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.

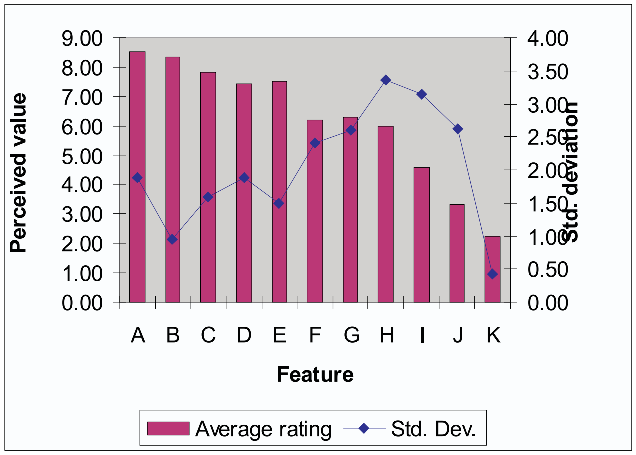

Figure 1: Survey results from DB2 consultants.

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.