Bulletin of the Technical Committee on

Data Engineering

June 1999 Vol. 22 No. 2 **IEEE Computer Society**

Letters

Special Issue on Self-Tuning Databases and Application Tuning

Conference and Journal Notices

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

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

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

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

The web page for the Data Engineering Bulletin is http://www.research.microsoft.com/research/db/debull. The web page for the TC on Data Engineering is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

ICDE'2000

As many of you perhaps are aware, the deadline for the ICDE'2000 conference, which is the flagship conference of our technical committee, was June 16. I am co-PC chair along with Gerhard Weikum. We had almost 300 submissions, so this promises to be a very selective and high quality conference. Our PC members are now just beginning the reviewing of the submissions, with authors notified of acceptance by October first. ICDE'2000 will be held in San Diego, California from Feb. 28 to March 3. San Diego is a great place (even more so in the winter when the weather is tough in many other places) and the conference will have an excellent program. While we will not know the complete technical program for a while, we do know that there will be two great keynote speakers,

Jim Gray: Senior Researcher in Microsoft Research, and 1999 ACM Turing Award winner;

Dennis Tsichritzis: Chairman of the Executive Board of GMD German National Research Center for Information Technology.

Both are terrific speakers with great insights and much useful information to convey. I would encourage you to plan to attend.

This Issue

Database performance has long been a topic of great practical importance. This is attested to by our industry history of benchmark wars. Initially, it was the transaction processing benchmarks, TPC-A and TPC-B (variants of the debit-credit benchmark). Subsequently, transaction processing benchmarking has moved to TPC-C (the order-entry benchmark). To gain insights into database performance on decision support applications, the TPC-D benchmark was introduced (recently transformed into two variants, TPC-H and TPC-R). In all the benchmarking, database systems are meticulously hand-tuned for optimal performance, frequently by system developers who know the systems inside out.

But times are changing. Good performance continues as an industry goal. But many users cannot afford the level of technical expertise that is required to achieve vendor benchmark performance levels. The small pool of experts, and their large expense, hence is substantial barriers to ordinary customers' ability to achieve satisfactory performance. This is where pre-tuning, self-tuning, and tuning tools enter the picture. There has been a substantial increase in interest in this area over the past few years. And this interest spans both academic and commercial worlds.

The current issue, ably edited by Surajit Chaudhuri, provides a good cross-section of the exciting and commercially important work now going on in this kind of performance tuning. The issue contains articles from a number of leading vendors as well as leading researchers. The result is that the issue provides a very good sense of the status of this field. I want to thank Surajit for his fine job and hard editorial work in organizing this issue.

David Lomet
Microsoft Research

Letter from the Special Issue Editor

Databases are not yet ready-to-go ”shrink-wrapped” software. Today’s database systems require application developers to tune a vast array of controls for optimal performance. Unfortunately, only a few of the database administrators and developers can fully fathom the impact and realize the benefits of such controls. More often than not, an application developer is overwhelmed by the complexity of the ”control panel” a database system presents. Therefore, an important technical challenge is to simplify the task of tuning a database system to reduce the total cost of ownership of databases, without compromising efficiency. The importance of this challenge is bigger than ever before, since today’s database systems are an integral component of a *diverse* set of applications. Thus, databases should be *self-tuning* and adjust themselves to the characteristics of each application. This will surely be an important step in making database systems ubiquitous.

This issue of the bulletin showcases the state-of-the art in the area of self-tuning databases. We have complemented the discussion of self-tuning technology with case studies of *tuning important applications* to illustrate the challenges in tuning today’s database systems. The first article, by Gerhard et al., introduces a framework for self-tuning memory management in relational databases. The article by Schiefer and Valentin highlights how a DB2 Universal Database server can be configured to suit application needs. The article by my colleagues and myself presents the self-tuning technology that has been incorporated in the Microsoft SQL Server product. In the next article, Subramanian and Krishnamurthy outline the challenges in optimizing the performance of object-relational database systems. The last two papers serve as case studies of tuning database applications. The article by Kemper, Kossman, and Zeller shows how the performance of a SAP R/3 application may be tuned against a database. Finally, the article by Shasha discusses how time series queries need to be tuned on relational databases.

I hope the articles in this issue will inspire you to study the challenges in building self-tuning and ready-to-go database systems. Incidentally, this is now time to say goodbye to you as an associate editor of the bulletin. This has been a rewarding experience for me. I sincerely hope that you have found the issues that I edited as interesting as I do.

Surajit Chaudhuri
Microsoft Research

Towards Self-Tuning Memory Management for Data Servers

Gerhard Weikum, Arnd Christian König
University of the Saarland, D-66041 Saarbrucken, Germany
{weikum,koenig}@cs.uni-sb.de

Achim Kraiss
Dresdner Bank AG, D-60301 Frankfurt, Germany
achim.kraiss@dresdner-bank.com

Markus Sinnwell
SAP AG, D-69185 Walldorf, Germany
markus.sinnwell@sap-ag.de

Abstract

Although today's computers provide huge amounts of main memory, the ever-increasing load of large data servers, imposed by resource-intensive decision-support queries and accesses to multimedia and other complex data, often leads to memory contention and may result in severe performance degradation. Therefore, careful tuning of memory mangement is crucial for heavy-load data servers. This paper gives an overview of self-tuning methods for a spectrum of memory management issues, ranging from traditional caching to exploiting distributed memory in a server cluster and speculative prefetching in a Web-based system. The common, fundamental elements in these methods include on-line load tracking, near-future access prediction based on stochastic models and the available on-line statistics, and dynamic and automatic adjustment of control parameters in a feedback loop.

