Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering

Gerhard Weikum[1], Axel Moenkeberg[2] Christof Hasse[3], Peter Zabback[4]

1 University of Saarland
Im Stadtwald
D-66123 Saarbruecken
Germany
weikum@cs.uni-sb.de

2 Swiss Re
Mythenquai 50-60
CH-8022 Zurich
Switzerland
Axel_Moenkeberg@swissre.com

3 UBS AG
P.O. Box
CH-8098 Zurich
Switzerland
Christof.Hasse@ubsw.com

4 Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
USA
pzabback@microsoft.com

Abstract

Automatic tuning has been an elusive goal for database technology for a long time and is becoming a pressing issue for modern E-services. This paper reviews and assesses the advances that have been made on this important subject during the last ten years. A major conclusion is that self-tuning database technology should be based on the paradigm of a feedback control loop, but is also bound to build on mathematical models and their proper engineering into system components. In addition, the composition of information services into truly self-tuning, higher-level E-services may require a radical departure towards simpler, highly componentized software architectures with narrow interfaces between RISC-style "autonomic" components.

1. Understanding the Problem

Mission-critical information systems require consistently good performance. To this end, virtually all large databases are managed by well paid system administrators who should be experienced in the black art of database tuning: adjusting the system's tuning knobs to the specific workload characteristics of the applications. This involves hardware capacity planning, physical database design, settings for run-time resource management, and the management of inter-system dependencies (e.g., between middleware and database server). Skilled "tuning gurus" are scarce and expensive, and the total cost of ownership for a mission-critical information system becomes more and more dominated by the money spent on human staff. Furthermore, the difficulty of system tuning may incur additional hidden costs by forcing application developers to restructure their application logic in a possibly unnatural manner (e.g., coding stateful applictions into stateless programs) in order to ensure acceptable performance. This situation calls for the automation of tuning decisions and a new generation of self-tuning database technology.

Over the last ten years the awareness of the problem has been growing [Be98] and significant progress has been made towards solutions [DE99]; however, there is no breakthrough yet. With Web-based E-services [DE01] such as auctions, brokerage, service outsourcing, and E-business supply chains, the problem becomes even more difficult and also more pressing.

1.1 Why is Auto-Tuning More Important Than Ever?

The IT industry today is driven by time to market, and software systems are developed and deployed at an amazing pace. As a result, many Web-based E-services are brittle, frequently exhibit inconvenient outages, and have absolutely unacceptable response times during popular business hours (i.e., load peaks). Exceptions are those well administered sites that are heavily investing in their human support staff, to monitor workload and performance trends, identify potential bottlenecks as early as possible, and take correcting actions such as hardware upgrades or adjustment of tuning knobs (e.g., multiprogramming levels).

Most people in the IT industry are aware of the high cost of unavailability. According to business analysts one minute downtime for an E-business site causes damage (through bad impact on the market position) on the order of $ 100,000 [Ora00, PB01]. Moreover, when performance is poor potential customers are not willing to wait for a server's reply; these customers will be lost and are unlikely to ever come back to the site. So lack of performance is as expensive as downtime.

There is growing awareness of the criticality of performance guarantees [BBK00, LGS00], as expressed by Internet performance rating companies such as Keynote or CDN providers such as Akamai. These kinds of compa-

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

nies primarily focus on simple long-term metrics such as average respone time over an entire week or month. However, guarantees along these lines do not reflect the performance during peak periods, the most popular business hours when responsiveness matters most. In traditional OLTP, well-tuned systems are geared to peak load for a very good business reason, but Web-based E-services are still far away from this standard.

1.2 Why is Auto-Tuning Becoming More Difficult in the E-Service Age?

There are a number of technical reasons why tuning of a Web-based E-service application is even more challenging than for traditional TP systems:

- *Sophisticated multi-tier architectures:* The system architecture of a typical E-service site is more complex than a traditional TP system. Both are usually three-tier architectures, but a decisive difference in the new world is that the middle-tier (Web) application servers often manage some persistent data in application-oriented object caches. When freshness of the data or bounded staleness matter these sophisticated caching architectures pose additional tuning problems that are not well understood.
- *Service federations:* Modern E-services offer very rich functionality, which often integrates data and functions from various sites both within and outside the company. For example, Internet-accessible travel services such as Expedia communicate with wholesalers like Amadeus or Sabre who are themselves full-fledged and autonomous E-service sites. The performance of one E-service transitively depends on that of other systems. Even within a single company, where business portals aim to provide a unified view onto different information sources and applications, the underlying systems are often operated and administered by different branches, departments, or subsidiaries, and thus at least semi-autonomous.
- *Workload diversity and variability:* In modern business the variety of functions that need to be supported by an E-service site is much higher than in the traditional TP world. Home banking includes not just simple money transfers, but also portfolio analyses, personalized investment recommendations, what-if scenario simulations, and so on. Whenever such highly diverse application functions require some shared data, global tuning becomes a lot more difficult. A second dimension of workload variability lies in the fact that E-services have about a hundred million potential clients from all over the globe. Consequently, there are potentially tremendous fluctuations in the system load, and this can dramatically amplify the factor between average and peak load.
- *Long-lived applications and long-range workload dependencies:* In traditional OLTP and also in the first-generation E-commerce applications, the focus has been on short-lived interactions that involve say ordering a few books and paying by credit card. In addition, various tricks are used to make most application functions pseudo-stateless (e.g., identifying customers and shopping carts via cookies). This approach simplifies performance tuning; there are no long-term commitments for serving certain requests at user-acceptable speed. The emerging classes of advanced B2B and B2C E-services will include long-lived workflows; for example, you may subscribe to some news feed and pay in installments, or some service may run a personalized, automated agent that acts on your behalf in the stock market, in auctions, and so on. These long-lived applications are much more difficult to configure properly and tune, because admitting the start of a new workflow implies long-range commitments by the server to process all follow-on activities at a specified user-acceptable performance level.

1.3 Message and Outline of the Paper

This paper reviews and assesses the advances that have been made during the last ten years towards the elusive goal of self-tuning database technology. It is largely based on lessons learned from the COMFORT project that we were working on in the early nineties at ETH Zurich. Our approach at that time was centered around the paradigm of a feedback control loop.

The quantitative nature of performance tuning requires mathematical models and their careful engineering into the system architecture. The complexity of the necessary models, the dependencies between different tuning issues, and the intricate interactions between different system components further call for a drastic simplification of the underlying system architecture. We advocate a radical departure from today's monolithic architectures towards a library of simpler, RISC-style, components with very narrow interfaces and very limited interaction between components as the basis for composing components into truly self-managing higher-level E-services.

