting the problems of the development team onto unsuspecting users (in this case, system administrators) is a losing strategy.

AC technology must be evaluated in complex, dynamic real world scenarios. Another negative habit that has become rampant in the industry is the design and evaluation of autonomic solutions around benchmarking systems. Industry-standard benchmarks are frequently used to assess the performance or recoverability of systems. The use of benchmarks is, in fact, a reasonable industry strategy that helps drive competition. However, the vast majority of these benchmarks are extremely well-behaved and static. Development teams often use benchmark systems to evaluate autonomic features because the benchmark system provides a well understood workload and performance baseline against which to pit the talents of a newly created autonomic feature. However, production systems are notoriously more complex and variable over time than benchmark systems. As a result, the success of autonomic technology with benchmark systems, while meaningful, is not sufficient.

Never automatically undo or contradict the explicit choices of administrators or applications. Autonomic systems typically execute a cycle of monitoring, analyzing, planning and execution. The analysis and planning could recommend changes to the system that contradict or replace the deliberate choices of a human administrator or system designer. Ideally, a perfect autonomic system would only recommend changes that were certain to improve on the human choices. In reality, there are several reasons why overriding the deliberate choices of humans is ill-advised. First, the quality of AC technology is not mature enough to ensure that the decision of an AC feature is superior to that of a deliberate human choice. Second, once a human administrator has made a choice, however suboptimal, the system can be reasonably assumed to be in a state acceptable to that human, and incremental (or even dramatic) improvements over the human design probably aren't needed. Third, the choices of human beings are often superior because people are able to observe the system as a whole, whereas any single component within a system cannot do so. If the administrator has taken the time to manually intervene, there are probably good reasons for this decision, even if the autonomic components of the system can't detect them. Thus, it is crucial for autonomic systems to distinguish between system changes made by human operators and those made by the autonomic component itself, so that those changes performed by humans will not be overridden.

Minimize policy and keep it human. Numerous system policy grammars and specifications have been proposed over the past 30 years. Because policies represent the specification by human administrators of knowledge that the system could not glean on its own, they should be largely obviated by autonomic technology. Elimination of the need for policy specification is clearly more than a decade away. What we can safely conclude is that: (1) policies are needed and will be needed for the next several years; (2) policies should represent business objectives that can be described in relatively human terms, indicating what is expected of a system, and not be a conduit for injecting configuration parameters and rules into an autonomic system; and (3)policies require standardization in order to facilitate the combination of system components. Sadly, today, "The nice thing about standards is that there are so many of them to choose from" (attributed to Andrew S. Tanenbaum).

5 Conclusions and Future Directions

The DB2 Autonomic Computing project has had considerable success in developing and incorporating into the DB2 products many powerful technologies to ease the burden of beleaguered DBAs. Overall, our customers generally find these autonomic features very helpful when they know to invoke them or, preferably, the feature is enabled by default. The latter requires that autonomic technologies must engender the trust of DBAs by robustly getting "good enough" results almost all the time and by allowing DBAs to disable them in the event of problems.

Adding autonomic features to an existing complex system is significantly more challenging than designing an entirely new system to be autonomic from day one. While we continue to work on additional AC features to simplify the administration of a DB2 environment, we are also investigating more revolutionary, longer-term approaches that obviate many administrator tasks in an information management appliance. Such an approach requires significant research in a variety of challenging new technologies that are already under investigation within IBM Research and that provide ample opportunity for collaboration with academic researchers, as well.

6 Acknowledgements

This work would not have been possible without many contributions from dedicated team members. Major contributions were made by Ashraf Aboulnaga, Bishwaranjan Bhattacharjee, Yixin Diao, Jessica Escott, Christian Garcia-Arellano, Wook-Shin Han, Randy Horman, Ihab Ilyas, Holger Kache, Mokhtar Kandil, Eser Kandogan, Mick Legare, Alberto Lerner, Kelly Lyons, Dale McInnis, Sujay Parekh, Hamid Pirahesh, Ivan Popivanov, Vijayshankar Raman, Kevin Rose, Aamer Sachedina, Berni Schiefer, Utkarsh Srivastava, Mike Stillger, Gary Valentin, Chun Zhang, and Calisto Zuzarte. Also contributing were Alexander Behm, Benjamin Bertow, Christian Brabandt, Matt Carroll, Fei Chiang, Kitman Cheung, Lee Chu, Jerome Colaco, Stephan Ewen, Marcus Fiess, Liam Finnie, Dengfeng Gao, Joseph L. Hellerstein, Fabian Hueske, Mathew Huras, Markus Kollotzek, Marcel Kutsch, Florian Leybold, Tim Malkemus, Nimrod Megiddo, Jack Ng, Michael Ortega-Binderberger, Sriram Padmanabhan, Denis Ricard, Bryan Smith, Sebastian Speiser, James Teng, Tam Minh Tran, and Scott Walkty.

References

[1] A. Aboulnaga, P. J. Haas, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *VLDB*, pages 1146–1157, 2004.

[2] C. M. Garcia-Arellano, S. Lightstone, G. Lohman, V. Markl, and A. Storm. A self-managing relational database server: Examples from IBM's DB2 Universal Database for Linux Unix and Windows. In *IEEE Trans. Sys. Man Cybernetics*, 2005. Special issue on Engineering Autonomic Systems.

[3] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[4] E. Kwan, S. Lightstone, K. B. Schiefer, A. Storm, and L. Wu. Automatic database configuration for DB2 Universal Database: Compressing years of performance expertise into seconds of execution. In *BTW*, pages 620–629, 2003.

[5] S. Lightstone and B. Bhattacharjee. Automating the design of multi-dimensional clustering tables in relational databases. In *VLDB*, pages 1170–1181, 2004.

[6] S. Lightstone, G. M. Lohman, and D. C. Zilio. Toward autonomic computing with DB2 Universal Database. *SIGMOD Record*, 31(3):55–61, 2002.

[7] S. Lightstone, B. Schiefer, D. Zilio, and J. Kleewein. Autonomic computing for relational databases: the ten year vision. In *IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA)*, 2003.

[8] G. M. Lohman and S. Lightstone. SMART: Making DB2 (more) autonomic. In *VLDB*, pages 877–879, 2002.

[9] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Sys. J.*, 42(1):98–106, 2003.

[10] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[11] S. Parekh, K. R. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang. Throttling utilities in the IBM DB2 universal database server. In *Amer. Control Conf. (ACC)*, 2004.

[12] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning optimizer. In *VLDB*, pages 19–28, 2001.

[13] A. J. Storm, C. M. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2 UDB. In *VLDB*, 2006.

[14] R. Telford, R. Horman, S. Lightstone, N. Markov, S. O'Connell, and G. M. Lohman. Usability and design considerations for an autonomic relational database management system. *IBM Sys. J.*, 42(4):568–581, 2003.

[15] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[16] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

[17] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC*, pages 180–188, 2004.

Oracle's Self-Tuning Architecture and Solutions

Benoit Dageville and Karl Dias
Oracle USA
{Benoit.Dageville,Karl.Dias}@oracle.com

Abstract

Performance tuning in modern database systems requires a lot of expertise, is very time consuming and often misdirected. Tuning attempts often lack a methodology that has a holistic view of the database. The absence of historical diagnostic information to investigate performance issues at first occurrence exacerbates the whole tuning process often requiring that problems be reproduced before they can be correctly diagnosed. Even when the problem root cause is identified, fixing it often requires a very high level of expertise that very few DBA possess. This is especially true for the inherently complex activity of SQL Tuning, requiring a high level of expertise in several domains: query optimization, access design, and SQL design.

In this paper we describe how Oracle overcomes these challenges and provides a way to perform automatic performance diagnosis and tuning. The ability to self-tune is a critical aspect towards building a self-managed database, which was one of the key objectives for the latest version of Oracle, Oracle10g, that was released in early 2004.

1 Introduction

In today's around-the-clock economy, the importance of an efficient and reliable IT infrastructure for the success of an enterprise hardly needs any explanation. As businesses increasingly rely on this infrastructure, system performance becomes more important than ever before. Businesses are building more and bigger databases, and database administrators (DBAs) are expected to take on this ever-increasing load. Hiring highly skilled administrative staff to manage such complex environments results in spiraling management costs, making self-managing technologies a must-have for modern database systems [4].

In this context, being able to effectively analyze system performance is crucial for ensuring good quality of service. Database systems traditionally expose a plethora of measurements and statistics about their operation and it can be hard to get an overall view of what is happening in the system. Identification of the root cause of a performance problem is not easy [10, 3, 2]. It is not uncommon for DBAs to spend large amounts of time and resources fixing performance *symptoms*, only to find that this has marginal effect on system performance. Lack of a holistic view of the database leads to incorrect diagnosis, misdirected tuning efforts and over-configured systems, increasing the total cost of ownership. [9, 3, 8].

Even when the proper methodology for analysis is followed, it is often found that the available data stops short of what is required to fully diagnose the root cause. Lack of adequate statistics is a very common issue because collecting appropriate ones is prohibitively expensive, especially since a very broad class of statistics is required to address a very large spectrum of potential issues. Worse, to be effective, statistics collection must be continuous and enabled by default, since a performance problem can strike any time. Additionaly, statistics need to be persisted since the analysis of a performance issue is often performed long after this issue has occurred.

When appropriate statistics are not available, an option is to reproduce the problem while collecting a larger set of targeted statistics, in the hope that this would be enough to complete the performance diagnosis. In real world, this solution is rarely feasible because it requires a full-scale test system and a way to simulate/reproduce a full-scale workload. This is either impossible to do or far too expensive to be practical.

Recognizing these challenging demands, Oracle 10g introduces a sophisticated self-managing database that automatically monitors, adapts, and fixes itself. This paper provides a overview of Oracle's self-tuning architecture along with a more detailed presentation of two automatic tuning solutions: *Automatic Database Diagnostic Monitor* (ADDM) which automatically diagnoses the bottlenecks affecting the total database throughput and provides actionable recommendations to alleviate them; and the *Automatic SQL Tuning Advisor* which provides comprehensive tuning recommendations for a SQL workload that span query optimization, access path analysis and statement restructuring.

2 Self-Tuning Architecture

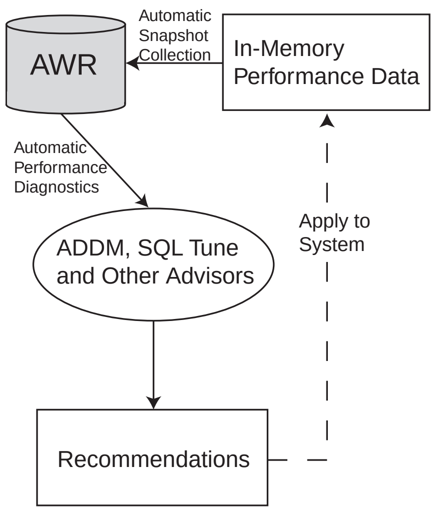

Automatic Snapshot Collection
AWR
In-Memory Performance Data
Automatic Performance Diagnostics
Apply to System
ADDM, SQL Tune and Other Advisors
Recommendations

Figure 1: The Self-Managing Database Framework.

Oracle's tuning framework developed in Oracle10g for self-managing databases is centered around the three phases of the self-managing loop: *Observe*, *Diagnose*, and *Resolve*. This framework enables a comprehensive tuning solution by providing the necessary components. Each component provided by the framework plays a key role in one or more of these phases, and can be broadly classified into two categories: *Statistics Collection and Storage (observe)* which includes components that measure and collect interesting statistics and performance data for current as well as historical analysis and alerting; and *Advisor (diagnose and resolve)* which includes the components that carry out a targeted analysis of the data and work towards optimizing the performance for a given area.

The phases in the self-tuning loop refer to a particular tuning cycle (e.g. total database tuning cycle via ADDM, or a SQL Tuning cycle), and there could be many such tuning cycles occurring concurrently each in different phases. A tuning cycle could contain other tuning cycles. In fact the system is designed with precisely such a hierarchical model in mind; a system-wide top-down throughput based tuning methodology is used wherein ADDM acts as the central advisor that directs further tuning activity in the system by invoking other subsystem specific advisors based on top issues affecting overall throughput. Figure 1 illustrates the relationship between the Statistics Collection and Storage components and the Advisors.

Before we briefly explore each stage in the self-tuning loop, we would like to introduce the key concept of *Database Time* that has enabled us to successfully tackle inter-component database wide tuning.

2.1 Database Time

Traditionally, performance of various subsystems of the database is measured using different metrics. For example, the efficiency of the data-block buffer cache is expressed as a percentage in buffer hit-ratio; the I/O subsystem is measured using average read and write latencies. Using such disparate metrics to find the performance impact of a particular component over the total database throughput is extremely hard, if not infeasible. We addressed this issue in Oracle10g by introducing the concept of *Database Time* or simply *DbTime* in this paper, a new time based measure.

DbTime is defined as the sum of the time spent inside the database processing user requests. It is only a portion of the response time perceived by the user since it does not include time spent in the intervening layers like the network or the middle tiers. It is directly proportional to the number and duration of user requests, and can be higher or lower that the corresponding wall-clock time. It is a measurement of the total amount of work done by the database, and the rate at which the database time is consumed can be thought of as the database load average, similar to the OS load average.

DbTime serves as a common currency for the measurement of a subsystem's performance impact. For example, the performance impact of an under-sized buffer cache would be measured as the total database time spent in performing additional I/O requests that could have been avoided if the buffer cache was larger.

2.2 Observe Phase

This phase is automatic, enabled by default and continuous in Oracle10g. It's reponsibility is to collect and store an extensive set of statistics. Oracle10g has been extensively instrumented to obtain precise timing information, both CPU and wait times, for a wide range of database operations. In addition, the observe phase records samples of database sessions activity at a frequency of one every second, to allow for fine grain analysis of user activity; it collects various statistics on resource usage, both at database and OS level, to help identifying any resource bottlenecks; finally it maintains statistics for highly used database entities, like high-load SQL statements and on often accessed objects like tables and indices.

Statistics collected by the observe phase are stored in the *Automatic Workload Repository* (AWR). AWR is a persistent store of performance data for Oracle10g and can be thought of as the Oracle performance datawarehouse. Statistics in AWR are organized chronologically, using hourly delta snapshots of in-memory statistics. The AWR is self-managed; it accepts policies for data retention and proactively purges data should it encounter space pressure. The same data is also used for feedback analysis, i.e. to analyze the result of tuning actions undertaken as part of previous analysis.

2.3 Diagnose Phase

Activities in this phase refer to the analysis of various parts of the database system using data in AWR or in in-memory views. The analysis is performed by a set of *Advisors*. Oracle10g introduces many advisors, each responsible for analyzing and optimizing the performance of its respective sub-components. ADDM and the SQL Tuning Advisor are presented later in this paper; other advisors include: *Segment Advisor* that analyzes space wastage by objects due to internal and external fragmentation; *Memory Advisors* that continuously monitor the database instance and auto-tune the memory utilization between the various memory pools for shared memory and process private memory [5]; *Undo Advisor* that provides optimal sizing of the Undo space.

2.4 Resolve Phase

The various advisors, after having performed their analysis, provide as output a set of recommendations that can be implemented or applied to the database. Each recommendation is accompanied by a benefit, in DbTime units, which the workload would experience should the recommendation be applied. The recommendations may be automatically applied by the database (e.g., the memory resizing by the memory advisors) or it may be initiated manually. This constitutes the Resolve phase.

Applying recommendations to the system closes an iteration of that particular tuning loop. The influence of the recommendations on the workload will then be observed in future performance measurements. Further tuning loops may be initiated until the desired level of performance is attained.

3 ADDM

The Automatic Database Diagnostic Monitor (ADDM) in Oracle 10g automates the entire process of diagnosing performance issues and suggests relevant tuning recommendations with the primary objective of maximizing the total database throughput. This advisor is executed out-of-the-box once every hour, each time an AWR snapshot is produced. Results of these analyses are kept by default for a month making it very easy for the DBA to address past performance issues.

Automatic performance diagnosis is very challenging because modern database systems have complicated interactions between their sub-components and have the ability to work with a variety of applications. This results in a very large list of potential performance issues such an automatic analysis could identify. Also, as new database technologies and applications are introduced, and older ones are made obsolete, it is pivotal that automatic diagnostic and tuning solutions can easily be adapted to accommodate such changes.

