

Qualifying self-maintenance resource costs in Autonomous Systems

William Mitchell

California State University, Sacramento

mitchell@ecs.csus.edu

Abstract

Self- capabilities impose demands on the architecture and runtime of an autonomous system S in proportion to the size and complexity of S's metadata. We consider a single-server for S that runs a hybrid of paradigms such as rule and analytical processing, a relational database backend, and a meta-data repository that undergoes continuous augmentation. Limiting overhead of a schedule $\{J_k\}$ of self-maintenance jobs that support autonomous functionality is a significant challenge. Relational database is the context for examples illustrating the cost/benefit of performance factors influencing $\{J_k\}$ throughput. J_k schedule efficiency is analyzed from the logical to physical levels of S's runtime. As autonomic capability becomes more commonplace, service level considerations will arise. Accordingly, we also advocate the use of assertions in J_k source code for expressing job-scheduling interfaces as well as for detecting resource exceptions during schedule execution. Performance and reliability should be considered together as much as possible during analysis and design of autonomous support.*

1. Introduction

Several commercial database management systems (DBMSs) are testimony to the trend toward injecting autonomous capability into pre-existing runtimes. Lack of human administrative talent coupled with ever-increasing complexity of such systems is a strong motivation for autonomous features in such systems [6], [18].

DBMSs such as DB2 and Oracle illustrate the degree to which such systems have successfully integrated paradigms such as rule processing, object-oriented programming, and sophisticated data analysis. We utilize terminology drawn from the relational DBMS context for our descriptive purposes. The heart of autonomous system support is the meta-data, referred to as the repository R. We consider a single-server (with a conventional computer architecture) having an autonomous system S that combines rule-based reasoning, relational query processing, and continuous access and update of the repository. Although complex, this combination is an industry trend.

By implication, R is physically much larger than available memory. By its nature, a schedule $\{J_k\}$ of self-maintenance jobs runs during “off hours” at elevated privileges and with uncontested access to system resources. In an autonomous system context, the “single-server” assumption requires multiple CPUs and process family instances so that S can tolerate processor loss. On the other hand, the issues in this paper apply whether servers are centralized or nodes in a distributed system.

In order to simplify notation and retain the essence of the problem, assume that each J_k contains a dominant query Q_k that interacts heavily with R, and requires significant resources (processing time and storage) in order to achieve acceptable execution time. Our use of the term query is generic. That is, the Q_k source code could be SQL, Java, Python, or any language suitable for expressing job processing directives.

We limit resource classes considered to CPU, I/O (including external I/O for import/export), and a memory hierarchy storage that includes remote (disk) mass storage. Not considered are network and various inter connect interfaces to base system S. Henceforth, we identify processing of J_k with that of Q_k .

2. J_k -specific resource allocations

Experience with database and related computation has identified two significantly different processing patterns, namely transaction processing (aka OLTP) and warehousing (aka OLAP). OLTP involves interactive database access by humans in many applications. OLTP queries are usually quite short duration and typically access at most two or three tables, and have completion time on the order of milliseconds. OLAP involves complex queries with expensive aggregation and sorting operations, and the query target tables are typically scanned. OLAP serves decision support by ranking, scoring (a million records in less than 5 seconds), and summarizing over huge data sets. It is far less ad-hoc in nature than OLTP and highly scripted. One difficulty when processing schedule $\{J_k\}$ is that different J_k exhibit mostly OLTP, or mostly OLAP, or a combination of these behaviors. Thus, resource allocation for the schedule must be very flexible.

The obvious way to process $\{J_k\}$ is to pre-assign a fixed set of resources RP that is used by all J_k in the

schedule. Even if RP allocated the maximum configurable amount of memory, CPU, and disk resources permitted, it is highly unlikely that RP suits the unique needs of each query Q_k . For example, if one Q_k reads large data sets and does extensive sorting before presenting results, then sort memory and CPU will be dominant resource needs, and memory for buffered I/O is insignificant. On the other hand, if another Q_j does a large amount of updating on many small and randomly-located data, then buffered I/O space will be the dominant resource need and sort space and CPU are insignificant. An obvious schedule improvement is replacing RP with several RP_i that dynamically alter resource configurations for various Q_k .

The platform resource management needed for allocating RP_k resource class settings, as well as rapid re-configuration, has evolved in a sophisticated fashion in commercial systems at both the operating system and DBMS levels. Such capability allows changing many configurations parameters (memory limits, address space boundaries, etc.) without runtime restart. For generality, assume henceforth that an RP_k is specified for each Q_k .