The paper is organized as follows. Sections 2 and 3 review previous approaches towards automatic tuning in general and in the context of the COMFORT project. Section 4 offers a subjective analysis of where we stand today. Section 5 presents our speculative architectural considerations towards the widely envisioned world of "autonomic computing".

2. Intriguing and Treacherous Approaches

The difficulty of the long-standing problem suggests that there is no easy solution, especially not for advanced E-services. This is in contrast to some developers' thinking that rules of thumb or even brute-force solutions are good enough for automatic tuning.

Sometimes *rules of thumb* do indeed work for certain aspects, namely, when sufficiently generic and robust settings for specific tuning knobs can be found without

quantitative analysis or merely with a simple back-of-the-envelope calculation [GS00]. A positive example would be choosing the size of index pages [GG97]. However, even seemingly simple issues such as choosing an appropriate database cache size for a given workload are so difficult that rules of thumb do not lead to viable solutions. Of course, the five-minute rule dictates a certain minimum cache size based on cost/throughput considerations, but many well-tuned applications use significantly larger caches for better response time (an issue that is not covered by the five-minute rule). Unfortunately, there is no quick-and-easy approach for quantifying the impact of the cache size on the response times of a multi-user workload.

Another approach in which many practitioners believe so much that it is even (mis-)conceived as a panacea is the *"KIWI method: kill it with iron",* that is, upgrade your hardware (more disks, more memory, etc.). Given the low cost of hardware versus the high cost of intellectual cycles, this is indeed a preferable approach, provided that it leads to a viable solution. However, there is a potential pitfall: some tuning problems cannot be solved with hardware upgrades alone or only with outrageously expensive upgrades. For example, an exact-match lookup that is processed by a sequential scan can, of course, be sped up by buying more disks, declustering the underlying table, and exploiting I/O parallelism for faster lookup. But in many cases a much smarter and far less expensive solution would be to create an index on the relevant column(s). Note also that every additional storage or CPU box inevitably creates some additional management work.

When hardware upgrading is the right recipe for a given tuning problem, one needs to be able to calculate how much extra memory, how many additional disks, and other resources are needed to achieve acceptable performance. This in turn requires predicting various performance metrics as a function of the hardware configuration and workload properties. As the relationships between resource settings and response time are inherently nonlinear, and in fact mathematically quite complex, cost-effective use of the KIWI method does require advanced mathematical modeling.

Finally, an intriguing approach that has been pursued in several research projects (e.g., [Br94, We94, He00, St01]), but to our knowledge, has not yet been seriously considered in products, is the concept of a *feedback control loop* where the settings of tuning parameters are continuously adapted to the current workload characteristics and the resulting performance. As this is the principle that we explored in the COMFORT project, it will be discussed in the next section.

3. Lessons from the COMFORT Project

Our methodology in the COMFORT project was twofold: on one hand, we wanted to identify, explore, and understand general principles of automatic tuning; on the other hand, we worked on individual tuning problems that we found interesting and challenging. These issues varied in scope, regarding both time and scale dimension, from short-term online reactions regarding very specific performance aspects to long-term configuration planning regarding global performance metrics. The idea behind this twofold research approach was that the work on individual problems would help us to identify general principles and would also serve as test cases for the general concepts. In the following, Subsection 3.1 reviews our ideas about general auto-tuning principles, whereas Subsections 3.2 through 3.4 look at individual tuning issues, in increasing order of scope. Subsection 3.5 summarizes our lessons learned.

3.1 The Quest for Auto-Tuning Principles

The main principle that we pursued in the COMFORT project is the concept of an *online feedback control loop*. The system continuously observes certain performance metrics, and whenever these exceed critical thresholds the system dynamically adjusts some online tuning knobs. While this general principle is very simple, the difficulties become obvious when one tries to apply it to a real system. It is unclear which tuning knobs one should consider under which conditions, and it is equally unclear how much they should be varied. A naive feedback control loop could overreact to some problems and may lead to oscillation between equally unacceptable system states. To address these problems, we imposed more structure on our approach and divided the feedback control loop into three phases: *observation*, *prediction*, and *reaction*, an OPR cycle for short.

The *observation phase* monitors performance metrics and workload parameters that can be viewed as indicators for a performance problem or a significant shift in the workload patterns. In this step, the most crucial question is which parameters one should look at. While throughput and query or transaction response times are good global indicators, they do not give any clues about the causes of a problem. So in addition to these macroscopic metrics, we investigated which microscopic data one should measure in order to identify the need for dynamic re-tuning. Furthermore, the microscopic metrics could also guide us in choosing the appropriate tuning knob for fixing the problem. Good examples that we will come back to later are the ratio of lock waits to lock requests and disk queue lengths.

The purpose of the *prediction step* is to assess the hypothetical adjustment of various candidate knobs in a quantitative manner. So it needs a quantitative and thus mathematical model for the function

workload parameters $\times$ *knob settings* $\rightarrow$ *performance metrics*.

A knob should be adjusted only when we predict a significant improvement and the system can be expected to remain operationally stable. The prediction also helps to choose the most effective knob if there are several can-

didates. The stability check is important, for we might otherwise improve only certain aspects while introducing a risk of degrading the system in other regards. For example, fixing all index pages in memory would improve index lookup performance but might leave insufficient working space for sorting or hash-based query operators. It turns out that a conservative prediction step is absolutely crucial for a robust feedback control loop. The critical role of this step and especially its need for a mathematical prediction model have often been underestimated.

The last step in the three-phase feedback cycle is the *reaction step*. From a scientific viewpoint this is the simplest of the three steps: when the prediction step gives us a clear recommendation on which parameter should be adjusted by how much, we merely have to turn the knob. However, this online adjustment may create engineering problems: it is not that easy to build a system where all tuning parameters can be dynamically varied while the system is serving its workload. For example, ten years ago no commercial system would have been able to change the amount of memory that is allocated to a hash join or a sort operator while the operation is being executed. Systems have made tremendous progress in this regard (see, e.g., [PCL93, LG98]). However, adaptable run-time *mechanisms* are merely a necessary prerequisite for dynamic tuning, they do not provide intelligent *strategies* for automatic tuning.

3.2 Example: Load Control

One of the individual tuning issues that we studied in detail within the COMFORT project was load control for locking; see our VLDB 1992 paper [MW92] and also [MW91, We94]. The problem here is that excessive lock conflicts may lead to thrashing [TGS85]: a performance catastrophe where transaction throughput drops sharply and response times become absolutely unacceptable and practically approach infinity. Of course, such dramatic behavior may occur only with update-intensive workloads that exhibit an unusually high degree of data contention (i.e., lock contention when locking is used for concurrency control). Note that the thrashing behavior is not necessarily caused by deadlocks; it is possible with deadlock-free access patterns and caused by transitive lock waits: transactions that are waiting for a lock may themselves block other transactions, and the frequency of such cases may abruptly increase so that all but a few transactions become blocked.