ADDM was designed with the following objectives:

- Should posses a holistic view of the database and understand the interactions between various database components.
- Should be capable of distinguishing symptoms from the actual root-cause of performance bottlenecks.
- Should provide mechanisms to diagnose performance issues on their first occurrence.
- Should easily keep up with changing technologies.

ADDM uses DbTime to identify database components that require investigation and also to quantify performance bottlenecks. Identifying the component consuming the most database time is equivalent to finding the single database component that when tuned will provide the greatest benefit. In other words, it is looking for ways to process a given set of user requests in the least amount of database time.

3.1 DBTime-graph and ADDM Methodology

The first step in automatic performance tuning is to correctly identify the root causes of performance problems, Only then is it possible to explore effective tuning recommendations to solve or alleviate the issue. ADDM looks at the database time spent in two independent dimensions: the first dimension looks at the database time spent in various phases of processing user requests, and includes categories like ’connecting to the database’, ’optimizing SQL statements’, ’executing SQL statements’; the second dimenstion looks at the database time spent using or waiting for various database resources used in processing user requests, and includes both hardware resources like CPU and I/O devices, and software resources like database locks and application locks.

ADDM looks at the database time spent in each category under both these dimensions and drills down into the categories that had consumed significant database time. This two dimensional correlation gives ADDM a very good judgment in zooming in to the more significant performance issues. The drill down process can be represented using a directed-acyclic-graph as shown in Figure 2, which we call the *DBTime-graph*.

It should be noted that this DBTime-graph is not a decision tree for a rule-based diagnosis system, where a set of rules is organized in the form of a decision tree that is traversed either to find the goal given a particular set of data or to find the data given a particular goal [1]. The DBTime-graph has various properties that differentiates itself from rule-based decision trees: (a) each node in this graph looks at the amount of database time consumed

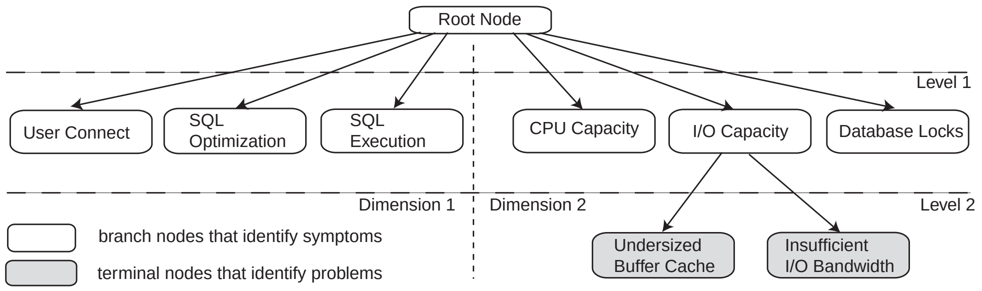


Figure 2: A Sample DBTime-Graph.

by a particular database component or resource; (b) all nodes in this graph are gauged with the same measure - DbTime; (c) all the children of a particular node are unconditionally explored whenever the database time spent in that node is significant; and (d) database time attributed to a particular node should be completely contained in the database time attributed to each of its parents. Any node that complies with all these properties can be added to the DBTime-graph making it easy to evolve with changing technologies, unlike the decision tree of a rule-based diagnosis system [1].

ADDM explores this DBTime-graph starting at the root-node and visiting all the children of a node if the database time consumed is significant. Branch nodes in this graph identify the performance impact of what is usually a symptom of a bottleneck, whereas the terminal nodes identify particular root-causes that can explain all the symptoms that were significant along the path in which the terminal node was reached. For example, in Figure 2, the branch node ”I/O Capacity” would measure database time spent in all I/O requests. Whenever significant database time was spent in I/O requests all the children of the ”I/O Capacity” node would be explored, which are the two terminal nodes in this example. The ”Undersized Buffer Cache” node would look for a particular root-cause, which is to see if the data-block buffer cache was undersized causing excessive number of I/O requests. The ”Insufficient I/O Bandwidth” node would look for hardware issues that could slow down all I/O requests.

Once a terminal node identifies a root-cause, it measures its impact in DbTime units. It then explores ways that can solve or alleviate the problem and comes up with actionable tuning recommendations based on the various workload measurements gathered. The nodes also estimate the maximum possible database time that could be saved by the suggested tuning recommendations, which need not necessarily be equal to the database time attributed to the root-cause.

It is interesting to note that ADDM doesn't traverse the entire DBTime-graph, rather it prunes the uninteresting sub-graphs. This is possible only because a node's database time is contained in the database time attributed to its parents. Consequently the cost of an ADDM analysis depends only on the number of actual performance problems that were affecting the database, and not on the actual load on the database or the number of issues that ADDM could potentially diagnose.

3.2 Workload Measurements

ADDM analysis can only be done if the appropriate data is available. Our first and most important requirement is that we collect all the data ADDM needs for each node in the DBTime-graph. ADDM needs data for the following operations: quantifying the impact in DbTime for the database components and operations; finding recommendations for alleviating root-cause problems and estimating the potential benefit in DbTime units. Our second requirement is the ”minimal intrusion principle”; it states that the act of collecting measurements for performance diagnostics should not cause a significant degradation in performance. All the data collection is done as part of the AWR snapshot mechanism described earlier. The various types of measurements include:

Database Time Measurements: The first priority in an ADDM analysis is to establish the main components that consume significant database time. This measurement is a cumulative non-decreasing function of time whose value over any time period can be got by a difference of the respective values from the start and end points. Direct measurements can only be done on database operations that usually take significant time to finish. The decision about which operations should be measured must be based on the cost of measurement (i.e. start and end a timer) and the expected length and quantity of such operations. For example, measuring the total time spent in I/O operations is reasonable while measuring the time spent in critical sections is not. Our solution to capture short duration operations is to use sampling, both frequency-based as well as time-based sampling.

Active Session History: We use regular time-based sampling to capture the activity in a system since it is not practical to collect a complete system trace of operations. This enables ADDM to narrow down root-causes of problems and give effective recommendations. We call the collection of sampled data the ”Active Session History” (ASH). Each sample contains information about what the database server is doing on behalf of each connected user (a.k.a. ”session”) at the time of sampling. We only collect data for sessions that are actively using the database during the sample time. If a specific operation consumes significant database time during the analysis period, there is a high probability that this operation will appear in a significant number of samples in ASH. This enables ADDM to diagnose such operations even if we do not measure them directly.

System Configuration Data: We collect system configuration data related to database settings. Since database settings do not change very often we maintain a full log of changes. This data can be crucial to giving recommendations for fixing specific problems. Examples of such data are size of memory components (like buffer cache), number of CPUs used by the system, special query optimizer settings.

Simulation Data: Sometimes, estimating the impact of a specific area of the database requires a simulation of various possible alternatives. For example to find that the buffer cache is the root-cause of an I/O issue we must determine that we spent time reading data blocks that were in the buffer cache at some point in time and were replaced. In other words, we need to determine how many read I/O operations could have been saved given an infinite buffer cache. Our solution is to simulate and quantify the effect of various cache sizes.

4 SQL Tuning Advisor

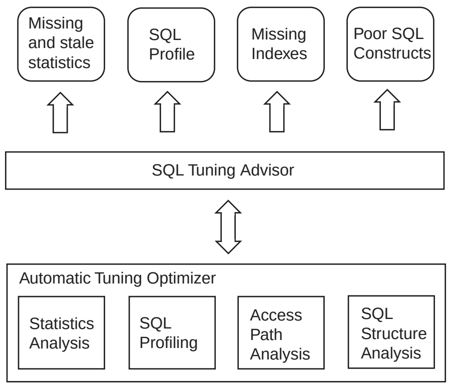


Figure 3: Automatic SQL Tuning Architecture.

The inherently complex activity of SQL Tuning requires a high level of expertise in several domains: query optimization, to improve the execution plan selected by the query optimizer; access design, to identify missing access structures; and SQL design, to restructure and simplify the text of a badly written SQL statement. Oracle10g address this problem by introducing a new advisor, the ”Automatic SQL Tuning Advisor”, implemented as a core enhancement of the Oracle query optimizer. It introduces the concept of *SQL profiling* to transparently improve execution plans. It also generates SQL tuning recommendations by performing cost-based access path and SQL structure ”what-if” analyses. Figure 3 shows the architecture of the Automatic SQL Tuning component. We term the special extension of the query optimizer as the Automatic Tuning Optimizer.

The advantage of using the Oracle query optimizer as the basis for Automatic SQL Tuning is multifold: tuning is done by the same component that is responsible for selecting the execution plan; future enhancements to the query optimizer are automatically considered; customized optimizer settings can be used based on the execution history of the SQL statement. The SQL Tuning Advisor acts as the front-end, accepting one or more SQL statements and passing it to the Automatic Tuning Optimizer along with other input parameters, such as a time limit. It then displays the results in the form of tuning recommendations, each with a rationale and an estimate of the benefit in DBTime units

4.1 SQL Profiling

The query optimizer relies on data and system statistics to function properly and by employing probabilistic models on these base statistics the query optimizer derives various data size estimates. Some of the main reasons for a sub-optimal plan include: missing or stale base statistics, wrong estimation of intermediate result sizes, and inappropriate optimization parameter settings.

To overcome these limitatione we introduce SQL profiling, a new concept that denotes the capability within the optimizer to obtain auxiliary information specific to a SQL statement based on 1) statistics analysis, 2) estimates analysis, and 3)parameters settings. A SQL Profile object is then built from this auxiliary information.

Once the user, acting on the recommendation generated, accepts a SQL Profile, it is stored in Oracle's data dictionary. When this SQL statement (same text with potentially different host variables and/or literal values) is subsequently presented to the system the optimizer will retrieve the SQL Profile from the dictionary and use it along with other statistics to build a well-tuned execution plan. The use of a SQL Profile remains completely transparent to the user, and more importantly its creation and use don't require changes to the application source code. The following is done as part of profiling:

Statistics Analysis: The goal here is to verify whether statistics are missing or stale. The Automatic Tuning Optimizer checks each of the statistics required during plan generation. It uses sampling to check the accuracy of the stored statistic. Iterative sampling with increasing sample size is used to meet this objective to obtain greater accuracy if needed. If a statistic is found to be stale, auxiliary information is generated to compensate for staleness. If it is missing, auxiliary information is generated to supply the missing statistic.

Estimates Analysis: One of the main features of a cost-based query optimizer is its ability to derive the size of intermediate results. Errors in estimates result in sub-optimal plans and can be caused by a combination of factors like uniform distribution assumption, column correlation and an inadequate statistical model for complex predicates. During SQL profiling, various standard estimates are validated by running parts of the query on a sample of the input dataset. When errors are found, compensatory information is added to the SQL Profile.

Parameter Settings Analysis: Here the past execution history of a SQL statement is used to determine the best optimizer settings. For example, the history may show that the output of a SQL statement is often partially consumed, consequently a setting to produce the first n rows is generated, where n is derived from this execution history.

4.2 Access Path Analysis

Creating suitable indexes is a well-known tuning technique that can significantly improve the performance of SQL statements. The Automatic Tuning Optimizer recommends the creation of indexes based on what-if analysis of various predicates and clauses present in the SQL statement being tuned. The recommendation is given only if the performance can be improved by a large factor.

4.3 SQL Structure Analysis

Often a SQL statement can be high-load simply due to the way it is written. This usually happens when there are different, but not semantically equivalent ways to write a statement to produce same result. It is important to understand that the optimizer, as part of regular plan generation process, already does semantically equivalent transformations. Semantic equivalence can be established when certain conditions are met; for example, a particular column in a table has the non-null property. However, these constraints may not exist in the database but instead are enforced by the application. The Automatic Tuning Optimizer performs a cost-based what-if analysis to identify missed query rewrite opportunities and issues recommendations.

5 Conclusions

In this paper we describe the Oracle's Self Tuning Architecture and how it enables a comprehensive automatic tuning solution. We then described two automatic tuning solutions: ADDM and SQL Tuning Advisor.

ADDM seeks to improve the overall throughput of the database via a comprehensive top-down performance analysis of the system. By using database time in conjunction with the two-dimensional DBTime-graph ADDM is able to quickly isolate the root causes of performance bottlenecks and provide very specific actionable recommendations, obtained by using fine-grained sampling data. Please refer to [7] for more details.

The SQL Tuning Advisor is based on the Automatic Tuning Optimizer, an extension of the Oracle query optimizer. We have described the multipronged approach to SQL Tuning, and the unique concept of SQL Profiling that results in a SQL Profile object associated with the SQL statement and used subsequently during plan generation. For more information please refer to [6].

Acknowledgements

The authors would like to thank all the members of the Server Manageability team at Oracle for their valuable contribution in making Oracle the first self-managed database and helping us to write this paper. In addition, we would like to specially thank Uri Shaft for helping us with LaTeX.

References

[1] D. G. Benoit. Automatic Diagnosis of Performance Problems in Database Management Systems, PhD Thesis, Queen's University, Canada, 2003.

[2] K. P. Brown, M. Mehta, M.J. Carey, M. Livny: Towards Automated Performance Tuning for Complex Workloads. VLDB 1994.

[3] S. Chaudhuri and G. Weikum: Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. VLDB 2000.

[4] S. Chaudhuri, B. Dageville, G. M. Lohman: Self-Managing Technology in Database Management Systems. VLDB 2004.

[5] B. Dageville, M. Zait: SQL Memory Management in Oracle9i. VLDB 2002, 962-973.

[6] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, M. Ziauddin: Automatic SQL Tuning in Oracle10g. VLDB 2004.

[7] K. Dias, M.Ramacher, U. Shaft, V.Venkataramani, G.Wood: Automatic Performance Diagnosis and Tuning in Oracle. Proceedings of the 2005 CIDR Conf.

[8] Gartner Group: Total Cost of Ownership: The Impact of System Management Tools, 1996.

[9] Hurwitz Group: Achieving Faster Time-to-Benefit and Reduced TCO with Oracle Certified Configurations, March 2002.

[10] G. Weikum, A. Mönkeberg, C. Hasse, P. Zabback: Selftuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering, VLDB 2002.

JOpera: Autonomic Service Orchestration

Cesare Pautasso, Thomas Heinis, Gustavo Alonso
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
{pautasso, heinist, alonso}@inf.ethz.ch

1 Introduction

The increasing interest in new software engineering technologies for application integration such as Service Oriented Computing and Service Orchestration has resulted in a proliferation of workflow management systems as the underlying representation and execution platform for service composition [7]. Workflow management system are also being applied to new domains (e.g., virtual scientific laboratories [1], Grid computing [12], service delivery and provisioning [6]). For these new applications, workflows are seen as the modeling metaphor behind the notion of *straight through processing* and *virtual organizations* where a collection of existing heterogeneous systems are composed into an integrated solution.

In all these settings workflow engines are at the core of a complex combination of applications and clustered computers. As such, they have become rather difficult to deploy and configure, let alone tune to obtain maximum performance. This problem is not unique to workflow and service composition engines but it is more difficult to address in these settings because there is only a limited understanding of the execution procedures behind a workflow engine. In this short paper we report on our ongoing work to design and develop an autonomic workflow engine that can be used for large scale service composition. The challenge we face in doing this is threefold. First, we need to design an execution procedure for service compositions that is amenable to autonomic treatment. Second, this procedure needs to be realized in an architecture that supports the deployment of different modules of the system across a computer cluster in order to achieve the desired level of performance. Third, an autonomic controller and appropriate control policies need to be developed to automatically provision the optimal amount of resources to the engine.

In what follows we provide a high level description of how we have accomplished these three goals and give a brief account of the performance of the system. The implemented system is part of the JOpera project. JOpera is an advanced SOA tool for Eclipse, which provides modeling, execution, monitoring and debugging tools for workflow-based Web service orchestration. A more detailed presentation of the autonomic capabilities of JOpera, including an extensive experimental evaluation of the approach can be found in [4, 11].