Per- J_k resource optimization benefits can be approximated using elementary queuing models [2]. The average time spent by a job at a single server (including queuing) is given by $(\mu - \lambda)^{-1}$, where μ and λ are mean job service and arrival rates respectively. This expression is based on exponentially-distributed job arrival and service processing. Although schedule $\{J_k\}$ is typically not exponential, this expression indicates the merit of improving the mean service rate for Q_k . A sum of such terms models the cumulative alternation between CPU and I/O elapsed time when there is no significant switching delay and there is little overlap processing. This is a reasonable model for our context when each J_k is not contending with other large jobs. The overlap assumption is made because most maintenance jobs tend to be I/O bound. However, OLTP is exhibiting more CPU-bound behavior as processor rates and caching improve [13].

The flexibility needed for executing the $\langle J_k, RP_k \rangle$ schedule is considerably beyond current operating system support. Thus, environments running on top of the operating system, or S itself, must implement create and modify schedule, cancel J_k , and enforce schedule pre-conditions. Such pre-conditions include completion of dependent jobs, event occurrences such as a data import finished, and so on. Oracle release 10g implements these capabilities with package DBMS_SCHEDULER.

3. Storage format and placement heuristics

The purposes of the jobs J_k are many. Assume that R is structured and implemented as a relational database catalog, aka, data dictionary. Thus, in S, we assume that

both R and application data are accessed using SQL. The size of R warrants use of physical access methods (AMs) such as indexes of various kinds, hashing and its storage layout, and so on. Each AM is intended to accelerate searching for Q_k 's target data and is usually based on key values or heuristic probes. The importance of searching deems it imperative to update AM structures due to continual changes to both R and application data. Inefficient searches must be kept to a minimum at each memory hierarchy level from cache to physical disk. This is because a small inefficiency repeated many times per second or per hour does not scale to large search spaces. A given J_k might involve AM creation, update in place, or movement of data and meta-data, depending on R's architecture. Operations as inane as (Oracle) SQL CREATE TABLE with a few columns with simple built-in data types and a PRIMARY KEY and a CHECK constraint cause insert of one or more rows into at least seven data dictionary views [17].

For jobs J_k that load and assimilate new information into S, it can be beneficial to also store intended use information (as a form of meta-data) as well as the data. Doing so can identify invariant (never to be updated) data that become candidates for compressed storage. Retrievals from large compressed data sets are much faster than an uncompressed counterpart. To illustrate, consider a relational query Q that joins three tables, the largest of which (call it X) is 40% sampled (rather than all rows), and the join result rows are written to a new table. When Q is re-run, after the largest table and its index were compressed, the runtime is 30% less. Figure 1 shows the creation of a compressed index for the key of X, followed by the second execution of Q. Thus, one-time compression overhead can be very cost-effective [16].

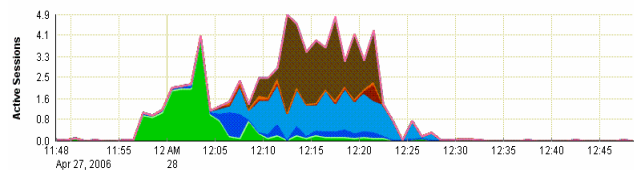


Figure 1. Faster runtime using data and AM compression

Note that all but the last two figures use the following colors and scales for representing session activity counts (vertical dimension) over intervals with a 5 minute tick (horizontal): green CPU, dark blue Query I/O, light blue System I/O, brown Configuration, orange Commit processing, pink Miscellaneous, and red/brown Concurrency. These figures were generated using the Performance displays of the Oracle 10g Enterprise Manager tool. An SMP system running the Sun Solaris 10 operating system ran the queries on Oracle release 10g, version 2. The tables involved in the query examples have integer and string column types. All

quantitative results described are specific to this data and solely intended to illustrate scheduling and resource tradeoffs.

In the schedule, each Q_k executes some sequence of operations that (almost invariably) involves persistent and temporary storage, search, and access. Some operations are notoriously expensive for even the fastest computer systems. Relational join operations are a familiar example. In machine learning, and mining, computing inference probabilities for an arbitrary Bayesian network is another example [15]. For many such operations, it can be beneficial to use physical storage clustering so that future accesses to related data are in proximity.