Although data-contention thrashing is certainly not a common everyday problem, there are real applications, for example, in banking and financial trading, where system administrators are concerned about it (even when fine-grained, row-level locking is used). In other applications with significant data contention, the problem has been worked around by either restructuring the application or by running transactions under relaxed isolation levels. The former incurs high costs in the application development, the latter bears a potential risk of overlooking possibilities for incorrect application behavior.

The run-time tuning knob that is usually considered in situations with high data contention is the *multiprogramming level (MPL)*, the maximum number of transactions that are admitted for concurrent execution. Unfortunately, it is not easy at all to find an appropriate setting of the MPL limit for a given workload. If it is set too high, the system becomes susceptible to thrashing; if it is set too low, unnecessarily many transactions wait too long in the system admission queue, and this waiting time is part of the user-perceived response time. Even worse, typical workloads consist of a mix of different transaction types with widely varying lengths, both time-wise and in terms of the number of accessed data items that need to be locked. For such mixed workloads a global MPL limit is actually inappropriate; rather one should limit the number of concurrent transactions for each transaction class and also critical combinations of transaction classes. Such sophisticated tuning options were supported by some TP monitors, but this is obviously a tuning nightmare.

Our self-tuning approach to this problem was a feedback control loop with short-term reactivity built into the transaction manager. A key problem for the *observation* phase was finding an appropriate metric that would indicate the level of lock contention. Our first attempt was based on the *fraction of blocked transactions*, a metric considered already in the analytical work of [TGS85]. However, it turned out that this does not sufficiently reflect the influence of the variability in transaction lengths. Our intuition led us to a metric that implicitly assigns weights to transactions in proportion to the number of locks that a transaction holds. This metric is the *conflict ratio*:

$$\frac{\#\,locks\ held\ by\ all\ transactions}{\#\,locks\ held\ by\ non-blocked\ transactions}.$$

We observed experimentally that, regardless of the actual transaction mix, a value for this metric around 1.3 indicates very high thrashing danger. We coined this value the *critical conflict ratio*. Our somewhat pragmatic findings were later nicely confirmed by an elaborated mathematical model by Thomasian [Tho93].

In the *prediction* step we calculate the probability that a newly admitted transaction would become blocked, and based on this value, the expectation of the conflict ratio if the new transaction were admitted.

Finally, the *reaction* phase included three mechanisms: admission control, cancellation control, and restart control. The *admission control* simply rejects new transactions and places them in a transaction queue whenever the conflict ratio is above or is predicted to exceed the critical threshold. It turned out that this measure alone could not guarantee the avoidance of thrashing, as freshly admitted transactions could extremely quickly and unpredictably drive the system from the safe state into the criti-

cal region. Therefore, we introduced an additional *cancellation control* that aborts transactions and puts them in the queue for later restart whenever the conflict ratio is too high. These victims were chosen based on their numbers of currently held locks and previous aborts (to prevent starvation), and only blocked transactions that blocked other transactions were considered as candidates. The *restart control* re-admits a cancelled transaction only after the transactions for whose locks it was waiting in its prior incarnation have left the system.

We carried out extensive experiments with various variants of the load control method including some competitors [CKL90,FRT91,HW91]. Our best variants outperformed the competitors by a substantial margin, and the performance difference to setups with a static MPL limitation or no load control at all was dramatic. The bottom line was that the feedback control loop approach did indeed work very well for this tuning issue, but to make it robust a number of non-obvious, subtle and critical, details had to be worked out.

3.3 Example: Dynamic Data Placement

Another tuning issue that we looked into in more detail was data allocation and dynamic migration for parallel disk systems [WZS91,SWZ93,SWZ98]. A main goal was to place data onto disks such that at all times each disk has approximately the same load. Despite the ample literature on such load balancing problems, we hardly found any work that would tackle the practical problem of workload patterns evolving over time. So previously cold (i.e., infrequently accessed) data may become hot (i.e., frequently accessed), and vice versa. Our approach to this problem, which we coined *disk cooling*, was again based on a feedback control loop. Compared to the load control issue this problem is of broader and mid-term scope: it affects the entire storage system, and the time scale for data migrations is hours or possibly minutes, but not seconds. Algorithmically, our method may be characterized as an online algorithm based on greedy heuristics.

In the *observation* phase we monitor the heat of files, extents, and blocks. Heat, originally proposed in [Co88], is an estimation of the access rate to a data object (i.e., the number of accesses per time unit). It is based on a sliding window that tracks the time points t_1, ..., t_k of the last k accesses to a data object. Then $k / (t_k - t_1)$ is a dynamically adjusted maximum-likelihood estimator for the stationary access rate of the object. (The same principle is used in the LRU-k caching algorithm [OOW93].) The heat metric is an abstraction of the actual disk load induced by an object as it does not distinguish many small versus few large requests (e.g., random access to a single block versus sequential accesses to physically consecutive blocks). Its key point, however, is that it is additive: the heat of an entire disk simply is the total heat of the data objects that reside on the disk. We consider the load of a parallel disk system as imbalanced if the load of the hottest disk exceeds the heat of the least utilized disk by some specified margin.

In the *prediction* step, we assess the benefit and cost of possible data migrations. As migration candidates we selected movable objects (e.g., extents) from the hottest disk in ascending order of temperature, where temperature is the quotient of heat and size [Co88]. The rationale for this selection criterion is to relieve the hot disk significantly while at the same time aiming to minimize the additional run-time cost of the migration itself. The disks onto which the objects are moved are chosen in ascending order of disk heat, subject to possible placement constraints. Before a data migration is actually initiated, a simple queuing model is evaluated to check if the migration step would not cause undue performance degradation for the already overloaded source disk. We also studied generalizations of the approach where the statistical profile captures periodic load patterns for certain classes of data objects; in this case we include a more elaborate benefit/cost analysis before invoking a data migration [SWZ93].

The *reaction* step of the disk cooling method is fairly straightforward. It merely requires online movement of data objects with proper maintenance of bookkeeping structures such as extent tables. In the early nineties such online reorganizations were still widely considered infeasible, but today all industrial-strength systems should have the necessary mechanisms [DE96].