2 Web Service Orchestration with Workflows

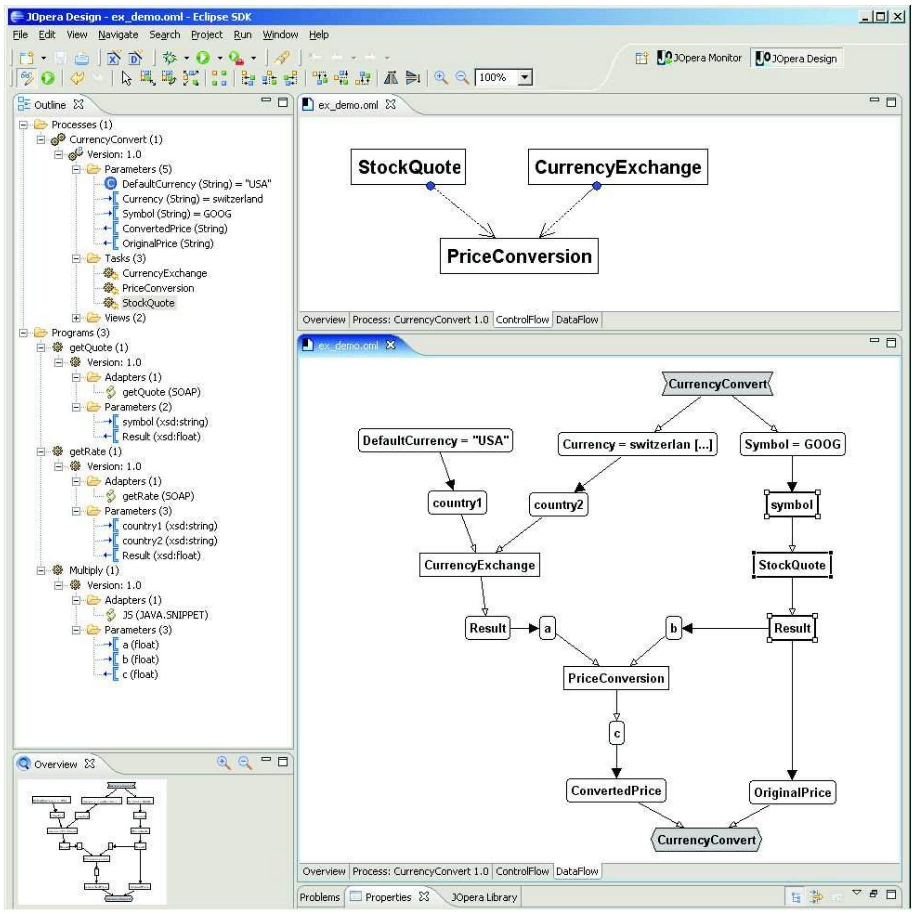
In Service Oriented Architectures (SOA) workflow modeling languages have found a good application to define an executable model of the flow of information between a set of services [7]. A workflow process defines the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

interactions between a set of services by scripting (or orchestrating) the exchange of messages between them. To simplify its integration and reuse, the resulting workflow is also typically published as a service.

As an example, we illustrate a small workflow for providing a value-added service out of the composition of two basic ones. In particular, the workflow shows how a service providing stock prices converted in any currency can be built out of the composition of two services: one returning stock prices in U.S. dollars and the other one returning currency exchange rates between dollars and the requested currency.

The screenshot of Figure 1 shows how the workflow is developed using JOpera. The outline view on the left contains the structure of the workflow in terms of its tasks and also lists the services to be composed. The editors on the right show two graphs defining the control flow and data flow relationships between the tasks of the workflow. The control flow graph (at the top) defines the order of execution of the tasks of the workflow. Since the stock quote and currency exchange services are independent of each other, the tasks invoking them can be executed in parallel. Once both of these tasks complete, the task computing the converted price is executed. The data flow graph of the workflow (shown in the bottom editor) defines where the information required by each service comes from. The result of the entire workflow, to be returned to its client, is produced by the StockQuote service invocation task for the OriginalPrice and by the PriceConversion task for the ConvertePrice. This task receives its input from the result of the invocation of both the StockQuote and the CurrencyExchange service. These are invoked passing data (the Currency and the Symbol identifying the stock) provided by the client as input of the whole workflow.



```
Processes (1)
  CurrencyConvert (1)
    Version: 1.0
      Parameters (5)
        DefaultCurrency (String) = "USA"
        Currency (String) = switzerland
        Symbol (String) = GOOG
        ConvertedPrice (String)
        OriginalPrice (String)
      Tasks (3)
        CurrencyExchange
        PriceConversion
        StockQuote
      Views (2)
Programs (3)
  getQuote (1)
    Version: 1.0
      Adapters (1)
        getQuote (SOAP)
      Parameters (2)
        symbol (xsd:string)
        Result (xsd:float)
  getRate (1)
    Version: 1.0
      Adapters (1)
        getRate (SOAP)
      Parameters (3)
        country1 (xsd:string)
        country2 (xsd:string)
        Result (xsd:float)
  Multiply (1)
    Version: 1.0
      Adapters (1)
        JS (JAVA.SNIPPET)
      Parameters (3)
        a (float)
        b (float)
        c (float)
```

Figure 1: Defining a simple Web service orchestration in JOpera

In addition to the separate visualization of the control flow and data flow aspects of a workflow and the use of control flow extraction algorithms to ensure the automatic reconciliation between the two, JOpera also offers efficient means of binding each task of the workflow to the service to be invoked while executing it [9]. In this example, the tasks devoted to collecting information from the external providers of stock prices and currency rates are bound to a standard-compliant Web service (described using plain WSDL, accessed using SOAP messages). With the price of additional complexity and overhead, this ensures the interoperability between the workflow engine and the service provider and removes the need of developing customized adapters to make the engine access external sources of information. For the third task, responsible for computing the converted price by running a multiplication between the original price and the corresponding exchange rate, it should not be necessary to pay the overhead of a remote SOAP call. In JOpera, the PriceConversion service can be implemented using a so-called Java snippet, which is invoked with the overhead comparable to a local Java method call.

3 Stage-based Workflow Execution in JOpera

Running such a workflow process involves executing the tasks of the workflow in the correct order and passing the data produced by one task to its successors. In the context of Web service orchestration, tasks are typically bound to service invocations and their execution involves the exchange of messages between the engine and an external service provider. Messages are also exchanged in the reverse direction, when clients of the workflow engine want to initiate the execution of a new process instance. Upon receipt of such message, the engine begins running a new process instance, analyzes its control flow structure and determines which tasks need to be executed next. Then, for each active task, the engine selects the service to be invoked, fetches data from the process variables to compose a message, which is then sent to the corresponding service provider. Once this invocation completes, the state of the process needs to be updated with the results so that other tasks can access them. Process execution continues until all tasks have been executed, or an explicit termination point in the workflow has been reached.

Clearly, workflow engines are capable of running more than one instance of a workflow at the same time. This feature is also very important in the context of Web service orchestration: once processes are published as a Web service, clients can send messages to the engine for starting a new process at any time. Given the limited amount of resources (i.e., CPU threads and memory) available to the engine, it becomes important to restructure the execution of a workflow so that the engine can scale to run a large number of concurrent process instances.

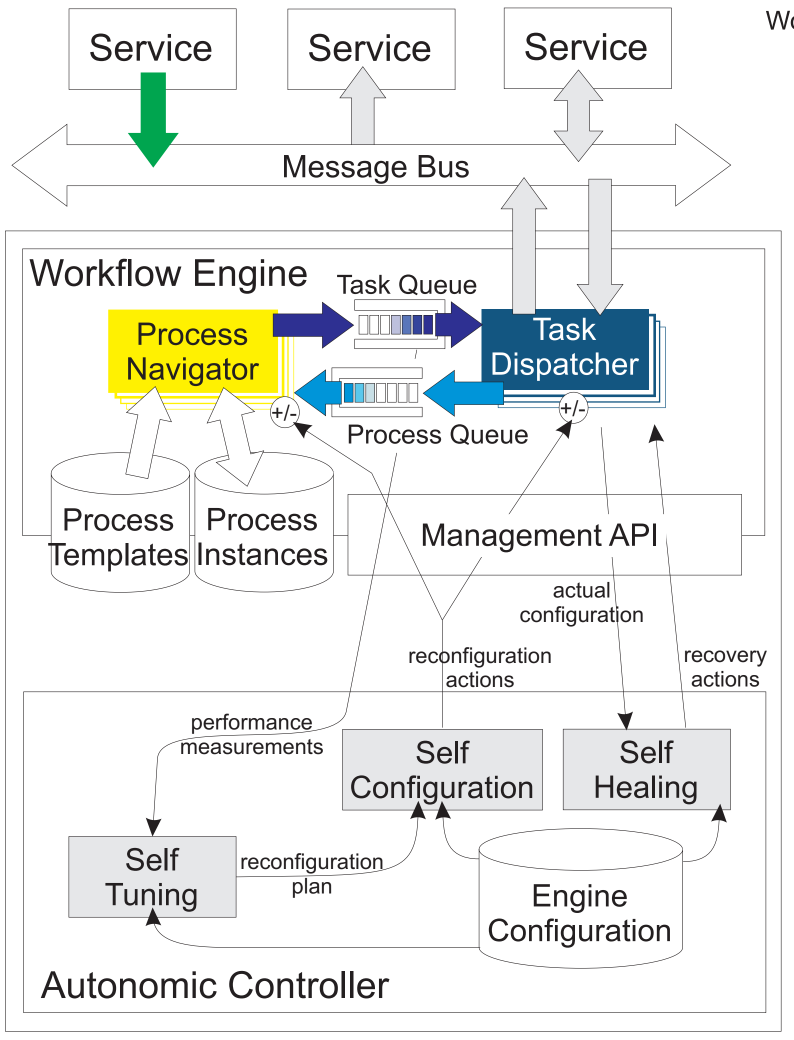
In this regard, the simple solution of permanently assigning a thread to run each process instance suffers from a number of limitations. The number of concurrent threads that are available in a virtual machine would set a limit to the number of processes that can be run by the engine at a given time (a few hundred). Furthermore, such threads would be underutilized as they would dedicate most of their time to I/O operations, i.e., sending and receiving SOAP messages. Finally, assigning one thread to each process instance would limit the amount of intra-process parallelism supported by the engine. In other words, even if the control flow structure of a process defines a partial execution order over its tasks, this engine threading model would serialize the execution of all tasks within a process.

One of the innovative design decisions of the JOpera engine lies in employing a threading model which effectively decouples the process instances from the threads executing them. Apart from shifting the factor limiting the maximum number of concurrent processes that can be executed from the number of threads to the amount of available memory, this decision also makes it possible for the same engine architecture to scale out from a centralized to a distributed configuration [10].

To do so, we have partitioned the execution of a process in two *stages*. The first involves the, so-called, process *navigation*, i.e., making the control and data *flow* through the process instance by using a graph traversal algorithm to determine which tasks of the process are to be executed next based on their dependencies to the already completed tasks. The second stage – *dispatching* – involves the actual execution of the tasks, which boils down to the synchronous or asynchronous exchange of messages with the provider of the Web service to which the task has been bound.

In the architecture of JOpera, these two execution stages have been assigned to two different (and loosely coupled) active components of the engine: the navigator and the dispatcher. The navigator runs processes, the dispatcher runs tasks. As it can be seen from Figure 2 they communicate asynchronously using queues. Whenever the navigator has determined that a new task is ready to be executed, the information required to perform such execution is added to the *task queue*. The dispatcher takes tasks from such queue and performs the corresponding Web service invocation. Once the invocation is complete, the dispatcher puts its results in the *event queue*. The navigator collects them, updates the state of the corresponding process instance and continues running it by sending the next tasks to be executed to the dispatcher.

Given an appropriate implementation of such task and event queues, the navigator and dispatcher components can be run by threads which are distributed on different physical hosts, e.g., a cluster of computers [5]. To achieve a large task execution capacity dispatchers can be run by a large thread pool. Similarly, navigators running different (and independent) process instances can also be replicated among a pool of threads.



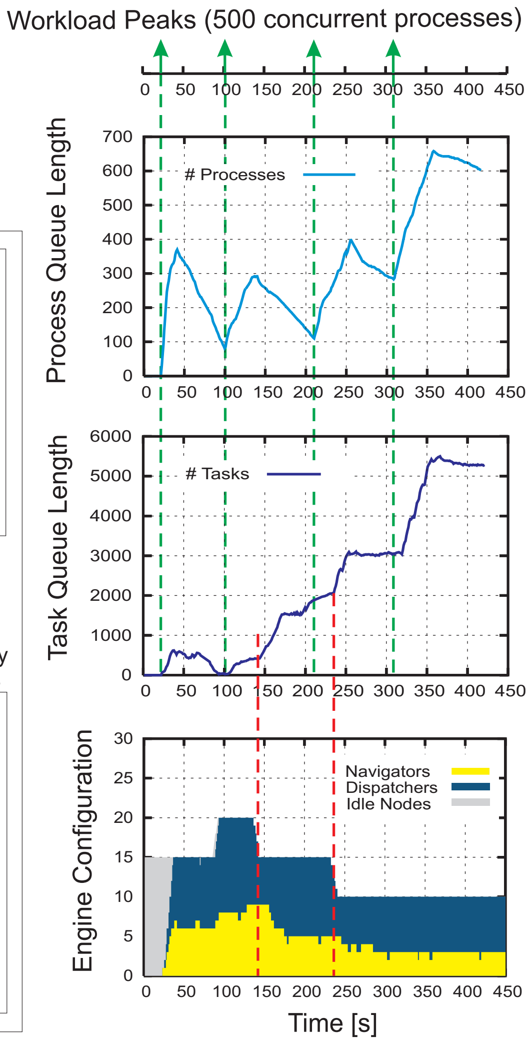


Figure 2: Architecture of a self-managing workflow engine (left) and performance evaluation (right)

4 A Deployment Dilemma

Although such stage-based architecture delivers the necessary flexibility to adapt the number of dispatcher and navigator components to the workload executed by the engine, it opens up another important problem, related to the *deployment* and *management* of such system. The engine may face an unknown number of internal and external clients that can define and run an unpredictable number of processes concurrently. Thus, the structure of these processes and the number of their instances that will have to be executed in response to clients cannot be determined a priori. This makes it difficult to choose between a centralized or a distributed solution for the deployment of the system. Moreover, in case a distributed approach is chosen to provide the required level of performance, the correct amount of resources must be provisioned and these must be managed and optimally configured, in terms of how the resources are allocated to navigators and dispatchers.

To illustrate this problem, in Figure 3 we include an example showing the sensitivity of the system to its configuration. In this example we started 1000 concurrent processes and measured their total execution time using different configurations of the engine.

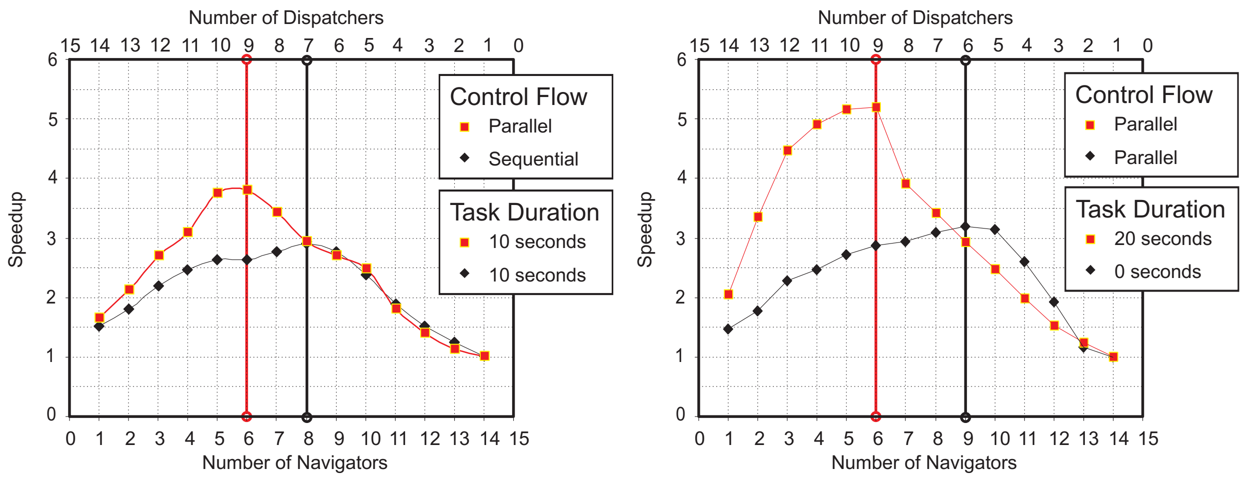


Figure 3: Speedup relative to the slowest configuration of four different workloads over all possible configurations of the engine deployed on a cluster of 15 nodes

Tuning the engine by allocating the “right” amount of navigators and dispatchers can – in this example – achieve a 5x speedup in the execution of the same workload. Still, the optimal configuration for one kind of workload could turn out to be sub-optimal for a different load. In the example (Figure 3 right), we use two workloads characterized by different task durations. Fast tasks can be consumed quickly by the dispatchers and shift the load onto the navigators, which have to resume executing processes after having issued the tasks which immediately complete. For short tasks duration, the highest speedup is found with a configuration which allocates more resources to navigators. The opposite occurs with slow tasks, which keep the dispatchers busy for a longer time. In Figure 3 left, we show that with tasks of the same duration, second order effects due to the process control flow structure become apparent.