Finally, data partitioning can be a powerful performance gain, when analyzed and designed properly. Physically separate storage for records having different date ranges (one partition per year, for example) is a typical application of the construct. Overall performance gains from parallel I/O (for OLAP queries) will result from data and index partitions stored in physical separation, by disk and controller. Additional benefit is increased availability as single point of data availability is replaced by multiple points. However, query performance, data availability, and administrative complexity are conflicting goals.

Physical storage layout for data and meta-data involves longer term assimilation issues according to how closely newly loaded data conforms to previously specified and declared properties such as arrival rate and value distributions. Storage initialization parameters should be chosen accordingly so that the need for overflow storage and related expensive re-adjustments is as small as possible. We feel that algorithms aiming to predict future data will continue to fall short of expectations. With respect to processing query Q_k , we do not advocate collecting histories of previous data set accesses. Given that the data are assumed to vary significantly over time, optimal sequences of AM operations themselves must adapt. Thus, the way that a given Q_k is best processed can change significantly over time. For this reason we advocate re-computing RP_k just prior to each Q_k execution. Because Q_k has significant runtime (15 minutes, perhaps much more), the few seconds to compute and allocate RP_k is negligible just-in-time overhead.

4. Q_k operation policy drivers

The application order of operations, often referred to in database terminology as a query execution plan (QEP), must be carefully chosen for queries on large data sets. Relational query languages have achieved impressive and ongoing improvement in QEP generation techniques. However, there is no guarantee that a query optimizer will always determine the best (meaning least costly) QEP for

an arbitrary query. To guard against executing a Q_k that uses a poorly chosen QEP_k, logic should be added to the schedule job processor that checks for a suspect QEP_k.

One example of erroneous query plans arises when the current optimizer strategy/heuristic drivers have a setting “fastest retrieval for any record that satisfying the query” when “fastest retrieval for all records satisfying the query” would be more suitable for a given Q_k . There are similar settings, such as a value in an influence scale (say from 0 to 100) that weights the choice made by the optimizer from among operator implementations such as join, using nested loops, hash, indexed or other low-level algorithms. Another faulty QEP example is a human-coded “hint” that overrides an optimizer decision to use or not use an operation or a structure. (“Hint” is Oracle-specific terminology for such an optimizer directive). In general, a hint might simply be an incorrect strategy directive, and second, the hint might be stale if data and meta-data have changed sufficiently since the previous Q_k execution, thus invalidating a previous (correct) hint. Note that modifying the strategy/heuristic settings or re-coding the source code hint can be done automatically.

We now illustrate hint adjudication for improving speed. Query Q was first run using a hint specifying that the query optimizer should use indexes for two joined tables X and Y. Q was then re-run specifying that the query optimizer compute the QEP without hints, and set the query evaluation policy that executes the query using minimum resources. The second run (Figure 2), in which the optimizer did not use indexes, finished in 27 minutes compared with 40 minutes for the first run.

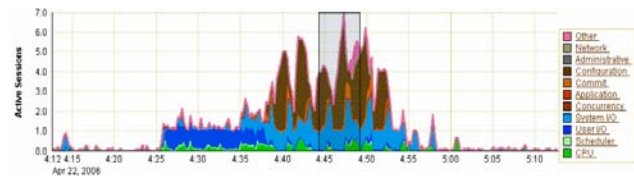


Figure 2. Improved runtime by query hint replacement

Regarding the term “query optimizer”, a cost-based QEP is not a mathematically optimal derivation. A QEP specifies the low-level (data set selection, data field elimination, etc.) operations and their execution order. QEP generation is based on current settings of evaluation heuristics (preference weights, transformation rules, etc.) and available meta-data about target data.

5. Tuning J_k 's physical query operations

This section considers low-level parameters for AMs and S's application data storage. Interaction of these parameters with a QEP strongly influences query runtime.

In the same way that the repository is a useful separation of meta-data from stored data, “logical” storage isolates some properties about object physical

storage from the stored data. Logical storage can specify the unit of I/O processing, initial and incremental storage allocation, physical storage location, and so on. Structuring based on storage type associates the physical storage properties for ordinary data, log data, etc. with different logical containers. In this way, I/O and storage management policies of specially optimized forms can be applied to all physical objects associated with each container. Such containers are bound to physical storage consisting of one or more operating system files. DB2 and Oracle call such containers tablespaces (early relational tablespaces mapped to tables and indexes).