Disk cooling is applicable to the file, extent, or block level of a storage system, or to the level of table fragments in a parallel database system. In addition to only migrating data, replication or caching of fragments could be added to the repertoire. Even generalizations to distributed data placement, say in a content delivery network, are conceivable and intriguing. In retrospect, however, one point that we underestimated and should have addressed earlier is the interaction of such self-tuning disk reorganizations with caching, indexing, and query processing. Some approaches along these lines have been explored in the literature (see, e.g., [Vi99, We99, KFD00, Lee00]), but these interactions are so complex that a comprehensive solution is still beyond the current state of the art.

3.4 Example: Workflow Server Configuration

A final example where we looked at the broad scope of configuring an entire system for long-term performance guarantees is our recent work on auto-tuning of workflow management platforms [Gi00,Gi02]. Such platforms are complex distributed systems that include one or more workflow engines, application servers, and middleware components like request brokers and message queue servers. For load distribution and better availability one or more of these servers may be replicated on different computers, and this gives rise to the question of how many replicas of each server type one needs in order to satisfy

specified goals for response time, throughput, and tolerated service downtime.

Although the nature of this long-term system configuration problem is quite different from that of the previous two tuning issues, our approach can be cast into the feedback control loop framework. The *observation* step collects statistics about activity invocations, durations, and control flow path frequencies; this is the input for a Markov reward model from which the expected load for the various server types is derived. Assuming a given system configuration, the *prediction* step computes the maximum sustainable throughput and the average turnaround times of the various workflow types, and it uses queuing models to assess the expected waiting and response times for interactive workflow steps. Similarly, the expected downtime is computed from a Markov availability model, based on estimated failure and restart rates of the different server types. Finally, the *reaction* step iterates over possible hypothetical configurations, uses the prediction models to assess each of these candidates, and searches the lowest-cost configuration that satisfies all performance and availability goals. The result is suggested to the system administrator as a recommended reconfiguration (typically involving hardware upgrades or re-allocation of software servers on the existing computers).

Compared to the examples of the previous two subsections, mathematical models play a much bigger role here; the long-term nature of the problem allows us to use much more sophisticated and computationally expensive techniques. Furthermore, the metrics that we aim to optimize are end-to-end performance measures, as opposed to the low-level metrics of the previous examples. In addition to user-perceived response times and availability, we also consider the so-called *performability* metric [HLR00] as an optimization goal, reflecting the impact of performance degradation caused by transient outages of some server replicas. This takes into account even planned outages for maintenance. A technical difficulty in this regard is to capture such regular events with the exponentially distributed state residence times that underlie a Markov model; the solution expands states into subnets such that the state residence time follows a generalized Erlang distribution and can thus approximate constant times as well.

3.5 Lessons Learned

Summarizing the discussion of individual tuning issues, here are the most important lessons that we learned:

+ The feedback control loop is indeed an appropriate framework for developing self-tuning algorithms and system components. However, it is far from being a panacea. While the framework provided general guidance, working out the details for the specific tuning issue at hand was the most difficult part.
+ Mathematical models are a key asset for making feedback-driven methods robust, aiming to minimize the risk of overreacting to fluctuations. Even fairly crude models were helpful, but for strategic tuning decisions such as adding servers to an E-service platform more sophisticated models seem to be in order. In contrast, a prediction step solely based on rules of thumb or without any quantitative extrapolation at all does not seem to be viable.
+ Gathering statistics about workload properties and performance metrics is another key asset. Ten years ago, the associated overhead seemed to be prohibitive, but today extensive statistics are affordable. However, this does by no means imply that the implementation details of the statistics management are second-order issues.
+ Even if some tuning decision such as re-configuring an entire system is not completely automated, having observation and prediction components in the spirit of self-tuning methods is of great value. Alerting a system administrator about increasing load or significant changes in workload patterns as early as possible can often save days of unacceptable performance degradation before the admin has realized and analyzed the problem. In addition, automatically narrowing down the possible bottlenecks and recommending appropriate remedies can drastically simplify the admin's job. So even a partial solution to automatic tuning is a big gain.
+ In a similar vein, there is benefit in replacing a delicate tuning parameter for which appropriate settings are hard to find by another tuning parameter that has more robust settings that are acceptable across a wide range of workloads. Our load control method can be viewed this way: we substituted the highly workload-sensitive MPL limit by the fairly robust critical conflict ratio.

The above positive results need some words of caution, however:

! Mathematical models may become extremely complex and computationally intractable. In such cases two complementary ideas can be pursued.
  First, we can aim at conservative approximations of the quantitative behavior by modeling strategies that yield (reasonably tight) upper bounds for the performance of the actual implementation. For example, when we investigated quality-of-service guarantees for the scheduling policy of a mixed-workload media server [NMW99], we developed a stochastic model for a different, analytically tractable, scheduling policy whose performance was consistently (slightly) worse than that of our best policy. Alternatively, we can resort to simulation (with proper statistical confidence) for submodels, and then use their results for the derivation of higher-level metrics. The simulation could even be an online component if its overhead is sufficiently low; a possible example could be a cache hit rate estimator that is dynamically fed by actual access traces and may suggest cache resizing.
  Second, we can aim to simplify the workload and thus the behavior of the component that we study, typically

by introducing dedicated resources for a workload class rather than aiming to exploit dynamic resource sharing to the largest possible extent. Placing different categories of data on separate disk or servers is an example. This consideration can be viewed as an instance of the "pick the low hanging fruit" engineering rule: try to achieve a 90-percent solution with 10 percent of the intellectual effort and complexity.

! It is often crucial to limit the space and time overhead of the observation and prediction steps, whenever statistics consume memory (e.g., the heat bookkeeping for disk cooling) or computations fall into time-critical inner loops of the system (e.g., the estimation of the conflict ratio). So precomputation techniques, incremental evaluation, and efficient approximations are key to viable solutions. In addition, the overhead of the statistics management must be predictable; we do not want the overhead to vary heavily with load bursts.

! The reaction phase bears an inherent risk of overreacting and ending up with unstable (e.g., oscillating) adjustments of tuning knobs, especially after major shifts in workload patterns. For the individual tuning issues that we studied we managed to make the feedback-driven iterative adjustments robust. However, this was achieved by additional techniques and tricks on a case by case basis (e.g., introducing the critical conflict ratio). It would be desirable to obtain better insight in this issue from a control-theoretic viewpoint and be able to guarantee fast convergence to stable settings after workload shifts.

Finally, we also gained some essential negative insight:

– Even if we fully understand a specific tuning issue and know how to automate it, our understanding of how different components and their tuning knobs interact with each other is extremely limited, and the same holds for the possible interference of different workload classes. This interaction can make the entire system unpredictable and unstable, and is the key problem to be addressed much more intensively in future work. We believe that progress along these lines requires also rethinking the architecture of database and information systems, and will address this point in Section 5.