From this example, it is clear that it is not enough to base the deployment decisions on estimating the performance of a certain configuration of the system by making assumptions about the properties of its workload [2]. Once these assumptions no longer hold, the system will be misconfigured and use the available resources inefficiently. Instead of a static solution, we choose to follow a dynamic approach based on a closed feed-back loop. As we are going to illustrate in the following section, we have extended the workflow engine with self-management capabilities so that it can adjust its configuration on the fly based on measurements of its performance under the actual workload.

To do so, we introduce an autonomic controller whose algorithms and policies do not make any assumption about the structure of the processes to be executed. This controller can dynamically grow and shrink the size of the system based on the number of processes that are currently active. Such self-managing engine can be initially be deployed in a centralized configuration and gradually evolve as a distributed engine as its workload gets larger. This avoids the problem of resource overprovisioning, where the workflow engine would have to be dimensioned for peak capacity at all times.

Such a solution requires the engine to support dynamic reconfiguration. Clearly, stopping the entire engine in order to migrate some of its components and to alter its configuration is not possible as it would affect the availability of the processes that are published as a Web service through the engine. Instead, with our design, to grow the size of the engine, i.e., to increase its process or task execution capacity, it is enough to provision an additional thread for executing the navigator or dispatcher components. In case of overprovisioning, threads can be relinquished and the size of the engine reduced accordingly.

5 Autonomic Deployment

The engine's architecture has been designed to provides all of the necessary extension points to follow a "bolt on" approach to achieve self-management [3]. A management API for monitoring the performance of the engine and for applying reconfiguration actions is available. This interface can be used both for manual system administration tasks, as well as automatic self-management when the appropriate autonomic controller component is added.

As opposed to measuring raw hardware metrics (e.g., CPU utilization) of the physical hosts running the engine, in our approach we have chosen to observe how the workload affects the performance of the engine at a higher level of abstraction. The workload for the engine can be defined as the number of active process instances that are being executed. This can be measured as the workload affects both execution stages and influences the length of the process queue serviced by navigators as well as the number of tasks to be invoked by dispatchers. The execution of a large number of processes will generate a large number of tasks. Still, monitoring the queues gives a precise picture of the performance of each stage. The amount of queued tasks depends on the degree of parallelism within a process and will change for processes having different control flow structures. Additionally, the size of the process queue will grow if several tasks complete their execution at the same time.

As shown in the lower part of Figure 2, this performance information is fed into the autonomic controller, which processes it and reacts by applying the appropriate reconfiguration and recovery actions to the engine. The controller is structured into three functional components: self-healing, self-tuning and self-configuration, which interact asynchronously and share a common model of the engine's configuration. As a first approximation, this model includes information about the available resources of the cluster and their allocation state. The configuration information is kept up to date by the self-healing component which periodically monitor its consistency with respect to the actual configuration of the cluster. Once a mismatch occurs, the self-healing component detects a failure in the engine, updates the configuration model and performs the appropriate recovery actions. For example, tasks executing on a failed dispatcher have been lost and have to be requeued to be retried.

The goal of the self-tuning and self-configuration component is to work together in order to keep the engine provisioned with the optimal amount of resources and ensure that the current configuration provides a good performance. The self-tuning component periodically reads performance measurements (i.e., the size of the process and task queues) from the engine and uses this information to detect imbalances in the current configuration. Measuring the length of the event and task queues makes it easier to define high level policies to control the engine's configuration. Such policies map observed variables into a reconfiguration plan. For example, if the task queue grows beyond a certain threshold, a dispatcher should be added to increase the rate of task execution.

This abstract plan (e.g., to add one dispatcher) is executed by the self-configuration component. Decoupling planning from execution is important not only because it takes time to perform the actual reconfiguration, but also because it allows the controller to choose the optimal resource targeted by the reconfiguration plan. While a reconfiguration is taking place, the self-tuning component can continue to observe the system's behavior and possibly update the reconfiguration plan with new decisions based on more up-to-date information. At the same time, the self-configuration component can choose the appropriate resource on which to apply the plan by minimizing the distruption caused by the reconfiguration.

6 Evaluation

To evaluate the architecture of the engine and its autonomic capabilities we have performed a number of experiments which (1) motivate the need for adding self-management capabilities to the engine (2) show that the controller can indeed automatically reconfigure and heal the system [4] and (3) compare the performance of different control policies [11].

Due to space limitations, in this section we only describe the results of a self-healing experiment. In addition to adjusting the configuration in response to changes in the workload applied to the system, in this case, the controller also reacts to external changes in the system configuration. The right side of Figure 2 shows a trace of how the engine evolves, from the point of view of the controller. The top two graphs include measurements of the performance of the engine in terms of the length of the task and process queues. The bottom graph shows various snapshots of the configuration of the engine over time, defined as the number of nodes of the cluster that have been allocated to run dispatchers and navigators.

Given the lack of benchmarks for autonomic workflow engines, we have performed a basic load test, where the system is periodically hit by a peak of n messages that are handled by starting the execution of the same number of processes in parallel. To simplify the analysis of the results, these processes have the same structure and contain the same number of tasks. In the experiments, four peaks of 500 processes arrive at $t = 20s, 100s, 205s, 305s$. The controller notices that the workload has increased by observing the evolution of the process queue length. When such queue gets longer, it means that the engine needs to allocate more process execution capacity. Thus, the controller allocates up to 5 navigators to service the process queue. Once the processes begin execution, also the task queue gets filled up and, in the first part of the experiment, the controller allocates up to 10 dispatchers to deliver the required task execution capacity.

While the second peak arrives, at $t = 100s$, the engine undergoes a maintenance operation. First, 5 nodes are added to the pool of resources of the engine, then other 5 nodes are taken out of the pool for maintenance (at $t = 140$). This manual node rotation is part of the normal maintenance of the system and should not disrupt its operations. The controller immediately makes uses of the newly added resources by allocating 3 additional dispatchers and 2 additional navigators. Still, once 5 nodes are taken out of the pool, the self-healing component notices their disappearance and recovers the tasks and processes that were running on the failed nodes. Also, the configuration of the remaining nodes is out of balance. This will be corrected before the next peak of processes arrives. At $t = 230$ the newly added nodes fail and the engine continues running with only 10 nodes. Clearly its performance has decreased as both task and process queues get increasingly longer. The controller tries to make use of the remaning nodes and the system keeps running.

7 Conclusion

In the same way that using database engines provides considerable savings in terms of code to be developed in large applications, workflow engines greatly simplify the orchestration problem in application integration settings. Yet, for developers to be able to take advantage of such savings in coding, the performance of existing workflow engines needs to be significantly improved.

In this paper we show how the processing capacity of a service composition engine based on the workflow paradigm can be extended automatically in response to changes in the load. Although we have shown how to apply it to JOpera, our approach is independent of the particular workflow engine, as its general principles can be applied to any process execution engine for Web service orchestration (e.g., other implementations of WS-BPEL [8]).

The solution described builds upon three important ideas: separating dispatching from navigation in the process execution, implementing them as separate modules, and designing appropriate policies for determining how many dispatchers and how many navigators are needed according to the current workload. The fact that the system can dynamically adjust the number of navigation and dispatching modules it utilizes by itself is an important property that frees up the developer and system administrator from having to worry about tuning and deployment configurations.

Downloading JOpera

The latest release of JOpera for Eclipse, including several examples to get started, can be downloaded from `www.update.jopera.org`. Additional publications and documentation can be found on `www.jopera.org`.

Acknowledgements

Part of this work is funded by the European projects: IST-FP6-004559 SODIUM (Service Oriented Development In a Unified fraMework) and IST-FP6-15964 AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

[1] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proc. of the 17th International Conference on Data Engineering (ICDE2001)*, pages 235–242, Heidelberg, Germany, 2001.

[2] M. Gillmann, W. Wonner, and G. Weikum. Workflow Management with Service Quality Guarantees. In *Proc. of the ACM SIGMOD Conference*, pages 228–239, Madison, Wisconsin, 2002.

[3] R. A. Golding and T. M. Wong. Walking toward moving goalpost: agile management for evolving systems. In *First Workshop on Hot Topics in Autonomic Computing*, Dublin, Ireland, October 2006.

[4] T. Heinis, C. Pautasso, and G. Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *Proc. of the 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005.

[5] L. jie Jin, F. Casati, M. Sayal, and M.-C. Shan. Load Balancing in Distributed Workflow Management System. In G. Lamont, editor, *Proc. of the ACM Symposium on Applied Computing*, pages 522–530, Las Vegas, USA, 2001.

[6] R. Khalaf, A. Keller, and F. Leymann. Business Processes for Web Services: Principles and Applications. *IBM Systems Journal*, 45(2):(to appear), 2006.

[7] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[8] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, 2006.

[9] C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proc. of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, pages 39–53, Toronto, Canada, August 2004.

[10] C. Pautasso and G. Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce (IJEC)*, 9(2):104–141, Winter 2004/2005.

[11] C. Pautasso, T. Heinis, and G. Alonso. Autonomic Execution of Service Compositions. In *Proc. of the 3nd International Conference on Web Services*, Orlando, FL, July 2005.

[12] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 2006 (to appear).

Challenges in building a DBMS Resource Advisor

Dushyanth Narayanan
Microsoft Research
Cambridge, UK
dnarayan@microsoft.com

Eno Thereska
Carnegie Mellon University
Pittsburgh, PA
enothereska@cmu.edu

Anastassia Ailamaki
Carnegie Mellon University
Pittsburgh, PA
natassa@cmu.edu

Abstract

Administration increasingly dominates the total cost of ownership of database management systems. A key task, and a very difficult one for an administrator, is to justify upgrades of CPU, memory and storage resources with quantitative predictions of the expected improvement in workload performance. We present a design and prototype implementation of a Resource Advisor that is able to answer "what-if" questions about DBMS performance under hypothetical conditions. We discuss the design issues and challenges involved in building such a Resource Advisor, as well as our experiences in building a prototype Resource Advisor for SQL Server.

1 Introduction

Administering database management systems (DBMS) is a complex and increasingly expensive task. There is a pressing need to raise the level of abstraction at which database administrators (DBAs) interact with the system, by automating tasks which currently require substantial human effort and expertise [8]. In this paper we focus on the task of *resource (re)provisioning*: determining the number, type, and configuration of hardware resources most appropriate to a given workload, hardware budget, and performance goals.

Resource provisioning is typically done by human experts using experience and rules of thumb to decide whether additional resources will improve performance [3]. The cost of such experts is significant for large enterprises and prohibitive for small ones. Even experts find it difficult to *quantify* the expected benefit of a resource upgrade. The net result is over-provisioned systems with no guarantees on performance [8].

The key technical challenge in automating resource provisioning decisions is automated prediction of performance in hypothetical hardware configurations. In other words, we wish the system itself to provide accurate, quantitative answers to "what-if" questions such as "*what* would be the increase in throughput *if* the server's main memory were doubled?" In this paper we discuss the design issues and challenges in building such a predictive capability, and also our experiences in building a specific system, a *Resource Advisor* for SQL Server.

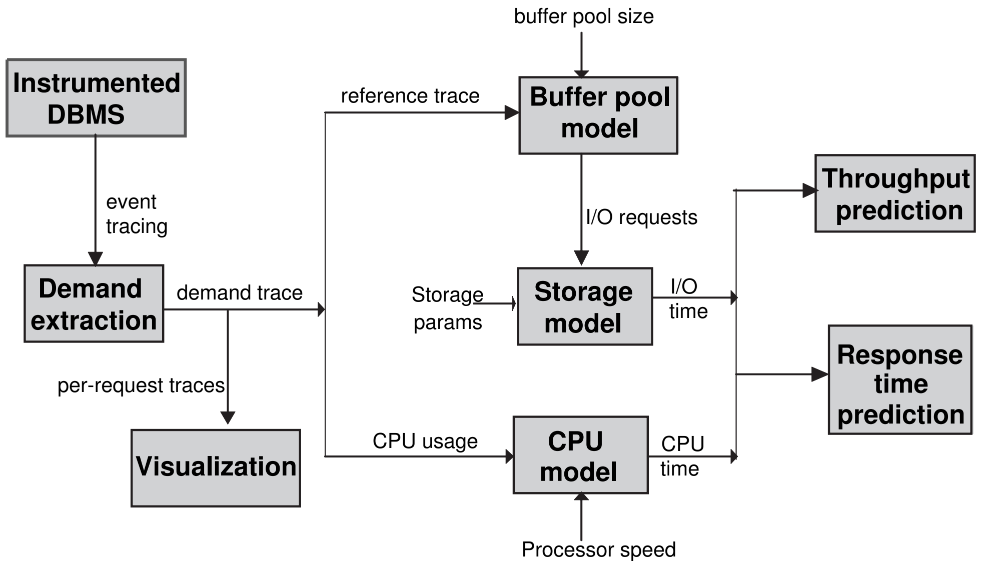


Figure 1: Resource Advisor architecture

2 Design principles

Large commercial databases are complex systems that depend on several physical resources such as the back end storage system, volatile main memory and CPUs. A database administrator (DBA) must decide on a good initial configuration of these resources, and then continuously monitor the system for new bottlenecks and changes in workload. To do this she must have an intimate understanding of the various database components, their interactions, and of the workload. Such experienced DBAs are expensive and even they do not have the tools to accurately and easily predict the performance effect of any resource provisioning decision.

Consider a DBMS running multiple application workloads with different resource demands and performance requirements. For example, DSS workloads have low concurrency and total run time is the metric of interest. OLTP workloads have high concurrency and require not only high throughput but also bounded response time. For any proposed resource provisioning the DBA must estimate the impact on the performance of each workload, taking into account the resource contention between them.

The most common approach to (re)provisioning such systems is to monitor the performance counters provided by most commercial systems. These counters measure aggregate load statistics for various resources, which is not always sufficient to find the global bottlenecks. They do not offer any insights into response time, as they do not track per-request resource usage or distinguish between critical-path and background resource usage. Finally, they place the heavy burden on the DBA of correctly interpreting 400+ performance counters.

We advocate a system architecture that addresses these problems by

1. *Tracing* per-request resource usage and control flow at fine granularity.
2. *Modelling* hardware resources and the algorithms that schedule or share them across multiple requests.
3. *Predicting* performance on hypothetical hardware by combining workload traces with hardware models.

The bottleneck in DBMS provisioning today is the human in the loop. CPU cycles are relatively abundant, allowing fine-grained yet low-overhead tracing of the live system, as well as offline trace processing using idle cycles. For example, SQL Server running an OLTP workload generate 500 events/transaction, with a CPU overhead of 1000 cycles/event and generated 68 bytes/event of trace data [5]. Extrapolating to the fastest TPC-C system as of date (3,210,540 tpmC with 64 processors at 1.9 GHz) [7], we get a CPU overhead of 5% and a trace data rate of 8 MB/s. This could be reduced further through optimisation, sampling, and runtime event filtering.