We now consider the lowest levels of the memory hierarchy. The performance of the lowest-level atomic query operations such as physically accessing data for a big join operation, grouping, sorting, etc. is greatly influenced by buffer counts, buffer sizes, work area sizes, proper index structure parameters and so on. Some of these parameter choices apply by default to all objects in given container. Others, such as sort area size and total space allocation for all buffer types, can be set per- Q_k .

Even at this level of processing, automated improvements can be instituted. For example, if a QEP has a large sort operation (as indicated by QEP provided estimates of CPU, I/O, and operation elapsed time), then memory parameters can be re-configured in order to allocate appropriate sort memory. This is not buffer space allocation because sort duration is determined by memory area size dedicated to sorting partitions of the given data.

Figure 3 illustrates the improvements that can be made by applying tuning to the low-level memory parameters discussed in this section. Starting with the same query associated with Figure 2, the I/O parameters that would improve sort and join operations were used (instead of Oracle 10g defaults). The results shown in Figure 3 indicate total query runtime of 23 minutes instead of 27 minutes. An additional improvement is that only 150 MB of temporary disk storage were used during query execution compared with 1.1 GB in the other query. Also discernable from Figure 3 is that, compared with the previous figures, there is very little session activity labeled “Configuration” (log buffer write and wait activities). This indicates additional temporary memory utilization efficiency during execution [10], [14], [9], [5].

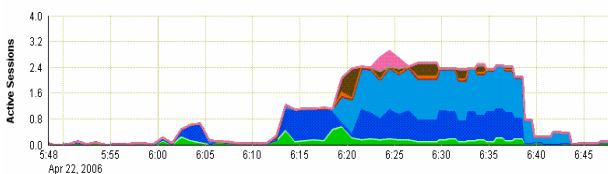


Figure 3. Low-level optimization of hint replacement

Now consider a larger query example named QJSW that joins tables X and Y (20 million and 2 million rows respectively) with a very small table Z. The join result is sorted by 3 columns and the result is written to a new table (approximately 6 million rows). This query is an extension of the queries considered earlier. Multi-level sorting is added as well as access to all of X, rather than a sampled access to a statistical fraction (40%) of its rows. QJSW contained the same inappropriate hint discussed above and illustrated in Figure 2. Figure 4 charts execution of this run. Re-expression of QJSW, named QJSWO (standing for optimized), implemented the same hint improvement as in Figure 2, and also implemented low-level settings for more efficient joins and sorts as in Figure 3. However, this run used no compression or clustering enhancements. Figure 4 shows that QJSW consumed over 75 minutes. Figure 5 shows that QJSWO ran slightly more than 60 minutes. Compression alone reduces total runtime of the un-optimized run by 33%.

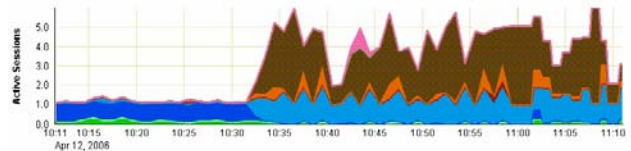


Figure 4. QJSW – 3 table join, 3-level sort, and write

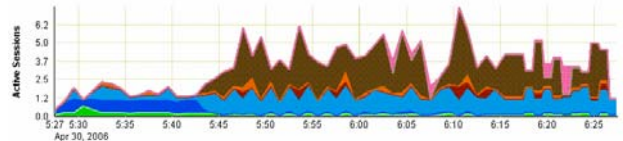


Figure 5. QJSWO - optimized form of QJSW

Our layered approach for improving Q_k throughput takes advantage of any available (and well-founded) tuning offered by the runtime. This is platform and runtime specific, but necessary for overall performance goals. Moreover, although not portable, such tuning usually scales well as data sizes grow [9], pp. 12-15. We note that recent research suggests that more effective caching should be possible in support of sorting, grouping, and other low-level computations [1], [13], [8].

6. Assertions in autonomous maintenance

Techniques for limiting the expense of maintenance schedule execution described in this paper are tempered by the possibility of interference with other background autonomous processes. Performance is one aspect of interference, and system-threatening exceptions are another interacting factor. For example, handling resource exhaustion (such as logical space full) and reconciling corrupt or invalid structures (such as bad physical index) cause their share of problems including

level of service consequences. Thus, the processes supporting autonomous functionality must implement query and job suspend/resume and abort/recover.