4. Where Do We Stand Today?

In the last ten years academia has detected automatic tuning as an interesting research topic and the database industry has intensified its efforts on manageability aspects. So, although the topic is still way underappreciated relative to popular megathemes such as XML, there has been steady and incremental progress. Today a number of important tuning issues are sufficiently understood while others still pose major challenges. In the following we offer our subjective assessment of the state of the art. We divide this discussion into obvious, simple, difficult but solved, and still challenging issues.

Among the *obvious* improvements we can observe:

+ Second-order tuning knobs, those that have only minor performance effects (e.g., group commit timers or log buffer sizes), have been largely eliminated from the admin interfaces offered by products. This includes virtually all logging- and recovery-related knobs (see, e.g., [La01]).

+ A growing number of tuning knobs come with reasonable default values (e.g., extent sizes, prefetching sizes for table scans) that may even be automatically calibrated based on the underlying hardware (and OS) configuration.

As for the second category of relatively *simple* recipes, most systems have adopted the following approaches:

+ Employ robust rules of thumb where existing. For example, there is no need for having the page size as an application-specific tuning option, and even striping units can be robustly chosen such that the resulting disk performance, say page access throughput, is within ten percent of the optimum. This has been realized in virtually all commercial systems.

+ Employ the KIWI method wherever applicable. This holds mostly for disk I/O bottlenecks. When adding disks or increasing memory can improve performance proportionally to the added resources, then this is a very cost-effective tuning measure. It is important, however, to realize the limitations of this approach, too. In this regard, mathematical models are crucial to predict the cost-effectiveness of additional hardware.

+ Remove tuning options that offer only marginal gains even under the best possible conditions. A candidate along these lines could be dropping hash indexes or join indexes from the repertoire of the database systems (i.e., provide only B+ tree indexes on single tables). This aggressive simplification is obviously debatable. In particular, it is in the eye of the beholder to define "marginal gains": for some this is a factor of two (and then hash indexes should be dropped), for others even ten percent improvement is worthwhile.

+ Remove tuning options that require black magic anyway. This refers to knobs that even experienced system administrators find almost impossible to cope with in an adequate manner. A candidate would be indexes on multi-table clusters. Although this option looks intriguing for certain access patterns, it is very difficult to assess the adverse impact that it may have on other patterns. In our teaching experience with practical assignments on physical database design, no student could ever make good use of multi-table clustering.

None of the above points requires deep scientific research. In contrast, some advanced tuning problems were difficult, but are now *solved*, leveraging research achievements of the last decade, most notably, in the area of physical database design:

+ The tuning issues at the disk storage level are essentially solved, based on a mix of rules of thumb (e.g., for striping units), the KIWI principle (e.g., mirror every-

thing), mathematical performance models, and some adaptive techniques that came out of research (see, e.g., [Al00]).

+ Index selection is a solved problem [DE99]. By combining exact optimization for subproblems with heuristics for search space pruning, the problem has become tractable even for large databases with many tables and complex workloads. Virtually all commercial database systems have an indexing wizard that recommends which indexes one should create, and it is very hard for a system admin to outsmart these tools. Note, however, that such wizards completely disregard the impact that the index selection may have on concurrency control and resource contention (and they may also ignore exotic features such as multi-table clustering and other fancy join indexes).
+ The selection of materialized views is mostly solved. Here the interaction of the physical database design with the query optimizer is more sophisticated, as one should consider only candidate views that will be exploited by the optimizer-generated execution plans. Physical design assistants that come with database system products have adopted recent research results on these problems (e.g., [TS97, ACN00]). Some assistants also provide advice on data partitioning for parallel database systems (e.g., [Ra02]).
+ The proper choice and organization of statistical data that a database system should maintain for the query optimizer, say histograms of some type or more general kinds of data synopses, now appears to be sufficiently understood [CN01,Ja01,KW02]. There are remaining open issues of the kind when exactly wavelets are superior to some alternative transform or other representation, but these issues are not likely to have a first-order impact. The main problem of deciding on which attribute combinations synopses should be built and how much memory they should be given is mostly solved. Nevertheless, it will take a while for products to adopt these results, and there is surely a nontrivial engineering part for integrating the best techniques into the database engines.

Despite these significant advances, we are still left with a number of *challenging* and largely unsolved tuning issues:

? Sufficiently accurate predictions of query run times and result sizes, and the generation of approximate query results with bounded errors and guaranteed confidence. This is important beyond selectivity estimations for choosing the best query execution plan, as it affects the scheduling and prioritization of concurrent queries, the choice of data sources for queries posed to a mediator, and even user interface issues about fast approximative results versus complete and exact results with non-interactive response times (e.g., in a scientific data analysis environment).
? Query optimization in general is still a big problem. Even with better statistics (see above) there are many potential error sources that can lead to poor execution plans, in particular, the complex interplay of a large number of (heuristic) rewriting rules, the inaccuracy of cost models for modern computer architectures with cache hierarchies, and the uncertainty about the available run-time resources and possible resource contention.
? Memory management. Flexible mechanisms for dynamically adjusting the size of query working spaces and cache areas are in place, but good policies for online optimization are badly missing.
? Transaction isolation levels. Although it is customary to use sub-serializability levels, hardly anybody really knows all the possible implications that the "relaxed" options could possibly have (with very low but non-zero probability). Despite some interesting work in this direction [Iso01], a comprehensive theory for when it is safe to use which isolation level is badly missing.
? MPL limitation, admission control, and scheduling in general are solved for conventional OLTP workloads, but are still widely open for mixed workloads with both OLTP and OLAP parts and potential contention on data as well as memory, disks, and processors. This holds also for memory management, including shared caches as well as query workspaces, where significant conceptual progress has been made (see, e.g., [BCL96,We99]), but advanced issues still remain open.
? Hardware resource configuration is still challenging unless one has a detailed and precise characterization of a static workload as input for appropriate capacity planning tools. For dynamic re-configuration because of evolving workloads there is little support.
? Finally, at the meta level, we are missing a theory, or at least some principles, of how to cope with tuning knob interactions. How can we ensure that knob settings in one component do not have undesired side effects on another component?