	Event Type	Arguments	Description
Control Flow	*StartRequest*		SQL transaction begins
	EndRequest		SQL transaction ends
	EnterStoredProc	*procname*	Stored procedure invocation
	ExitStoredProc	*procname*	Stored procedure completion
CPU scheduling	*SuspendTask*	*taskID*	Suspend user-level thread
	ResumeTask	*taskID*	Resume user-level thread
	Thread/CSwitchIn	*cpuID, sysTID*	Schedule kernel thread
	Thread/CSwitchOut	*cpuID, sysTID*	Deschedule kernel thread
Buffer pool activity	*BufferGet*	*pageID*	Fetch a page (blocking)
	BufferAge	*pageID*	Reduce the “heat” of a page
	BufferTouch	*pageID*	Increase the “heat” of a page
	BufferDirty	*pageID*	Mark a page as dirty
	BufferReadAhead	*startpage, numpages*	Prefetch pages (non-blocking)
	BufferEvict	*pageID*	Evict and free page
	BufferNew	*pageID*	Create a new page
	BufferSteal	*numpages*	Allocate memory from free pool
	BufferFree	*bufferID*	Release memory to free pool
Disk I/O	*DiskIO*	*startpage, numpages*	Asynchronously read/write pages
	DiskIOComplete	*startpage, numpages*	Signal read/write completion
Locking	*EnterLockAcquire*	*resourceID, mode, timeout*	Attempt to lock a resource
	ExitLockAcquire	*status*	Success/failure of lock acquisition
	LockRelease	*resourceID, mode*	Release a held lock

Table 1: Instrumentation events

```
8113984086 0 XactionStart tpcc_neworder,0
8113984086 1:0 CPU 3663
8113987749 2:1 CPU 187
...
8113990027 12:11 LOCK KEY: 5:844424932360192 (e102aa462451),S,ACQUIRE
...
8114036559 269:268 MEM ALLOC,1
...
8114152008 368:367 CPU 8544
8114160900 369:368 BUF 00000005,00000001,00000170,Fetch
...
8114160900 432:368 BUF 00000005,00000001,000001AF,Fetch
8114160900 433:368 CPU 109
...
```

Each resource demand contains a timestamp and a “demand ID”, followed by a list of previous demands that must precede this one, in the transaction execution. This allows us to capture any in-transaction concurrency: e.g., demands 369–432 are asynchronous prefetch requests to the buffer manager, which are executed concurrently with demand 433, i.e. computation is overlapped with I/O here. Each demand has additional type-specific parameters, e.g. lock demands specify a resource ID, a mode, and an action (acquire or release).

Figure 2: Simplified snippet of demand trace

3 Experience

Based on the principles and high-level design described above, we have designed and implemented a *Resource Advisor* for SQL Server, which predicts the performance of a live workload under hypothetical hardware upgrades. Here we briefly describe our experiences with an early prototype based on analytic models, which was described in detail in an earlier paper [6]. We then describe our current simulator-based approach.

3.1 Analytic modelling

Figure 1 shows the high-level design of the Resource Advisor. It relies on fine-grained, low-overhead event tracing from an instrumented DBMS. The instrumentation points are chosen to enable *end-to-end tracing* [2] of each request from the moment it enters the system to its completion. We record each use of system resources — CPU, memory, I/O — as well as virtual resources such as locks. Table 1 shows the set of events traced by our instrumentation. These events allow the Resource Advisor to reconstruct exactly the sequence of resource demands issued by the workload. Since this sequence is an aggregate of many concurrently executing requests, the Resource Advisor first separates it out into *per-request* demand traces. This requires instrumentation of all context switches: points where a resource such as CPU stops working on one request and starts work on another.

The raw event trace is transformed into a per-request demand trace, where each request is represented as a partially ordered set of resource demands, each for a specific resource. Figure 2 shows a simplified snippet of a demand trace for an OLTP transaction. The aggregate demand on the system is then the effect of concurrently executing these per-request demands.

Subsequent steps in processing are parametrised by the characteristics of the hypothetical "what-if" hardware: the buffer cache memory size, the CPU clock speed, and disk parameters such as rotational speed. The buffer references are processed by a cache simulator to generate an I/O trace, and the I/O and CPU traces are fed to analytic models that predict the throughput and mean response time of each transaction type.

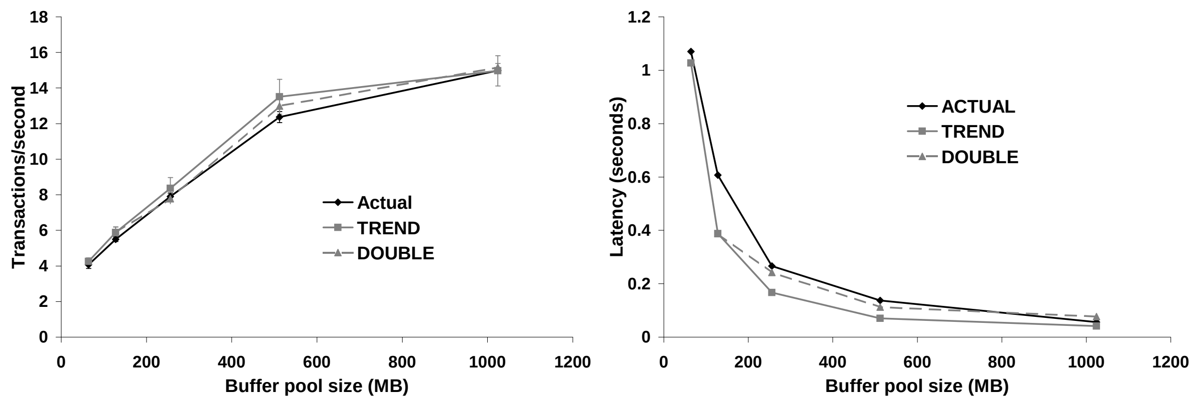
Our analytical models are able to accurately predict the effect of changing the buffer cache memory on the throughput and response time of an OLTP workload. Figure 3 shows the prediction accuracy for two different types of "what-if" questions. DOUBLE predicts the effect on performance of doubling the memory of the current configuration (e.g. from 128 MB to 256 MB). TREND predicts performance over the entire range of memory sizes, bases on observing the system with 64 MB.

Thus the models have good accuracy but restricted applicability. They make two major assumptions about the workload, which are valid for OLTP but not for other workloads such as DSS:

- that buffer cache misses cause a random-access I/O pattern;
- that the throughput bottleneck remains the same throughout the workload execution, i.e. the workload does not have different phases with different bottlenecks.

With analytic models based on operational analysis, it is easy to predict aggregate throughput, assuming sufficient concurrency that the bottleneck resource is always busy. However, if there are multiple concurrent users, each with a different workload (for example a different transaction mix), then it is difficult to predict the throughput of each user individually.

Analytic models also make it difficult to predict response time. Our models predict mean response time per transaction type but are specific to OLTP. They also assume that a request's response time is dominated by its resource demands rather than queueing and scheduling delays caused by concurrently executing requests. To correctly model queueing and scheduling delays, and to compute second-order metrics such as variance in response time, we need queueing models. However, analytic queueing models rely on assumptions about request arrival time distributions that are often unrealistic.



(a) Throughput (b) "New order" response time

Figure 3: Predicting OLTP performance for hypothetical memory changes

3.2 Simulation-based modelling

The current version of the Resource Advisor is based on event-driven simulation rather than analytical throughput and response time models. Live workload traces are decomposed into per-request demand traces as before, and the concurrent execution of the requests on hypothetical hardware is modelled by the simulator. The result of the simulation is a execution trace with the predicted timing information (including scheduling delays) of each resource demand within each request. This allows us to compute the predicted throughput, response time, or any other performance metric of interest. Unlike the analytic models, the simulation approach is workload-agnostic and also enables a wider range and a finer granularity of performance metrics.

Each request's demands are executed by the simulator according to the partial order specified in the demand trace. Demands are executed by passing them to the appropriate resource model, which determines their completion time by adding any queueing/scheduling delay as well as the predicted service time:

- The *CPU model* computes scheduling delay by simulating a non-preemptive FIFO scheduler. Service time is computed by scaling, i.e. the speed of CPU execution is assumed to be proportional to the clock speed.
- The *buffer cache model* simulates an LFU eviction policy. Cache misses generate disk demands which are handled by the disk model. Note that disk demands are not directly present in the demand trace: we capture the workload's reference trace above the buffer cache, so that we can model the effect of changes in buffer cache memory.
- The *disk model* is a simple approximation of a single-spindle storage system without on-disk caching. Track and sector positions are inferred from LBNs (logical block numbers) based on the known disk geometry, and seek and rotational times are inferred from these. Based on this, the disk model is able to simulate the SSTF (shortest seek time first) scheduling policy used by most disks.
- The *lock model* handles requests for locks at various granularities — page, record, etc. — and in different modes — shared, exclusive, shared-intention, etc. It uses the same default policies as the DBMS to make decisions on competing lock grant/upgrade requests.

We are confident that these models, although simple, can provide good accuracy for a wider range of workloads than the analytic models. Our philosophy is to start with these simple models and refine them only if necessary for improved accuracy.

Simulation has a higher overhead than analysis but is still typically much faster than real time: we simulate a CPU computation of arbitrary length in constant time, and a disk access with a few cycles of computation. For in-memory, lock-bound workloads simulation is slower than real time, as the simulator's buffer and lock management are no faster than that of the DBMS.

4 Ongoing challenges

There are many open questions on designing, building, and deploying systems such as the Resource Advisor. Here we present some of these questions and our thoughts on answering them.

Granularity. What is the best granularity to represent resource demands? For example, we represent a CPU "demand" as a single number: the number of cycles of computation. Including information such as L2 cache misses and the integer/floating point instruction mix could allow "what-if" questions about different processor architectures rather than just different clock speeds. However, this finer granularity comes at the cost of increased complexity in instrumentation and modelling.

We envision a need for models at multiple levels of complexity, with the DBA using a "drill-down" approach to increase complexity where needed. For example, crude CPU and disk models might suffice to indicate that a faster CPU would be more valuable than a faster disk. The DBA could then use a more refined CPU model to exactly quantify the performance benefits of different processor upgrade options.

Scope. How much of the system should we model? The key insight that makes performance prediction feasible is that we only need to model those aspects of the system that affect performance and are affected by resource availability. Aspects which are essential to the correct functioning of the system but independent of resource availability can be ignored. For example, when simulating a disk access we need to predict its timing but not the contents of the accessed block.

Thus we must trace the system at a level *above* that of the resource manager but *below* that of any resource-agnostic components, to avoid the complexity of modelling them. For example, in the Resource Advisor, we trace page accesses above the buffer cache rather than below, since the latter will change with the size of the buffer cache. In contrast, we trace the physical execution of query plans, i.e. below the query optimiser. This frees us from the task of modelling the query optimiser and tracing all its run-time inputs. However, it limits us to modelling resource-agnostic query optimisers that are not adaptive to changes in resource availability but make decisions solely based on data statistics and cardinality estimates.

Evolution. When building a Resource Advisor for a legacy DBMS, we chose to insert only passive instrumentation, while maintaining the simulation/analytic models separately. However, this introduces the additional burden of keeping the models consistent with the DBMS components as they evolve. For example, if the lock scheduling algorithm changes in the DBMS, a corresponding change must be made to the lock model. If the code itself is restructured, then the tracing instrumentation points may need to be changed; if this results in a change in the semantics of the traced events, this will cause a change in the models as well. We surmise that tighter integration of predictive models with DBMS components, i.e. making each component truly self-predictive, will help to alleviate this problem. However, we currently lack the programming tools and techniques for developers to maintain a performance model for each component in tandem with its functionality.

Hierarchical models. The drill-down approach also requires us to ask and answer "what-if" questions at different component granularities. For example, the storage component could be a file server with a network RAID back end. For the initial phase of resource planning, the DBA might simply ask "What if the entire storage subsystem were twice as fast?" If the predicted benefits of this look promising, she might investigate different ways to achieve this speedup, for example "What if I made the file server 4-way SMP" or "What if I moved from mirroring to RAID-5?" This hierarchical approach would avoid the need for asking "what-if" questions about all possible hardware configurations.

Administrative boundaries. In a typical 2- or 3-tier architecture, there are multiple components — application servers, database servers, networked storage — typically from different vendors and possibly with different administrators. We could hope that in the future each of these would be self-predicting, but it is likely that they will provide this prediction as a "black-box" functionality that does not expose model internals. Thus the tight integration of different predictive components that we use in the Resource Advisor may not be feasible. Rather than predict the performance of individual resource demands, we might have to process the entire workload trace with the DBMS to create a "storage access trace" and pass that to the storage model to get the timings of the I/Os generated. Since the I/O timings would affect the timings of the entire workload, we would have to iterate this process to converge on a solution.

Distributed modelling. End-to-end performance prediction for large distributed systems is a significant challenge. Individual hosts can efficiently generate local trace information; however, a request in a multi-tiered or clustered configuration might trigger activity on multiple hosts. Backhauling all event traces to a centralised location is a simple but non-scalable solution, and hence we need distributed modelling and prediction algorithms.

5 Related work

Our work on end-to-end tracing in SQL Server was directly inspired by the Magpie project [2], which used end-to-end tracing in 2-tier web services to model workload resource demand and control flow. Our broad aim — automated resource provisioning — is one of many self-tuning scenarios suggested by Weikum et al [8]. Other researchers have investigated self-tuning for other aspects of the DBMS: for example, the DB2 Advisor [4] and the Database Tuning Advisor [1] suggest the most appropriate set of indexes and materialised views as well as the best physical layout of tables.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *Proc. 30th VLDB conference*, Aug. 2004.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[3] J. Gray. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann, 1992.

[4] G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, Feb. 2000.

[5] D. Narayanan. End-to-end tracing considered essential. In *Proceedings of High Performance Transaction Systems – Eleventh Biennial Workshop (HPTS '05)*, Sept. 2005.

[6] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *Proceedings of IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005)*, Sept. 2005.

[7] Transaction Processing Performance Council. Top ten TPC-C by performance. `http://www.tpc.org/tpcc/results/tpcc_perf_results.asp`, Mar. 2005.

[8] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. 28th VLDB conference*, Aug. 2002.

Automated Storage Management with QoS Guarantee in Large-scale Virtualized Storage Systems[*]

Lin Qiao [*] Balakrishna R. Iyer [§] Divyakant Agrawal [†] Amr El Abbadi [†]

[§] IBM Silicon Valley Lab
San Jose, CA
balaiyer@us.ibm.com

[†] Department of Computer Science
University of California, Santa Barbara
{agrawal, amr}@cs.ucsb.edu

[*] IBM Almaden Research Center
San Jose, CA
lsqiao@us.ibm.com

Abstract

Storage virtualization in modern storage systems allows variability in the number of "physical" disks supporting a single "virtual" disk. In practice, IO workloads vary with time. Presuming the evolution of reasonable predictive models with the power of accurately predicting IO workload, it may be argued that it is straightforward to compute the number of disks needed at any time to satisfy QoS constraints. However, because disks contain data, the data also needs to be reallocated amongst disks. Not only does data migration need to be scheduled ahead but it must also be scheduled in such a way that QoS violations do not occur because of the extra migration IOs. In this paper, we present a novel analytic framework, PULSTORE, *for autonomically managing the storage to balance both cost and performance. Given the workload characteristics of an application and storage QoS requirements, our* PULSTORE *framework yields an optimal schedule for adding and removing disks to support a time varying IO workload without QoS violations. A case study based on real IO traces shows that* PULSTORE *is very effective in achieving both QoS and utility goals.*

1 Introduction

Traditionally, storage has been purchased and attached to a single computer system. Such storage is accessible only through the computer system to which it is locally attached. In the last 10 years, especially in corporate data centers, storage is being increasingly purchased independent of the processors, and independently managed and administered. Because of the standardization of disk IO protocols, storage can be easily shared amongst various heterogeneous processors running different applications. The shared storage is accessed over a network interconnecting the processors to the shared disk subsystem, known as the *storage area network* - a network on which processors send IO calls to virtual disks. It is the task of the storage controller to manage the mapping of virtual disks to physical disks, a task known as *storage virtualization*, similar to memory virtualization of processors. The storage virtualization layer has been exploited to provide diverse storage functions.

[*]The work was supported in part by NSF awards IIS-02-23022, IIS-02-20152 and EIA-00-80134. Lin Qiao was also supported in part by IBM Corporative Fellowship.

The storage virtualization layer may be used not only statically but also dynamically to change the mapping of virtual disks to physical disks. This can be exploited to dynamically change the physical disk configuration. Many IO workloads exhibit cyclic behaviors, and alternate between bursts of high activity and periods of low activity. A significant number of practical instances show that the IO workload lends itself to be predictable. For example, the IO workload for a week extracted from HP Cello92 IO traces [11] collected from an HP-UX file server exhibits highly repetitive behaviour. It has also been observed that there is an order of magnitude difference between the peak and average IO rates. Configuring the IO sub-system for the peak leads to over-provisioning and waste. In a virtualized storage management system, it is possible to provision on demand, especially when the storage is shared amongst multiple applications, each of which would have peak requirements at different times.