The recovery sub system SR has involvement with all updates and inserts. Assume that SR recovery is based on saving changed data in a (disk) log. Note that persistently storing all versions of an item (journaling in the Linux ext3 file system, for example) is becoming commonplace [19]. SR runs in the background during normal circumstances wherein update logging and database vs. log I/O synchronization are the only detectable overheads. The logging subsystem must provision available space to store the next increment of logged records, accomplished by a circular arrangement always having at least one free storage container.

A Q_k with heavy updating creates the real possibility of simultaneous and high log I/O. At the least, Q_k throughput will be degraded, and at worst, the ongoing Q_k execution must be throttled back or suspended. Figure 6 displays a query similar in nature to Figure 3, and Figure 7 shows a (coincidentally simultaneous) large amount of log processing activity that was initiated automatically by background recovery processing.

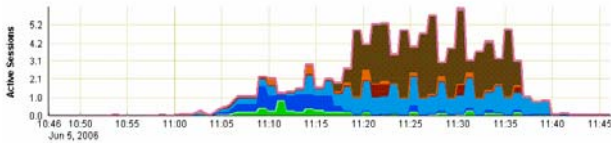


Figure 6. Query optimized for low-level I/O

In Figure 7, the horizontal time scale is hours, and the gray area at the left depicts the log activity. The query and log concurrency causes the query runtime to increase by a factor of 4 (28 minutes without log contention and 110 minutes with it). This example shows the importance of minimizing such concurrency whenever possible.

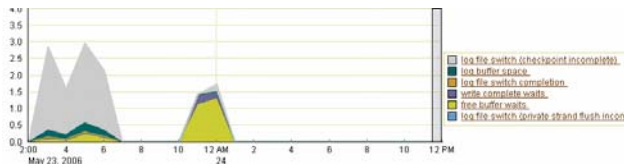


Figure 7. Background database log/recovery processing

Large jobs and processes running concurrently can interact in unfortunate ways. Suspension/resume and other state representations constitute a mechanism for approaches such as “resumable storage” (Oracle version 9i terminology). It is a line of defense for S aimed at avoiding job and query aborts. Resumable storage would declare and enable the job or query to suspend when various storage anomalies arise. When a storage exception happens, the job or query is suspended for a time interval during which SR resolves the situation (by acquiring more storage and/or reorganizing existing

storage). Of course, S would be forced to request human intervention when the storage exception cannot be resolved. Job suspend/resume control should be interfaced with messages, alerts, and many enabled exception types. Although overhead in processing each J_k , the typical number of jobs in the schedule is small.

Defining and enforcing conditions for transitioning J_k among states is analogous to dynamically enforced class assertions as in object-oriented programming languages like Eiffel [11]. Programming language deployment of assertions helps in preventing a referenced class instance from interacting with the rest of its system when it violates assertions or invariants. A violation error is returned to the referencing module. By analogy, we advocate similar execution-time enforcement of an exception management interface for $\{J_k\}$. The pseudo code (Eiffel-like syntax) in Figure 8 illustrates scheduling and exception pre-conditions and assertions for a J_k .

```

-- This job does high insertion rates into R
-- Only user 'SystemMaintenance' can run this job
require uname: username = 'SystemMaintenance';
-- Reusable storage must be enabled
require reusable_store: enable Reusable_Storage = true;
-- Repository AMs must be refreshed every 7 days
require R_current: R_AM_currency < 7 D = true;
-- Remote emergency log capacity minimum needed
assertion rem_log: (remote_log_path = 'alpha/rem_log/'
and alpha_local_log_space > 100 MB) = true;

```

Figure 8. Typical assertions for a J_k

Runtime-enforced assertions have been instrumental in finding elusive bugs and development errors in some very large software runtimes [11], [3], [4]. We therefore encourage their inclusion in autonomous systems because they are an automatic (and self-documenting) methodology for detecting runtime errors. Furthermore, assertions would reduce error cascading (by early detection), and thereby, the complexity of correction and subsequent recovery. They can be expression equivalents of various configurations of diagnosis structures [12]. For example, suppose assertion `rem_log` evaluates as false on entry to a job with excessive updating. The failed assertion signifies that remote log is not available as a mirror or store alternative should the local log fail.

Figure 9 summarizes the considerations in this paper. Not included are potential tuning gains based on how the Q_k Result might be processed in an autonomous context.