5. Where Do We Go From Here?

5.1 The Vision of Autonomic Computing

Despite various attempts to promote automatic tuning and enhanced system manageability as key objectives on our community's research agenda (see, e.g., [Be98]), the subject has mostly been underappreciated and not well covered. These days a new bold vision of *"autonomic" computing* is emerging as a new strategic goal for computer science and the IT industry (see, e.g., [IBM01]). In the envisioned world of automatic and autonomous components all building blocks and entire systems should be self-organizing, self-configuring, self-inspecting, self-optimizing, self-protecting, and self-healing. So this theme is even broader than the goal of automatic tuning and encompasses issues such as dealing with denial-of-service attacks and other forms of sabotage or maintaining and adapting software for billions of embedded devices in the anticipated world of "ambient intelligence". Auto-

nomic computing aims at end-to-end service assurance and spans many areas from storage systems and grid computing to multi-agent technology and Web services.

Of course, one may argue that this new wave merely advertises old wine in new bottles. On the other hand, the attention that the topic is receiving also demonstrates the growing awareness about system manageability problems. The potential danger that lies in such broadening is, however, that research overly focuses on high-level "big picture" approaches such as biological computing or agent technology as if they were panaceas, and again underestimates the difficulties that lie in the technical details. In the next subsection, we suggest a more mundane rationale for addressing the tuning-related parts of the great vision.

5.2 The Case for RISC-style Components

On the path to the bold vision of trouble-free, autonomic systems the following question arises: is the goal feasible at all with the high complexity of today's information systems, or do we need a radical departure from current system architectures as they might be inherently inappropriate for automatic tuning? In this subsection, we consider, in the sense of a Gedankenexperiment, the second alternative, following arguments from [CW00].

Database systems have become extremely complex and overloaded with features. Furthermore, they are typically packaged as monolithic systems (although they are componentized internally). This poses serious manageability problems for application classes like ERP, OLAP and data mining, and also Web-based E-services. Database systems have a very poor gain/pain ratio: they create much pain about having to install, administer, and operate the system, whereas the gain of simplifying the application development by using a full-fledged database system is sometimes questionable. Note, for example, that SAP R/3 considers the underlying database system more or less as a mere storage manager [Mu99].

Following role models such as automobiles or aircrafts where dependable engineering appears to be much more advanced, we should consider a radical departure along the lines of the RISC paradigm shift in computer architecture back in the eighties. The key to understanding such complex artefacts as computers (i.e., the hardware) or aircrafts and making their construction and deployment manageable lies in three principles: 1) these artefacts are highly componentized, 2) the components have limited and relatively simple, i.e., "RISC-style", functionality so that the behavior of a component is predictable, including its timing behavior (i.e., performance properties), and 3) the interfaces of the components are designed as narrow as possible so that the interaction complexity between components is kept to a minimum. These principles would, for example, suggest replacing the ubiquitous SQL language by a suite of much simpler APIs. To a limited extent, research and industry is pursuing this approach already, e.g., in the form of light-weight or specialized data management engines such as SleepyCat (aka Berkeley DB) or TimesTen, but this is not exactly the mainstream in the database software industry. Of course, components in this system philosophy would be relatively large-grained rather than fine-grained objects; for example, a storage manager (including concurrency control and recovery), a select-project-join engine, or a text document manager would be components, but a universal database system would be against the spirit of this approach. Also, all components should apply Occam's razor to their interfaces as well as their internals, and should avoid features that are rarely used or implementation techniques that improve performance only marginally or only under special circumstances (e.g., no hash structures as B-tree indexes are good enough in most situations).

In the context of self-tuning E-services, the key point of components is their *predictability* that falls out from their (relative) simplicity. The simpler an interface and the underlying internals are, the fewer tuning knobs need to be exported and the easier it becomes to mathematically analyze and predict the component's performance. For example, query result size and runtime estimations for a select-project-join engine becomes more tractable if the complexity of the search predicates is limited (e.g., disallows user-defined functions or arbitrary multidimensional predicates, which are rarely needed in E-business applications). For OLAP or GIS applications, on the other hand, we could build a specialized multidimensional query engine and construct a separate performance prediction model for this engine, which should again be much more tractable than for a universal database system. The (mathematical) performance model for a component should be developed *together* with the component software, which is much easier than doing performance modeling in retrospect. Such a performance model would be parameterized, as the component's performance depends on the underlying hardware configuration and workload properties. When the component is bound to a specific hardware and application environment (with a known workload profile), the performance model would become a performance (or service quality) *contract* by which certain guarantees are given about metrics of interest (e.g., the 95th percentile of the response time distribution for a specific class of user requests).

Of course, the highly componentized approach to data management and E-service software is viable only if multiple components can be composed into value-added services without re-introducing a poor gain/pain ratio. Again, the RISC-style simplicity of component interfaces and the cross-component interactions would be the key asset for low-effort *composability*. This argument includes the performance contracts: the value-added service should in turn provide a performance model and contract that are derived from the models of the underlying components. Of course, this line of thinking about system architecture bears the risk that value-added services exhibit very high overhead for component interaction (e.g., for copying data

across interfaces). However, this would be only constant overhead, say a slow-down factor of 2. We can make up for this performance loss by proportionally adding (inexpensive) hardware resources; this is a case where the KIWI method makes perfect sense. The true gain would lie in the fact that such systems and E-services composed from RISC-style components would have predictable performance, so that automatic tuning could finally move from wishful thinking to viable engineering.

5.3 Why Does RISC Simplify Automatic Tuning?

A major incentive for moving towards RISC-style components is to enable auto-tuning of information services. Tuning must consider the relationship between workload characteristics, knob settings, and the resulting performance in a quantitative manner. Therefore, mathematical models (in combination with online feedback control) are crucial. Unfortunately, these models work only in a limited context, i.e., when focusing on a particular knob (or a small set of inter-related knobs). Attempting to cover the full spectrum of tuning issues with a single, comprehensive model is bound to fail because of the lack of sufficiently accurate mathematical models or the intractability of advanced models. This is why limiting ourselves to using only RISC components is so important: we no longer need to aim for the most comprehensive, elusive performance model, and there is hope that we can get a handle on how to auto-tune an individual service. It is much easier to tune a system with a less diverse workload and less dynamic resource sharing among different data and workload classes. Of course, the global tuning problem is now pushed one level above: how do we tune the interplay of several RISC components? Fortunately, a hierarchical approach to system tuning appears to be more in reach than trying to solve the entire complex problem in one shot. In the following we outline the main steps of such a hierarchical auto-tuning framework:

- *Identify the need for tuning:* Each RISC data management component must include a self-inspection module. This module should be in charge of monitoring long-term workload and performance metrics, comparing their values to some target settings for relevant metrics such as average response time for a given pair of user-request class and user category or the 90th percentile of the response time distribution. If one of these application-level metrics degrades significantly, this indicates the need for tuning and should turn on a ”DBA attention” red light or, much better, trigger an auto-tuning procedure. The rationale for this self-inspection and alerting approach is that information systems typically run reasonably well when they are initially deployed and start suffering performance problems as workloads undergo long-term evolution (e.g., when a Web site becomes more popular). It is crucial to react quickly, hence automatically, to the performance degradation, as poor performance is often perceived as equivalent to service unavailability by users (e.g., because of many timeouts in Web connections).
- *Identify the bottleneck:* When performance is unsatisfactory, a bottleneck analysis is mandatory. In the hierarchical world of components and higher-level services, a first approximation would be to identify the most performance-critical building block (i.e., RISC data server or higher-level application server). This can be done quite easily by hypothetically adding infinite resources to each server, one at a time, and computing (using mathematical models or fast simulation) whether this would improve the currently unacceptable application-level response times.
- *Analyze the bottleneck:* Once the bottleneck component is determined, a more detailed analysis is necessary to understand which tuning knobs within the component might help alleviating the performance problem. This analysis should again proceed in a top-down manner by estimating the performance of the workload’s most dominant transactions/queries/requests on a *hypothetically* modified configuration. So an auto-tuning wizard would consider what if the server had more memory, more disks, a faster CPU or more processors, no lock contention, or simply a higher multiprogramming level, to name the most important possible bottlenecks. This assessment again needs mathematical models (or very fast online simulators), e.g., an analytic model for the disk system. However, by considering one potential bottleneck at a time, each of these models can focus on an *individual* aspect, and such limited-scope models are, to a large extent, available.
- *Estimate performance changes:* Once the primary bottleneck is known, the analysis procedure should be refined and remedies should be considered. For example, if the bottleneck were the disk system, we should assess possible remedies like adding a disk, increasing the cache size in memory, creating an index, rearranging data on one disk (e.g., to reduce seek times), rearranging data across disks (to reduce load imbalances), etc. Finally, the wizard should then determine which one is the best in terms of cost, risk of making something else in the operational system unstable, and possibly further aspects. The result could be alerting a responsible person to spend a few hundred dollars on extra memory or disks, or it could be an internal measure such as creating an index or rearranging data across disks. In either case, a few hours later the system would ideally be in good shape again. Note that for selection of which tuning measure should be applied, the wizard needs to estimate the anticipated performance improvement. This is needed because some of the possible tuning options are parameterized, and we need to determine an appropriate parameter value to make sure the performance improvement is significant but to avoid spending too much money. For example, when the wizard decides to increase the cache size, we need to know how much

additional memory we should purchase. Again, this step crucially relies on mathematical models or fast but accurate simulation (e.g., to estimate the cache hit ratio for a hypothetical cache size).

- *Adjust tuning knob:* The final step is mere mechanics. Once the decision about the best tuning option is made, the corresponding knob has to be adjusted. As mentioned before, this may require interaction with a human person to add hardware resources. For availability, it is highly desirable that the adjustment takes place online without having to shut down the operational system. This is feasible for most kinds of tuning steps these days (including adding disks and rearranging data across disks).

The hierarchical nature of the outlined auto-tuning procedure is in line with good practice for manual/intellectual tuning [SB02]. In particular, our approach also adopts a "think globally, fix locally" regime. Mathematical models have been and remain to be key assets also in the practical system tuning community [MA98]. The key to making the mathematics sufficiently simple and thus practical lies in the reduced complexity of the component systems and their interfaces and interplay. We believe that there is a virtue in engineering system components such that their real behavior can be better captured by existing mathematical modeling techniques, even if this may lead to some (tolerable) loss of high-end features and efficiency. The benefit of this *engineering-for-predictability* paradigm lies in the improved manageability of systems.

6. Conclusion

When we started the COMFORT project in 1990, we had the hope that automatic tuning can be achieved with a few simple principles. This was clearly wishful thinking. While the feedback control loop framework provides useful guidance, the difficult problems are in the details of the various tuning issues. For robust solutions workload statistics and mathematical models are key assets, and for viable engineering these must be carefully designed so as to ensure acceptable overhead. Our field in general has made significant progress towards self-tuning database technology, but there is no breakthrough.

The biggest challenges that our research community should address as high-priority problems are the interactions of different system components and their tuning knobs and the interference between different workload classes. For tackling this complexity we believe that a drastic simplification of today's overly complex system architectures is overdue. Assuming that we are able to build individually self-tuning components, the composition of these building blocks into higher-level E-services with service-quality guarantees seems feasible only with sufficiently simple component interfaces and radical minimization of cross-talk.

On the technical side a fundamental issue to investigate is how to cope with multi-class workloads, at both the database and the middleware level, such that we can provide differentiated quality of service, say guarantees about the 95th percentile of the response time distribution, on a per class basis. For example, in seemingly simple home banking, we can observe different request classes such as account lookups, brokerage orders, or sophisticated portfolio analyses, different customer classes such as premium customers, regular customers, and guests (i.e., potential customers who might be offered hypothetical portfolios to play with), and different connection types such as cell phones or other gizmos, high-speed Internet access from a PC, or local access from a call center. Combining all these options yields a large number of different workload classes with specific performance goals. Orthogonal to this complexity is the need to capture workload behavior in a more context-sensitive manner. Today, tuning is mostly driven by stationary frequencies of stateless request types, but the contextual patterns that are exhibited in query execution plans or entire user sessions do matter, and likewise long-term periodicity of certain load patterns (e.g., Monday morning peaks) needs to be considered as well.

Given the paramount importance of the manageability of modern information services, our community should intensify its efforts both on the technical issues posed by individual aspects and the strategic dimension of system architectures that are more amenable to automatic tuning.

References

[ACN00] S. Agrawal, S. Chaudhuri, V.R. Narasayya: Automated Selection of Materialized Views and Indexes in SQL Databases, VLDB Conf. 2000.

[Al00] G. Alvarez, K. Keeton, A. Merchant, E. Riedel, J. Wilkes: Storage Systems Management, Tutorial, SIGMETRICS Conf. 2000.

[Be98] P. Bernstein et al.: The Asilomar Report on Database Research, ACM SIGMOD 27(4), December 1998.

[BBK00] N. Bhatti, A. Bouch, A. Kuchinsky: Integrating User-Perceived Quality into Web Server Design, WWW Conf. 2000.

[Br94] K.P. Brown, M. Mehta, M.J. Carey, M. Livny: Towards Automated Performance Tuning for Complex Workloads, VLDB Conf. 1994.

[BCL96] K.P. Brown, M.J. Carey, M. Livny: Goal Oriented Buffer Management Revisited, SIGMOD 1996.

[CKL90] M.J. Carey, S. Krishnamurthi, M. Livny: Load Control for Locking : the Half-and-Half Approach, PODS 1990.