If a reasonable prediction of IO workload can be made, the storage virtualization layer could optimize the mapping of physical disks to virtual disks to satisfy applications' IO response time requirements. This problem is usually referred to as *on-demand utility provisioning* [10]. This is analogous to the problem of processor allocation to shared concurrent workloads. The difference is that disks contain data. Hence if we change the number of disks supporting a specific collection of data we also need to redistribute the data. This task involves data movement, which generates even more IO. Under some circumstances, it may not always be advisable to change the number of disks. If the number of disks are to be changed, then the change needs to be done in advance. When must it be done? These are questions that will be answered in this paper.

In this paper, we tackle the problems of moving data in a storage hierarchy under both capacity/performance constraints and on-demand resource provisioning constraints. The challenges are two folds. First, for a single data movement action, as it interacts with the applications, we need to limit its impact on application performance. In particular, we need to control the invocation time of a data movement such that there is no performance constraint violation during and after the data movement. Second, as workload varies over time, it is likely that the previous storage provisioning action may have been either too small or too large, resulting in either capacity/performance constraint violation (under-provisioning) or wasteful storage configuration (over-provisioning). Hence it is important to dynamically generate a sequence of carefully tuned data movement actions in order to adapt to the changing workload. To solve these problems, we have developed an analytic framework, namely PULSTORE, to maintain QoS constraints, while avoiding waste of resources.

2 The System Overview

In this paper, we specifically address the problem of balancing the conflicting goals of the storage utility cost and QoSS requirements. Our study is based on a hierarchical storage structure, storage QoS goals, an online data migration model, and a storage utility cost model.

2.1 Storage Architecture

Modern applications access the storage in terms of logical disks, which are then transparently mapped to the physical disks by the storage virtualization engine [3]. Figure 1 depicts the architecture of such a hierarchical storage system. Here the virtualization engine can reside in a storage controller or the storage management layer of a DBMS.

Each layer in the hierarchy contains a pool of identical storage devices, called a *storage pool*. Moving upward through the hierarchy, each layer provides faster access speed but is more expensive. As we move downward through the hierarchy, the cost decreases and so does the access speed. Such a storage hierarchy scheme provides flexible control over the tradeoffs between access speed and cost through data movements. The overall objective of the storage hierarchy is to provide the fastest average access speed with the least expensive average cost of data. Such a storage model is widely used in large-scale data centers [2].

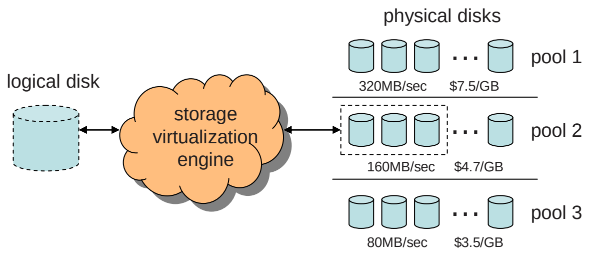


Figure 1: Virtualization for Hierarchical Storage Pools

2.2 Workload and QoSS

An IO workload is a time series of IO requests. It is characterized by the IO request size, IO arrival rate, etc.The storage level quality-of-service (QoSS) specifies the IO performance for a particular storage object that must be guaranteed under any workload. QoSS goals are associated with logical disks [5]. We use L_{QoS} to denote the QoSS goal on IO latency for a logical disk. To enforce the QoSS goals, it is crucial to predict the performance outcome for a given workload and storage system configuration. There are numerous models [4, 7, 1] on models that predict performance outcome. In these models, the IO latency can be represented as a function of the workload and the logical disk configuration. Since these two parameters are a function of time, latency can simply be predicted as a function of time, denoted as $L(t)$. The QoSS goal is expressed as a bounded latency: i.e., $L(t) \leq L_{QoS}$ at any time t.

2.3 Online Data Migration

Data migration is initiated when the storage system experiences performance degradation or anticipates disk failure. Furthermore, data migration needs to be *online*, i.e., no application should be interrupted during data migration. In our model, one logical disk is striped across all the physical devices assigned to it. We study two aspects which directly impact the performance of a striping storage system. We can either increase the physical disk speed or the striping width to decrease the IO latency, or vice versa to increase the IO latency[1]. The change of physical disk speed is realized through the data movement across pools (inter-pool migration), and the change of the striping width is implemented through the data movement inside a pool (intra-pool migration) by adding or removing disk(s) from the current configuration. A migration that reduces the application latency is called *up migration*. A migration that reduces resource utility cost is called *down migration*.

We now describe a typical online data migration scheme, which has been exploited in commercial DBMSs, such as IBM DB2 UDB [6]. The data location is maintained by a data element map. Once a data element is moved from one source disk to another destination disk, the data element map is updated. Therefore, part of the workload is directed to the destination disks and the other part of the workload continues to be directed to the source disks until migration completes.

2.4 Storage Utility Cost

The storage cost over time for a particular logical disk is called the *resource utility cost*, or $\mathcal{U}$. It is the summation of the cost of the provisioned storage system, C_t, at each time t over the whole time period.

$$\mathcal{U} = \sum_t C_t$$

Our goal is to minimize $\mathcal{U}$ while guaranteeing that no QoSS violation occurs.

[1]Changing the striping unit size dynamically for varying workloads has been studied [12]. We do not address this problem here.

3 PulStore

We present the design of a system, named PulStore [8, 9], whose goal is to produce a migration sequence so that the performance outcome satisfies the QoSS specification while minimizing the resource utility cost.

3.1 Requirements

In order to adapt to the variations in workload while providing QoSS guarantees, the following challenges need to be addressed:

- *Guarantee QoSS requirements.* The challenge is to detect when performance will violate QoSS requirements, and determine how to avoid this via appropriate resource re-allocation. Analytical techniques are needed to identify the new resources as destination for data migration, if current resources are not capable of ensuring QoSS guarantees in the future.

- *Ensure minimal resource consumption.* The challenge is that the goal of minimizing utility cost conflicts with QoSS constraints. This goal is to reduce the resources allocated to applications when the system exhibits over-provisioning. However, too aggressive resource reclaim may hurt the performance, and violate QoSS, which is not desirable. Therefore, migrations that reduce resources must be considered together with migrations that increase resources for QoS guarantees.

- *Control migration.* The migration IOs should be controlled to enforce QoSS for the application workload when the migration is ongoing. This is achieved by finding the appropriate time to initiate migration. If the migration invocation time is not chosen carefully, the migration task may never complete due to "migration thrashing", which occurs when applications consume the entire storage bandwidth.

3.2 Single Migration Action

In this section, we analyze how a migration process from source disks, $\mathcal{S}$, to destination disks, $\mathcal{D}$, affects the workload and IO latency on each individual disk. We note that there are two exclusive sets of physical disks, with opposite change in the amount of resident data, which indicates performance outcome, during migration. We call a disk an α-$disk$ if the amount of data residing on the disk is decreasing over time, or a β-$disk$ otherwise. We enumerate three migration cases, as shown in Figure 2. For up migration, *an α-$disk$ is a disk $\in \mathcal{S}$ and β-$disk$ is a disk $\in \mathcal{D} - \mathcal{S}$; for down migration, an α-$disk$ is a disk $\in \mathcal{S} - \mathcal{D}$ and a β-$disk$ is a disk $\in \mathcal{D}$.*

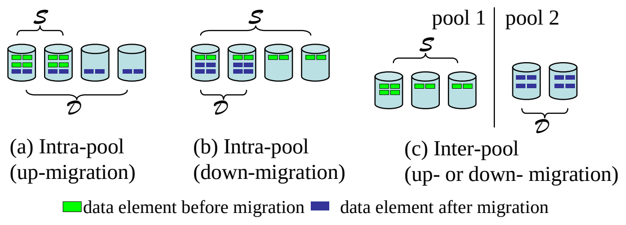

$\mathcal{S}$
$\mathcal{D}$
pool 1
pool 2
(a) Intra-pool (up-migration)
(b) Intra-pool (down-migration)
(c) Inter-pool (up- or down- migration)
🟩 data element before migration
🟦 data element after migration

Figure 2: Cases of Migration

In order to find the appropriate time to start migration, we analyze the relationship between IO workload and migration invocation time. The fraction of I/O workload is proportional to the portion of data residing on the disk based on a uniform IO distribution assumption. In this paper, we use the workload fraction and the data fraction interchangeably. Let $\mathcal{I}$ denote the migration starting time, and t_m denote the migration duration. $p^{\alpha}(t, \mathcal{I})$ and $p^{\beta}(t, \mathcal{I})$ represent the fraction of total data in the logical disk on α-disks and β-disks respectively, at time t, and with a migration starting at time $\mathcal{I}$.

Lemma 1: Given time t, for an $\alpha\text{-}disk$, $p^{\alpha}(t,\mathcal{I})$ is non-decreasing when $\mathcal{I}$ increases, or $\frac{dp^{\alpha}(t,\mathcal{I})}{d\mathcal{I}} \geq 0$, and for a $\beta\text{-}disk$, $p^{\alpha}(t,\mathcal{I})$ is non-decreasing when $\mathcal{I}$ increases, or $\frac{dp^{\beta}(t,\mathcal{I})}{d\mathcal{I}} \leq 0$, on interval $t - t_m \leq \mathcal{I} \leq t$.

In this paper, we study one aspect of the IO workload, namely IO rate. (There are other aspects of the IO workload, such as read IO to write IO ratio, random IO to sequential IO ratio, etc.) We observed that in most cases the relationship between IO rate and IO latency can be described by the Monotonicity Assumption.

Monotonicity Assumption. During the migration process, if the application IO rate to a physical disk increases, the application IO latency increases too. Otherwise, if the application IO rate to a physical disk decreases, the application IO latency decreases too.

Based on this Monotonicity Assumption and Lemma 1, we derive that the latency function for the α-disks, referred to as $L^{\alpha}(t,\mathcal{I})$, is monotonically increasing, and the latency function for the β-disks, referred to as $L^{\beta}(t,\mathcal{I})$, is monotonically decreasing, when $\mathcal{I}$ increases.

Given the relationship between the latency function and migration invocation time, the main challenge now is to determine when to start migration so that during migration there is no QoSS violation.

Since α-disks and β-disks have different performance characteristics, we must ensure that both α-disks and β-disks do not violate the QoSS requirements during migration. Formally, we need to find a valid migration invocation time, $\mathcal{I}$, so that when $\mathcal{I} \leq t \leq \mathcal{I} + t_m$ (i.e., during the migration period), we always have $L^{\alpha}(t,\mathcal{I}) \leq L_{QoS}$ and $L^{\beta}(t,\mathcal{I}) \leq L_{QoS}$. We call the range of valid $\mathcal{I}$ a *safe time zone* for the migration, referred to as $\mathcal{RI}$.

The basic method for finding the *safe time zone* is to first calculate the safe time zone for α-disks and then generate the safe time zone for β-disks. The solution is based on two properties: (1) We utilize the monotonic property (Section 3.2) of $L^{\alpha}(t,\mathcal{I})$ and $L^{\alpha}(t,\mathcal{I})$ on $\mathcal{I}$ to enforce QoSS guarantees; (2) We only examine the latency during a migration. That is, we need to guarantee that for $t \in [\mathcal{I}, \mathcal{I} + t_m]$, there is no QoSS violation. The final *safe time zone* is simply the intersection of safe time zones of both α-disks and β-disks. Function *ActionRange* takes the migration source ($\mathcal{S}$), destination ($\mathcal{D}$), the earliest migration invocation time (t_{min}) and the latest migration invocation time (t_{max}) as input parameters and calculates the safe time zone $\mathcal{RI}$ as output.

4 Global Schedule of Migration Actions

We developed the algorithm to find the safe time zone for a single migration in the previous section. In this section, we extend the scope of the algorithm to handle multiple migrations in a sequence.

The main problem is to find a sequence of migration invocations so that no QoS violation exists in a given time frame and the resource utility is minimized. Recall that one migration invocation is defined as $(\mathcal{D}, \mathcal{I})$. We formally define the multiple migration problem in Definition 2.

Definition 2: Given discrete time points [$t_1, t_2, ..., t_n$], and let the number of disks in the jth pool be k_j, the QoS-aware action sequence is defined as $\Gamma = \{(\mathcal{D}_1, \mathcal{I}_1), (\mathcal{D}_2, \mathcal{I}_2), ..., (\mathcal{D}_m, \mathcal{I}_m)\}$, $t_1 \leq \mathcal{I}_1 < ... < \mathcal{I}_m \leq t_n$, $1 \leq |\mathcal{D}_i| \leq k_j$ in the jth pool. Let $\mathcal{U}(\Gamma)$ denote the utility cost for sequence Γ. For each $(\mathcal{D}_i, \mathcal{I}_i)$, $1 \leq i \leq m$, there is no QoS violation. Γ is an optimal schedule iff $\forall\, \Gamma' \in$ QoS-aware action sequence, $\mathcal{U}(\Gamma) \leq \mathcal{U}(\Gamma')$.

We exploit dynamic programming to find the optimal solution and more details are presented in [8, 9]. The compuational complexity of the optimal algorithm is $O(n^3)$. As n is a measure of time, it is desirable to have an algorithm with lower compuational complexity. We now develop a sub-optimal but practical algorithm based on the above concerns. Compared to the optimal algorithm, it has linear computational cost. The intuition behind the approximate algorithm is that in the case of up migration, given $\mathcal{D}_{i-1}$, $\mathcal{D}_i$ and a migration deadline, we want to postpone the migration starting time as late as possible. This delays the increase in the utility cost. In the case of down migration, we want to start the action as early as possible, so that the utility cost is reduced at the earliest time. The ActionRange function (Section 3.3) finds appropriate migration starting time, $\mathcal{RI}$. Hence, for a single migration, the approximate algorithm picks $\max(\mathcal{RI})$ as the value of $\mathcal{I}$ for up migration, and $\min(\mathcal{RI})$ as the value of $\mathcal{I}$ for down migration. Another aspect of the approximate algorithm is when making the choice for up or down migration, the up migration is considered only when necessary, i.e. when QoS violation is expected, and the down migration is considered whenever possible.

The approximate algorithm will look forward a fixed number of N time steps. Multiple migrations are scheduled in N future time steps, and the decision on one migration will affect the later migrations. Hence, the problem of coordinating a sequence of migrations arises. Consider two migrations happening one after another, after the completion of the first migration, we may not be able to find the safe time zone for the second migration. In this case, we say that the two migrations *conflict*. The above concern motivates looking ahead into the migration sequence invoked in an IO prediction window and adjusting consecutive migrations to resolve conflicts.

5 A Case Study

In this section, we conduct a case study to examine the effectiveness and efficiency of our proposed algorithms.

5.1 Setup

We used I/O traces published by the Storage Performance Council (SPC) [13]. These traces were collected by monitoring requests to disks of an OLTP application at a large financial institution. The IO workload lasts for 2000 minutes. The IO rate varies from less than 200 IOs per minute to more than 12000 IOs per minute.

There are two pools, Pool1 and Pool2, in our storage hierarchy. Each pool contains 3 identical disks. Since we do not consider multi-pool disk assignment, there are a total of 6 disk configurations: 1) D1: using 1 disk in Pool1; 2) D2: using 2 disks in Pool1; 3) D3: using 3 disks in Pool1; 4) D4: using 1 disk in Pool2; 5) D5: using 2 disks in Pool2; 6) D6: using 3 disks in Pool2. We use two linear latency functions for disks in two pools. These two functions are collected from a storage controller (FAStT 900) with different settings. For all the disks in Pool1, their latency function is 6.6 +0.00023*IO rate (msec). For all the disks in Pool2, their latency function is 6.2 +0.00011*IO rate (msec). The first part of the latency represents the disk seek time per IO, while the second part represents the data transfer time. Considering the data transfer time, a disk in Pool2 is about twice as fast as a disk in Pool1.

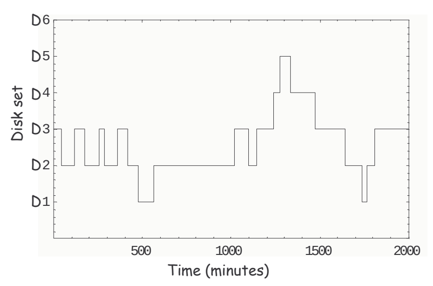
In this case study, we normalize the utility cost of each disk set over the utility cost of D1. For D1 to D3, their utility costs per minute are respectively 1, 2, 3; for D4 to D6, their utility costs per minute are respectively 3, 6, 9. This setting mimics the situation that disks in Pool2 have double the IO processing speed and triple the cost of disks in Pool1.