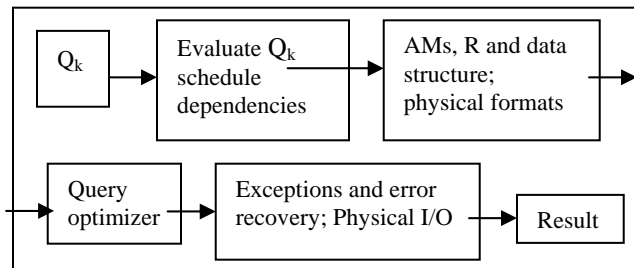


Figure 9. Analysis and design factors for a $\{J_k\}$ schedule

7. Conclusions

Autonomous system support is being integrated into many vendor products. This paper discussed various generic performance and scheduling issues associated with a self-maintenance job schedule $\{J_k\}$ for an autonomous system. The schedule was considered from outer to inner runtime layers and performance improvement opportunities were identified. Selection of AM paths, I/O parameters, and storage options such as compression and partitioning, are generic design issues and apply over any disk architecture and for any large autonomous runtime [7]. However, performance parameters, storage techniques, and access efficiency implementations are platform-specific. In a similar generic sense, we advocate using software assertions for expressing $\{J_k\}$ scheduling interfaces as well as improving runtime error detection. The above generic and platform-dependent distinctions are best treated during analysis and design of $\{J_k\}$ in order to more readily achieve scale up.

Going forward, distributed server nodes must also integrate concurrent and highly unpredictable internet workloads with the single-server issues in this paper. Such runtimes will require extensive research and practical validation to effectively support the terabyte-scale autonomous environments looming on the horizon.

8. References

- [1] S. Adams, "Oracle 8i Internal Services", O'Reilly, 1999, pp. 6-18, 88-111.
- [2] Banks and Carson, "Discrete Event System Simulation", 3rd edition, 2003, pp. 204-243.
- [3] W. Berg, M. Cline, and M. Girou, "Lessons learned from the AS/400 OO project", Communications of the ACM, 38(10), October 1995, pp. 54-65.
- [4] R. Binder, "Testing Object-Oriented Systems", Addison-Wesley, 2000, pp. 807-913.
- [5] D. Bursleson, "Creating a Self-Tuning Oracle Database", Rampant TechPress, 2003.
- [6] S. Elnaffar, W. Powley, D. Benoit, and P. Martin, "Today's DBMSs: How autonomic are they?", Proceedings of the First IEEE International Autonomic Systems Workshop, DEXA, Prague, 2003.
- [7] G. Ganger, J. Strunk, and A. Klosterman, "Self-* Storage: Brick-based storage with automated administration", CMU-CS-03-178, Carnegie Mellon University, August 2003.
- [8] T. Harder, "DBMS Architecture – Still an Open Problem", BTW, LNI, P-65, Springer, pp. 2-28, 2005.
- [9] T. Kyte, "Effective Oracle by Design", McGraw-Hill/Osborne, 2003, pp. 12-15, 216-239, 303-404.
- [10] J. Lewis, "When a script isn't a proof", http://www.jlcomp.demon.co.uk/not_proof.html, April 2005.
- [11] B. Meyer, "Object-Oriented Software Construction", second edition, Prentice Hall, 1997, pp. 379-408.
- [12] J. Musa, A. Iannino, and K. Okumoto, "Software Reliability", McGraw-Hill, 1987, pp. 77-112.
- [13] W. O'Mullane, J. Gray, N. Li, T. Budavari, M. Nieto Santisteban, A. Szalay, "Batch Query System with Interactive local storage for SDSS and the VO", Proc. ADASS XIII, ASP Conference Series, 314, 372 (2004).
- [14] "Oracle Database Performance Tuning Guide, 10g Release 2", Oracle Corporation, 2005, chapters 2-10.
- [15] T. Mitchell, "Machine Learning", McGraw-Hill, 1997, pp. 184-191.
- [16] M. Poess and H. Baer, "Decision Speed: Table Compression in Action", http://otn.oracle.com/oramag/webcolumns/2003/techarticles/poess_tablecomp.html, 2003.
- [17] SQL standards for schemata reference: ftp://sqlstandards.org/SC32/WG3/Progression_Documents/FCDD/4FCD1-11-Schemata-2002-01.pdf, 2002.
- [18] E. Thereska, D. Narayanan, and G. Ganger, "Towards self-predicting systems: What if you could ask "what if"?", 3rd International Workshop on Self-adaptive and Autonomic Computing Systems. Copenhagen, Denmark, August, 2005.
- [19] S. Volker, "Linux on zSeries Journaling File Systems", Share, Anaheim, March, 2005.