[CN01] S. Chaudhuri, V. Narasayya : Automating Statistics Management for Query Optimizers, IEEE Transactions on Knowledge and Data Engineering 13(1), 2001.

[CW00] S. Chaudhuri, G. Weikum: Rethinking Database System Architecture - Towards a Self-tuning RISC-style Database System, VLDB Conf. 2000.

[Co88] G.P. Copeland, W. Alexander, E.E. Boughter, T.W. Keller: Data Placement in Bubba, SIGMOD 1988.

[DE96] IEEE CS Data Engineering Bulletin 19(2), Special Issue on Online Reorganization, June 1996.
[DE99] IEEE CS Data Engineering Bulletin 22(2), Special Issue on Self-Tuning Databases and Application Tuning, June 1999.
[DE01] IEEE CS Data Engineering Bulletin 24(1), Special Issue on Infrastructure for Advanced E-Services, March 2001.
[FRT91] P.A.. Franaszek, J.T. Robinson, A. Thomasian: Wait Depth Limited Concurrency Control, ICDE 1991.
[Gi00] M. Gillmann, J. Weissenfels, G. Weikum, A. Kraiss: Performance and Availability Assessment for the Configuration of Distributed Workflow Management Systems, EDBT 2000.
[Gi02] M. Gillmann, G. Weikum, W. Wonner: Workflow Management with Service Quality Guarantees, SIGMOD 2002.
[GG97] J. Gray, G. Graefe: The Five-Minutes Rule Ten Years Later, and Other Computer Storage Rules of Thumb, ACM SIGMOD Record 26(4), December 1997.
[GS00] J. Gray, P.J. Shenoy: Rules of Thumb in Data Engineering, ICDE 2000.
[HLR00] G. Haring, C. Lindemann, M. Reiser: Performance Evaluation: Origins and Directions, Springer, 2000.
[HW91] H.-U. Heiss, R. Wagner: Adaptive Load Control in Transaction Processing Systems, VLDB Conf. 1991.
[He00] J.M. Hellerstein, M.J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, M.A. Shah: Adaptive Query Processing: Technology in Evolution, IEEE CS Data Eng. Bulletin 23(2), 2000.

[IBM01] Autonomic Computing: IBM's Perspective on the State of Information Technology, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
[Iso01] Isolation Testing Project, University of Massachusetts at Boston, http://www.cs.umb.edu/~isotest/
[Ja01] H.V. Jagadish, H. Jin, B.C. Ooi, K.-L. Tan: Global Optimization of Histograms, SIGMOD 2001.
[KW02] A.C. Koenig, G. Weikum: A Framework for the Physical Design Problem for Data Synopses, EDBT 2002.
[KFD00] D. Kossmann, M.J. Franklin, G. Drasch: Cache Investment: Integrating Query Optimization and Distributed Data Placement, TODS 25(4), 2000.
[La01] T. Lahiri, A. Ganesh, R. Weiss, A. Joshi: Fast-Start: Quick Fault Recovery in Oracle, SIGMOD 2001.
[LG98] P.-A. Larson, G. Graefe: Memory Management During Run Generation in External Sorting, SIGMOD 1998.
[Lee00] M.-L. Lee, M. Kitsuregawa, B.C. Ooi, K.-L. Tan, A. Mondal: Towards Self-tuning Data Placement in Parallel Database Systems, SIGMOD 2000.
[LGS00] C. Loosley, R.L. Gimarc, A.C. Spellmann: E-Commerce Response Time: a Reference Model, White Paper, Keynote Systems Inc., 2000.
[MA98] D.A. Menasce, V.A.F. Almeida: Capacity Planning for Web Performance - Metrics, Models & Methods, Prentice Hall, 1998
[MW91] A. Moenkeberg, G. Weikum: Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing, ICDE 1991.
[MW92] A. Moenkeberg, G. Weikum: Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing, VLDB Conf., 1992.
[Mu99] R. Munz: Usage Scenarios of DBMS, Keynote, VLDB Conf. 1999.
[NMW99] G. Nerjes, P. Muth, G. Weikum: A Performance Model of Mixed-Workload Multimedia Information Servers, 10th German Conf. on Performance Assessment of Computer and Communication Systems, 1999.
[OOW93] P.E. O'Neil, E.J. O'Neil, G. Weikum: The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD 1993.
[Ora00] Oracle8i with Oracle Fail Safe 3.0, White Paper, Oracle Corp., 2000.
[PB01] D. Patterson, A. Brown: Recovery-Oriented Computing, Keynote, HPTS Workshop 2001.
[PCL93] H. Pang, M.J. Carey, M. Livny: Partially Preemptive Hash Joins, SIGMOD 1993.
[Ra02] J. Rao, C. Zhang, G.M. Lohman, N. Megiddo: Automating Physical Database Design in a Parallel Database, SIGMOD 2002.
[SWZ93] P. Scheuermann, G. Weikum, P. Zabback: Adaptive Load Balancing in Disk Arrays, 4th FODO Conference, 1993.
[SWZ98] P. Scheuermann, G. Weikum, P. Zabback: Data Partitioning and Load Balancing in Parallel Disk Systems, VLDB Journal 7(3), 1998.
[SB02] D. Shasha, P. Bonnet: Database Tuning: Principles, Experiments, and Troubleshooting Techniques, Morgan Kaufmann, 2002.
[St01] M. Stillger, G.M. Lohman, V. Markl, M. Kandil: LEO: DB2's LEarning Optimizer, VLDB Conf. 2001.
[TGS85] Y.C. Tay, N. Goodman, R. Suri: Locking Performance in Centralized Databases, TODS 10(4), 1985.
[TS97] D. Theodoratos, T.K. Sellis: Data Warehouse Configuration, VLDB Conf. 1997.
[Tho93] A. Thomasian: Two-Phase Locking Performance and its Thrashing Behavior, TODS 18(4), 1993.
[Vi99] R. Vingralek, Y. Breitbart, M. Sayal, P. Scheuermann: Web++: A System for Fast and Reliable Web Service, USENIX Conf. 1999.
[We94] G. Weikum, C. Hasse, A. Moenkeberg, P. Zabback: The COMFORT Automatic Tuning Project, Information Systems 19(5), 1994.
[We99] G. Weikum, A.C. Koenig, A. Kraiss, M. Sinnwell: Towards Self-tuning Memory Management for Data Servers, IEEE CS Data Engineering Bulletin 22(2), 1999.
[WZS91] G. Weikum, P. Zabback, P. Scheuermann: Dynamic File Allocation in Disk Arrays, SIGMOD 1991.