We set the other critical parameters with the following values: QoS latency requirement is 7.4 msec; number of migration IOs 20,000; migration duration is 30 minutes; prediction window size is 100 minutes.

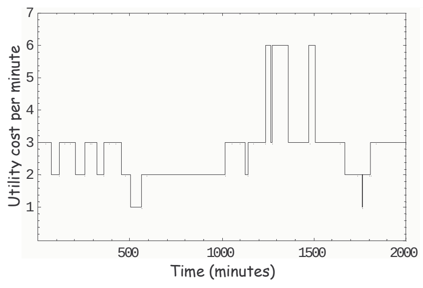
5.2 Global migration sequence

We now analyze the performance outcome of the schemes listed in Table 2 between time 0 and time 2000. Scheme1 and Scheme2 exploit static configurations, and SchemeOpt, the output of optimal dynamic programming algorithm, and SchemeApp, the output of approximate algorithm, provide dynamic configurations. We report the migration sequence and utility cost of all schemes in Figure 3.

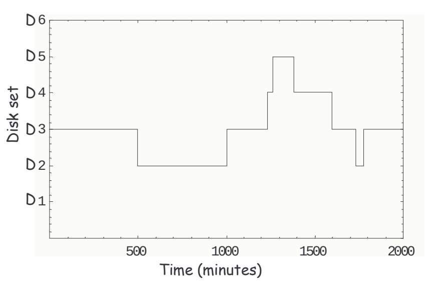
The migration sequences generated by SchemeOpt and SchemeApp are shown in Figure 3(a) and (c). Both schemes initialize the system configuration using D3. SchemeOpt generates the migrations at a finer granularity than SchemeApp. However, SchemeApp accurately captures the trend of the big changes, although it is less responsive to small fluctuation in the workload. Specifically, SchemeApp only runs 8 migrations, less than half



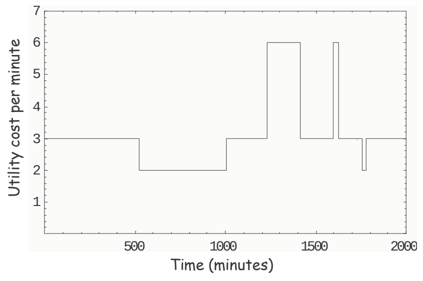
(a) SchemeOpt migration sequence



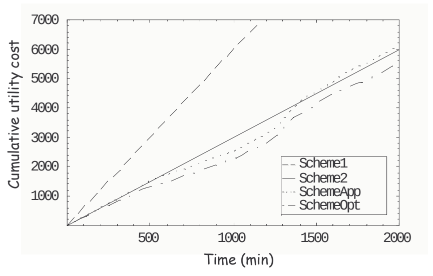
(b) Utility of SchemeOpt



(c) SchemeApp migration sequence



(d) Utility of SchemeApp



(e) Cumulative utility

Figure 3: Performance

Schemes	Policy	Meaning
Scheme1	Using Diskset5 only	Over provisioning
Scheme2	Using Diskset3 only	Average provisioning
SchemeOpt	Using the optimal algorithm	Ideal solution
SchemeApp	Using the approximate algorithm	Practical solution

Table 2: Schemes

of SchemeOpt, which dramatically reduces the amount of data movement over time. Figure 3(b) and (d) depict the utility cost per minute of the outputs for both SchemeOpt and SchemeApp. Note that the shape of the utility cost is different from that of the migration sequence because during a migration, the utility cost is the sum of the utility costs of all the disks participating in that migration. The savings in the utility cost of SchemeOpt is not significant compared with SchemeApp, although SchemeOpt aggressively moves data to minimize utility cost. Figure 3(e) shows the cumulative utility cost. The utility cost of SchemeApp grows at almost the same rate as that of SchemeOpt. The overall utility cost of SchemeApp is merely 10% more than SchemeOpt. By using slightly more utility than Scheme2, SchemeApp provides QoS guarantee. The system using Scheme2 suffers from 400 minutes QoS violation. Although the system using Scheme1 does not have any QoS violation, it incurs twice as much utility cost as SchemeApp (Figure 3(e)). In summary, SchemeApp, the approximate algorithm, efficiently minimizes the utility cost and effectively provides QoS guarantees.

6 Conclusion

PULSTORE provides an automated storage management service which balances the conflicting goals of both QoS guarantees and economical provisioning of resources. As the IO workload changes over time, storage resources need to be re-allocated to satisfy both service goals. In this paper, we developed an analytical framework to schedule data migration in large scale storage management systems. The analytical framework determines safe time zones for invoking migration actions. Using the analytical model, we then present an optimal algorithm for data migration which minimizes storage utility cost, and ensures that no QoS violation occurs. However, the optimal algorithm is predicated on complete knowledge of future workload and the computation cost is huge. Therefore, we present an approximate algorithm that schedules data migration in real time with a fixed-length IO workload prediction. The paper concludes with an analysis of real IO traces and demonstrates that the proposed approach results in significant savings in storage utility cost while preventing QoS violation effectively. More details of PULSTORE can be found in [8, 9].

References

[1] S. Chen and D. Towsley. A performance evaluation of RAID architectures. *IEEE Transactions on Computers*, 45(10):1116–1130, 1996.

[2] E. I. Cohen, G. M. King, and J. T. Brady. Storage Hierarchies . *IBM System Journal*, 28(1):62–76, 1989.

[3] IBM Corp. *AIX Logical Volume Manager from A to Z: Introduction and Concepts*. IBM Redbooks, 2000.

[4] E. Lee and R. Katz. An Analytic Performance Model of Disk Arrays. In *Proceedings of the 1993 ACM SIGMETRICS*, pages 98–109, 1993.

[5] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *Proc. of USENIX FAST'02*, pages 219–230, 2002.

[6] S. Parekh, K. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang. Managing the Performance Impact of Administrative Utilities. In *14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM*, pages 130–142, 2003.

[7] O. I. Pentakalos, D. A. Menasce, M. Halem, and Y. Yesha. Analytical performance modeling of hierarchical mass storage systems. *IEEE Transactions on Computers*, 46(10):1103–1118, 1997.

[8] L. Qiao, D. Agrawal, A. El Abbadi, and B. R. Iyer. Pulsatingstore: An analytic framework for automated storage management. In *International Workshop on Self-Managing Database Systems (ICDE Workshops)*, pages 12–19, 2005.

[9] L. Qiao, B. R. Iyer, D. Agrawal, and A. El Abbadi. Automated storage management with qos guarantees. In *ICDE*, pages 150–152, 2006.

[10] J. W. Ross and G. Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM System Journal*, 43(1):5–19, 2004.

[11] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes, HP Laboratories Technical Report HPL-OSR-93-23, April 1993.

[12] P. Scheuermann, G. Weikum, and P. Zabback. Data Partitioning and Load Balancing in Parallel Disk Systems. *VLDB Journal: Very Large Data Bases*, 7(1):48–66, 1998.

[13] Storage Performance Council:http://www.storageperformance.org.

Early experiences on the journey towards self-* storage

Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger,
James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad,
Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen,
John D. Strunk, Eno Thereska, Matthew Wachs, Jay J. Wylie[*]
Carnegie Mellon University, [*]HP Labs, Palo Alto, CA

Abstract

Self- systems are self-organizing, self-configuring, self-healing, self-tuning and, in general, self-managing. Ursa Minor is a large-scale storage infrastructure being designed and deployed at Carnegie Mellon University, with the goal of taking steps towards the self-* ideal. This paper discusses our early experiences with one specific aspect of storage management: performance tuning and projection. Ursa Minor uses self-monitoring and rudimentary system modeling to support analysis of how system changes would affect performance, exposing simple What...if query interfaces to administrators and tuning agents. We find that most performance predictions are sufficiently accurate (within 10-20%) and that the associated performance overhead is less than 6%. Such embedded support for What...if queries simplifies tuning automation and reduces the administrator expertise needed to make acquisition decisions.*

1 Introduction

The administration expenses associated with storage systems are 4–8 times higher than the cost of the hardware and software [2, 6, 8]. Storage systems are key parts of important data-centric applications, such as DBMSes, hence their high administration cost directly translates to higher costs for the latter. Storage system administration involves a broad collection of tasks, including data protection (administrators decide where to create replicas, repair damaged components, etc.), problem diagnosis (administrators must figure out why a system is not behaving as expected and determine how to fix it), performance tuning (administrators try to meet performance goals with appropriate data distribution among nodes, appropriate parameter settings, etc.), planning and deployment (administrators determine how many and which types of components to purchase, install and configure new hardware and software, etc.), and so on.

Like many [3, 7, 15], our goal is to simplify administration by increasing automation [5]. Unlike some, our strategy has been to architect systems from the beginning with support for self-management; building automation tools atop today's unmanageable infrastructures is as unlikely to approach the self-* ideal as adding security to a finished system rather than integrating it into the system design. We have designed, implemented, and are starting to deploy a cluster-based storage infrastructure (called Ursa Minor) with many self-management features in a data center environment at Carnegie Mellon University.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

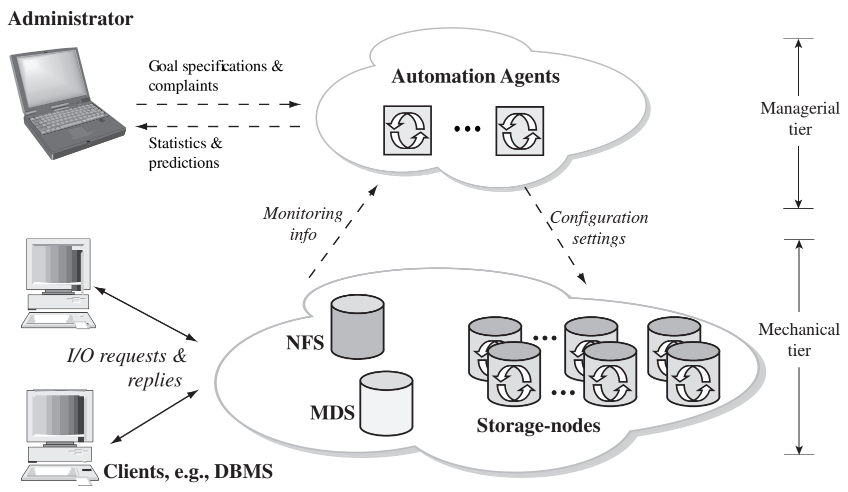


Figure 1: Architecture of Ursa Minor

Ursa Minor's high-level architecture is shown in Figure 1. The design separates functionality into two logical tiers: a mechanical tier that provides storage of and access to data and a managerial tier that automates many decision and diagnosis tasks. This separation of concerns, with clean interfaces between them, allows each tier to be specialized and evolved independently. Yet, the two tiers collaborate to simplify administration. The mechanical tier provides detailed instrumentation and status information to the managerial tier and implements decisions passed down from it.

Ursa Minor's mechanical tier consists of versatile cluster-based storage [1]. We focus on cluster-based storage, rather than traditional monolithic disk arrays, because it can simplify some aspects of administration by its nature. For example, unlike monolithic arrays, cluster-based storage naturally provides incremental scalability. This feature reduces the consequences of not over-provisioning on initial purchases and the effort involved in growth over time—one can simply add servers to the cluster as demand increases. Ursa Minor's data access protocols are versatile, allowing per-object data distribution choices, including data encoding (e.g., replication vs. erasure codes), fault model (i.e., numbers and types of faults tolerated), and data placement. This versatility maximizes the potential benefits of cluster-based storage by allowing one scalable infrastructure to serve the needs of many data types, rather than forcing administrators to select the right storage system for a particular usage at the time of purchase or migrate data from one to another as requirements change.

The managerial tier contains most of the functionality normally associated with self-* systems. It provides guidance to the mechanical tier and high-level interfaces for administrators to manage the storage infrastructure. The guidance comes in the form of configuration settings, including the data access versatility choices mentioned above. Various automation agents examine the instrumentation data exposed by the mechanical tier, as it serves client requests, to identify improvements and solutions to observed problems. These automation agents also condense instrumentation data to useful information for administrators and allow them to explore the potential consequences/benefits of adding resources or modifying a dataset's performance and reliability goals.

This paper focuses on our experiences with one specific aspect of storage administration: predicting the performance consequences of changes to system configuration. Such predictions represent a crucial building block for both tuning and acquisition decisions. Yet, such predictions are extremely difficult to produce in traditional systems, because the consequences of most configuration changes are determined by a complex interaction of workload characteristics and system internals. As such, it is a substantial source of headaches for administrators working with limited budgets.

Ursa Minor supports performance prediction with a combination of mechanical tier instrumentation and managerial tier modeling. The mechanical tier collects and exports various event logs and per-workload, per-resource activity traces [13]. The mechanical tier processes this information and uses operational laws and simple models to support *What*...*if* queries (e.g., "*What* would be the expected performance of client A's requests *if* I move its data to the set S of newly purchased storage-nodes?") [12].

Our experiences with this approach, to date, have been very positive. Instrumentation overheads are acceptable (less than 6%), and prediction accuracies are sufficiently high (usually within 10–20%) for effective decision making. This paper discusses these experiences, some lessons learned, and directions for continuing work.

2 Tuning knobs in Ursa Minor

Like any substantial system, Ursa Minor has a number of configuration options that have a significant impact on performance and reliability. In this paper, we focus on two sets on knobs: those that define the data's encoding and those that decide where to actually place the data once the encoding decision has been made. Both encoding and placement selection involve many trade-offs and are highly dependent upon the underlying system resources, utilization, and workload access patterns. Yet, significant benefits are realized when these data distribution choices are specialized correctly to access patterns and fault tolerance requirements [1]. Expecting an administrator to understand the trade-offs involved in tuning these and to make informed decisions, without significant time and system-specific expertise, is unreasonable. This section describes the encoding and placement options, and the next section explains how Ursa Minor supports choosing among them.

Data encoding: A data encoding specifies the degree of redundancy with which a piece of data is encoded, the manner in which redundancy is achieved, and whether or not the data is encrypted. Availability requirements dictate the degree of data redundancy. Redundancy is achieved by replicating or erasure coding the data [4, 10]. Most erasure coding schemes can be characterized by the parameters (m, n). An m-of-n scheme encodes data into n *fragments* such that reading any m of them reconstructs the original data. Confidentiality requirements dictate whether or not encryption is employed. Encryption is performed prior to encoding (and decryption is performed after decoding). The basic form of *What*...*if* questions administrators would like answers to is "*What* would client A's performance be, *if* its data is encoded using scheme E?".

There is a large trade-off space in terms of the level of availability, confidentiality, and system resources (such as CPU, network, storage) consumed as a result of the encoding choice [12, 14, 16]. For example, as n increases, relative to m, data availability increases. However, the storage capacity consumed also increases (as does the network bandwidth required during data writes). As m increases, the encoding becomes more space-efficient: less storage capacity is required to provide a specific degree of data redundancy. However, availability decreases. More fragments are needed to reconstruct the data during reads. When encryption is used, the confidentiality of the data increases, but the CPU demand also increases (to encrypt the data). The workload for a given piece of data should also be considered when selecting the data encoding. For example, it may make more sense to increase m for a write-mostly workload, so that less network bandwidth is consumed—3-way replication (i.e., a 1-of-3 encoding), for example, consumes approximately 40% more network bandwidth than a 3-of-5 erasure coding scheme for an all-write workload. For an all-read workload, however, both schemes consume the same network bandwidth.

Data placement: In addition to selecting the data encoding, the storage-nodes on which encoded data fragments are placed must also be selected. When data is initially created, the question of placement must be answered. Afterwards, different system events may cause the placement decision to be revisited, such as when new storage-nodes are added to the cluster, when old storage-nodes are retired, and when workloads have changed sufficiently to warrant re-balancing load. Quantifying the performance effect of adding or subtracting a workload from a set of storage-nodes is non-trivial. Each storage-node may have different physical characteristics (e.g., the amount of buffer cache, types of disks, and network connectivity) and may host data whose workloads lead to different levels of contention for the physical resources.

Workload movement *What*...*if* questions (e.g., "*What* is the expected throughput/response client A can get *if* its workload is moved to a set of storage-nodes S?") need answers to several sub-questions. For example, the buffer cache hit rate of the new workload and the existing workloads on those storage-nodes need to be evaluated (i.e., for each of the workloads the question is "*What* is the buffer cache hit rate *if* I add/subtract workload A to/from this storage-node?"). The answer to such a question will depend on the particulars of the workload access patterns and the storage-node's buffer cache management algorithm. Then, the disk service time for each of the I/O workloads' requests that miss in buffer cache will need to be predicted (i.e., for each of the workloads, the question is "*What* is the average I/O service time *if* I add/subtract workload A to/from this storage-node?"). The new network and CPU demands on each of the storage-nodes needs to be predicted as well.

3 Performance prediction support

With hundreds of resources and tens of workloads it is challenging for administrators to answer *What*...*if* questions such as the above. Doing so accurately requires detailed knowledge of system internals (e.g., buffer cache replacement policies) and each workload's characteristics/access patterns (e.g., locality). Traditionally, administrators use two tools when making decisions on data encoding and placement: their expertise and system over-provisioning. Most administrators work with a collection of rules-of-thumb learned and developed over their years of experience. Combined with whatever understanding of application and storage system specifics are available to them, they apply these rules-of-thumb to planning challenges. Since human-utilized rules-of-thumb are rarely precise, over-provisioning is used to reduce the need for detailed decisions. Both tools are expensive, expertise because it requires specialization and over-provisioning because it wastes hardware and human resources — the additional hardware must be configured and maintained. Further, sufficient expertise becomes increasingly difficult to achieve as storage systems and applications grow in complexity.

Ursa Minor is designed to be self-predicting: it is able to provide quantitative answers to performance questions involved with administrator planning and automated tuning. Instrumentation throughout the system provides detailed monitoring information to automation agents, which use simple models to predict the performance consequences of specific changes. Such predictions can be used, internally, to drive self-tuning. They can also be exported to administrators via preconfigured *What*...*if* query interfaces The remainder of this section describes the two primary building blocks, monitoring and modeling, and illustrates the effectiveness with example data.

System self-monitoring: The monitoring is to be detailed so that per-workload, per-resource demands and latencies can be quantified. Aggregate performance counters typically exposed by systems are insufficient for this purpose. Ursa Minor uses end-to-end instrumentation in the form of traces of *activity records* that mark steps reached in the processing of any given request in the distributed environment. Those traces are stored in relational databases (Activity DBs) and post-processed to compute demands and latencies. The monitoring is scalable (hundreds of distributed nodes with several resources — CPU, network, buffer cache and disks) and easy to query per-workload (tens of workloads). The central idea in designing the monitoring is for it to capture the work done by each of the system's various resources, including the CPUs used for data encoding/decoding, the network, the buffer caches, and the disks. There are less than 200 instrumentation points in Ursa Minor. All those points of instrumentation are always enabled, and the overhead has been found to be less than 5-6%, as quantified by Thereska et al. [13]. As a general rule of thumb, we observe that approximately 5% of the available storage capacity is used for Activity DB storage. Different clients' access patterns generate different amounts of traces; the main insight we had from the work on the instrumentation of multiple systems [9, 13] is that it is inexpensive to monitor a distributed system that has storage at its core. This is because the rate of requests to such a system is relatively slow, since the system is usually I/O bound. We find the performance and statistics maintenance cost a reasonable performance price to pay for the added predictability.

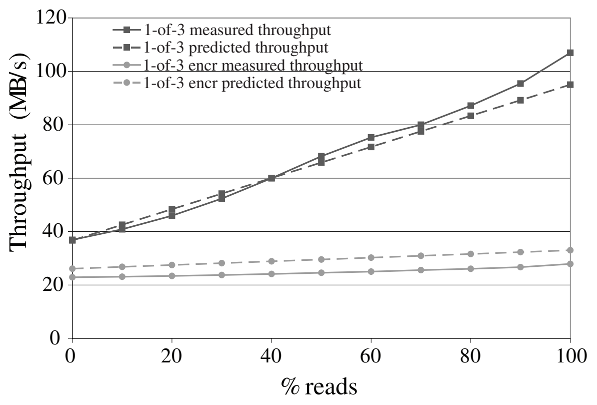
Performance modeling tools: Modules for answering *What*...*if* questions use modeling tools and observation data to produce answers. Tools used include experimental measurements (for encode/decode CPU costs), operational laws (for bottleneck analysis of CPU, network and disks), and simulation (for cache hit rate projections). *What*...*if* questions can be layered, with high-level *What*...*if* modules combining the answers of multiple lower-level *What*...*if* modules. For example, "*What* would be the performance of client A's workload *if* we add client B's workload onto the storage-nodes it is using?" needs answers to questions about how the cache hit rate, disk workload, and network utilization would change. All *What*...*if* modules make use of the observation data collected through self-monitoring.

The basic strategy for making a high-level prediction involves consulting low-level *What*...*if* modules for four resources: CPU, network, buffer cache and disk. To predict client A's throughput, the automation agents consult these resource-specific *What*...*if* modules to determine which of the resources will be the bottleneck one. Client A's peak throughput will be limited by the throughput of that resource. In practice, other clients will share the resources too, effectively reducing the peak throughput those resources would provide if client A was the only one running. The automation agents adjusts the throughput predicted for client A to account for that.

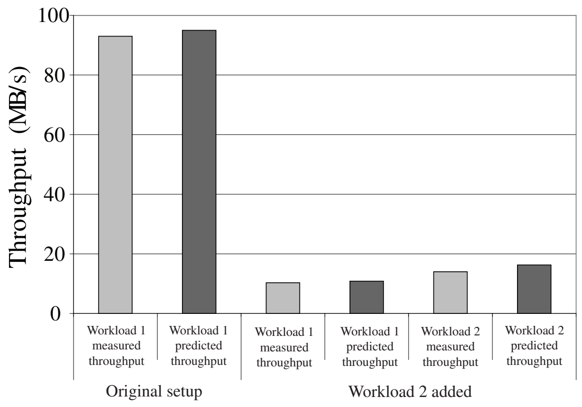
The CPU *What*...*if* module answers questions of the form "*What* is the CPU request demand for requests from client i *if* the data is encoded using scheme E?". The CPU modules use direct measurements of encode/decode costs to answer these questions. Direct measurements of the CPU cost are acceptable, since each encode/decode operation is short in duration. Direct measurements sidestep the need for constructing analytical models for different CPU architectures. The network *What*...*if* module answers questions of the form "*What* is the network request demand for requests from client i *if* the data is encoded using scheme E?". To capture first-order effects, the network module uses a simple analytical function to predict network demand based on the number of bytes transmitted. Intuitively, schemes based on replication utilize little client CPU but place more demand on the network and storage resources (n storage nodes are updated on writes). Schemes based on erasure coding are more network and storage efficient (data is encoded in a "smart" way), but require more client CPU work to encode the data (math is needed for the "smart" way). All schemes require significant amounts of CPU work when using encryption.

The buffer cache module answers questions of the form "*What* is the average fraction of read requests $1 - p_i$ that miss in the buffer cache (and thus have to go to disk) *if* a workload from client i is added to a storage-node?". The buffer cache module can similarly answer questions on other workloads when one client's workload is removed from a storage-node. The buffer cache module uses simulation to make a prediction. The module uses buffer cache records of workloads that are to be migrated (collected through monitoring) and replays them using the buffer cache size and policies of the target storage-node. The output from this module is the fraction of hits and misses and a trace of requests that have to go to disk for each workload. Simulation is used, rather than an analytical model, because buffer cache replacement and persistence policies are too complex and system-dependent to be accurately captured using analytical formulas. The storage-node buffer cache policy in Ursa Minor is a variant of least-recently-used (LRU) with certain optimizations. The disk *What*...*if* module answers questions of the form "*What* is the average service time of a request from client i *if* that request is part of a random/sequential, read/write stream?" The average service time for a request is dependent on the access patterns of the workload and the policy of the underlying storage-node. Storage-nodes in Ursa Minor use NVRAM and a log-structured disk layout [11], which helps with making write performance more predictable (random-access writes appear sequential). When a disk is installed, a simple model is built for it, based on the disk's maximum random read and write bandwidth and maximum sequential read and write bandwidth. These four parameters are easy to extract empirically. The disk module is analytical. It receives the sequence of I/Os of the different workloads from the buffer cache *What*...*if* module, scans the combined trace to find sequential and random streams within it, and assigns an expected service time to each request.

Figure 2 illustrates the prediction accuracy for two high-level *What*...*if* questions the administrator may pose (for the exact setup of these experiments, please refer to Thereska et al. [12]). In general, we have observed predictions accuracies are within 10-20% of the measured performance [12].



(a) **Predicting peak throughput for CPU/NET-bound workloads.** "*What* is the throughput *if* I use encryption (or if I do not)?" The question is answered for different read:write ratios. CPU can be the bottleneck resource when using encryption. The network can be the bottleneck resource if the workload is write-mostly.



(b) **Predicting peak throughput for workload movements.** "*What* is the throughput client 2 can get *if* its workload is moved to a new set of storage-nodes?" The set of nodes contains a workload from client 1 that is sequential and hits in the buffer cache. When workload 2 is added, the hit rate for both drops and the interleaved disk access pattern looks like random access.

Figure 2: Prediction accuracies for two example *What*...*if* questions.

4 Lessons learned and the road ahead

We have had positive experiences with Ursa Minor's two-tiered architecture, particularly in the space of performance self-prediction and its application to self-tuning and provisioning decision support. With acceptable overheads, sufficient instrumentation can be continuously gathered to drive simple models that can effectively guide decisions. This sections expands on some key lessons learned from our experiences thus far and some challenges that we continue to work on going forward.

4.1 Lessons learned

Throw money at predictability: Administration, not hardware and software costs, dominate today's data center's costs. Hence, purchasing extra "iron" to allow self-prediction may be warranted. Ursa Minor utilizes "spare" resources to aid with both self-monitoring and modeling. Spare CPU is used to collect and parse trace records (we measure about 1-5% of the CPU goes towards this per machine). Spare network is needed to ship traces to collection points for processing. Spare storage is needed to store these traces and statistics (about 5% of the storage is dedicated to them). Spare CPU time is also used by automation agents to answer *What*...*if* questions.

Per-client, per-resource monitoring is a must: Exporting hundreds of performance counters to an administrator is counter-productive. Performance counters neither differentiate among workloads in a shared environment nor correlate across nodes in a distributed environment. The instrumentation in Ursa Minor tracks a request from the moment it enters the system until it leaves, from machine to machine. Such instrumentation is the only way to know 1) where requests spend their time, 2) what was the context during which a client experienced a performance degradation, and 3) what are the bottleneck resources for one specific workload in the distributed system.

Separate data collection from usage: We found that there is value in separating the system instrumentation from its use in specific tuning and control loops, rather than tightly coupling the two. This separation has allowed easy data access for new uses of the instrumentation, such as performance debugging. It has also allowed us to continuously refine our notions of what data are needed to make an informed tuning decision.

Rough system models work well: Resources in Ursa Minor (CPU, buffer pool, network, disks) have simple models associated with them. These models are based on direct measurements (CPU), analytical laws (network, disk) and simulation (buffer pool). These resources are complex, especially when shared by multiple workloads (e.g., the disk's performance may range over two orders of magnitude depending on the workload's and disk's characteristics). However, basic modeling works well, at least to pinpoint the bottleneck resource and give bounds on improvement if the bottleneck is removed. Furthermore, rough modeling is usually sufficient to pick one from among four or five possible configurations.

4.2 Research agenda

We are following several research directions toward making storage systems truly self-* [5], including automated data protection, problem diagnosis and repair, and of course tuning. This paper discusses our experiences with one building block: performance prediction support. Even in this one sub-area, several difficult and exciting research issues still remain:

Predicting values beyond the average: We need to develop a common terminology for how to measure predictability (and thus know when we have reached a satisfactory outcome). All our predictions so far concentrate on expected values, or averages. Making predictions about variance requires assumptions about workload patterns (e.g., Poisson arrival times) that may not hold. How can we ensure the variance is predicted within reasonable bounds as well? Can we get a notion of confidence associated with each prediction?

Co-operation with other self-* systems: How will Ursa Minor interact with other self-* systems, e.g., a DBMS that also has self-tuning at its core? The DBMS may decide to do an optimization (e.g., suggest to its administrator to double the amount of buffer cache). That change may alter the workload that Ursa Minor sees, triggering in turn an optimization from Ursa Minor (e.g., Ursa Minor could suggest to its administrator to switch the encoding from 3-way replication to 3-of-5). It is desirable for the combined DBMS$+$Ursa Minor system to be stable, settle on good global configurations and avoid repeating cycles of optimization. Should the DBMS micro-manage Ursa Minor's operations and optimizations, or should the DBMS convey high-level performance goals to Ursa Minor and let the latter take any necessary action to meet those goals?

Integration of legacy components: We built Ursa Minor from scratch and were thus able to insert enough detailed instrumentation inside it to answer the above *What*...*if* questions. However, it is convenient to be able to incorporate off-the-shelf components, such as databases, for various services within Ursa Minor (e.g., a metadata service, an asynchronous event notification service, etc). Is performance prediction possible when such legacy systems are introduced within Ursa Minor? In particular, how will we account for their resource utilization (they may use all four system resources just like clients)? What kinds of *What*...*if* questions can be answered for these legacy components and how fine-grained can they be?

Performance isolation for predictability: Without a basic level of performance isolation in a shared environment with competing workloads, predictions will not be meaningful. Whenever a prediction is made that workload W_n will get X MB/s of throughput (a QoS guarantee), that prediction should not be annulled when another workload W_{n+1} comes inside the system. Although performance isolation for the CPU and network resources is usually straightforward to do (utilizing well-known scheduling techniques), it still eludes researchers for the disk resource, which is traditionally non-work-conserving (the cost of a disk “context switch” is prohibitively high, on the order of milliseconds).

Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun and Symantec) for their interest, insights, feedback, and support. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389, by NSF grant number CNS-0326453, and by the Air Force Research Laboratory via contract F49620-01-1-0433.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] N. Allen. Don't waste your storage dollars: what you need to know, March, 2001. Research note, Gartner Group.

[3] S. Chaudhuri and G. Weikum. Rethinking database system architecture: towards a self-tuning RISC-style database System. *International Conference on Very Large Databases*, pages 1–10. Morgan Kaufmann Publishers Inc, 2000.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.

[5] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[6] J. Gray. A conversation with Jim Gray. *ACM Queue*, **1**(4). ACM, June 2003.

[7] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, **36**(1):41–50. IEEE, January 2003.

[8] E. Lamb. Hardware spending sputters. *Red Herring*, pages 32–33, June, 2001.

[9] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Atlanta, GA, 27–29 September 2005), 2005.

[10] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[11] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–58. USENIX Association, 2003.

[12] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. *International conference on autonomic computing* (Dublin, Ireland, 12–16 June 2006), 2006.

[13] E. Thereska, B. Salmon, J. Strunk, M. Wachs, Michael-Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint-Malo, France, 26–30 June 2006), 2006.

[14] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. *International Workshop on Peer-to-Peer Systems* (Cambridge, MA, 07–08 March 2002). Springer-Verlag, 2002.

[15] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *VLDB*, pages 20-31, August, 2002.

[16] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis, published as Technical Report CMU-PDL-05-108. Carnegie Mellon University, October 2005.

Call for Participation

VLDB2006
32nd International Conference on
Very Large Data Bases

Very
Large
Data
Bases

The 32nd International Conference on Very Large Data Bases
The Convention and Exhibition Center (COEX), Seoul, Korea
September 12-15, 2006
http://www.vldb2006.org/

VLDB is the premier international conference on database technology, organized every year by the VLDB Endowment. VLDB 2006 will be held in Seoul, the biggest city in Korea. A city with a population of 11 million, Seoul has been the capital city of Korea for nearly 600 years and has many historical relics and tourist resorts, attracting over 5,000,000 annual visitors.

The VLDB 2006 has assembled an excellent program that consists of **2 keynote speakers**, **83 research papers, 11 industrial papers, 29 demonstrations, 2 panels, 5 tutorials**, **12 co-located workshops, and the 10-year best paper talk.** For the detailed program, go to the main home page of VLDB 2006.

The registration for VLDB 2006 is now open at the VLDB 2006 home page. Early registration deadline for reduced rates is July 31st, 2006.

VLDB 2006 is sponsored by LG Electronics, Microsoft, IBM, Samsung Electronics, Oracle, SAP, SK Telecom, Google, Naver, HP, Samsung SDS, Asian Office of Aerospace R&D, US Army ITC-PAC Asian Research Office, and Aju Information Technology. VLDB 2006 is hosted by KAIST and AITrc, Korea.

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398