1 The Need for Memory Tuning

Although memory is relatively inexpensive and modern computer systems are amply equipped with it, memory contention on heavily loaded data servers is a common cause of performance problems. The reasons are threefold:

- Servers are operating with a multitude of complex software, ranging from the operating system to database systems, object request brokers, and application services. Much of this software has been written so as to quickly penetrate the market rather than optimizing memory usage and other resource consumption.
- The distinctive characteristic and key problem of a data server is that it operates in multi-user mode, serving many clients concurrently or in parallel. Therefore, a server needs to divide up its resources among the simultaneously active threads for executing queries, transactions, stored procedures, Web applications, etc. Often, multiple data-intensive decision-support queries compete for memory.
- The data volumes that need to be managed by a server seem to be growing without limits. One part of this trend is that multimedia data types such as images, speech, or video have become more popular and are being merged into conventional-data applications (e.g., images or videos for insurance claims). The other

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

part is that diminishing storage costs have made it possible and business competition has made it desirable to keep extended histories of business events for marketing and other enterprise-critical decisions.

Like resource contention in general, memory contention can lead to performance disasters, also known as "thrashing". When memory becomes scarce in a multi-user server, performance does not degrade gracefully at all. For example, when too many join or grouping operations run in parallel and not all of them can be given sufficiently large in-memory hash tables or sort buffers, too much intermediate data needs to be written onto disk and read again too often. In this case, the memory bottleneck causes a dramatic increase of the disk load, leading to long queueing delays for disk accesses (or network queueing in other settings), and ultimately, such excessive queueing kills the application performance. The possible remedy of limiting the multiprogramming level (i.e., the maximum number of concurrently active operations) is not a real solution as this forces clients to wait until their requests are admitted for execution and may thus lead to poor user-perceived response times, too.

The critical nature of memory contention makes careful tuning of the server's memory management a necessity. As today's IT operation costs are increasingly dominated by the costs of human administration staff rather than hardware or software, automating tuning decisions will become mandatory in the long run. In particular, we need dynamically self-tuning methods that can adjust the policies and control parameters of a server's memory management to evolving workload characteristics in an online manner without human intervention.

This paper gives an overview on the state of the art of self-tuning memory management methods for data servers, as a critical piece along the path towards a new generation of zero-admin, trouble-free information systems [1]. The rest of the paper is organized as follows. Section 2 reviews approaches to traditional data caching, the aim being to maximize the cache hit ratio of a server. Section 3 extends the workload setting by adding long-running queries that require large amounts of private working memory. Section 4 extends the architectural model by considering how to exploit the distributed memory in a server cluster. Section 5 adds self-tuning prefetching techniques to the repertoire of memory management methods. Finally, Section 6 discusses how to exploit advanced forms of both prefetching and caching in a widely distributed system like the Internet. Section 7 concludes the paper with some general lessons learned.

2 Self-tuning Server Caching

The traditional usage of memory in a data server is for caching frequently accessed data to avoid disk I/O. Such a cache is typically organized to hold fixed-size pages (e.g., of size 32 KBytes) and is managed by a cache replacement policy that drops a page whenever a new page needs to be brought in. The goal of the cache manager is to maximize the *cache hit ratio*, i.e, the fraction of page accesses that can be served from the cache. The most widely used replacement policies are variations of the LRU (Least Recently Used) algorithm which selects the page whose last reference time is oldest among all currently cache-resident pages as the replacement victim.

LRU is a robust policy that performs well under a wide variety of workload characteristics. However, it is suboptimal with regard to specifically idiosyncratic, albeit fairly frequent page reference patterns:

- Sequential scans over a large set of pages which compete for memory with random accesses to individual pages. Plain LRU would rank pages that are freshly touched during a scan as worthy, although the probability for accessing them again in the near future is low.
- Random accesses to different page sets with highly skewed cardinalities. For example, LRU treats data and index pages as equally worthy when both exhibit the same access pattern, say for a batch of exact-match queries where index and data accesses alternate, although the access probability for an index page is much higher.

To overcome these deficiencies of plain LRU, researchers and system architects have developed, during the past decades, a number of tuning methods for managing a page cache. It turns out, however, that most of them are troublesome in that they are not fully self-tuning: either they rely on human intuition or experience for calibration, or they are geared only for relatively static workloads where access patterns remain stable over a long time period. We can categorize the various approaches as follows:

- The "Pandora" approach relies on explicit tuning hints from programs, either application programs of file servers or the query-processing layer of a database server. For example, a query-processing engine knows about sequential access patterns and can inform the cache manager so that pages referenced only once during a scan are attached to the low-priority end of the LRU chain [22]. However, this hint-passing approach is limited; trying to generalize it to other access patterns bears a high risk that "local" hints issued by individual operations are detrimental for global performance metrics (e.g., by monopolizing memory on behalf of a single operation). So hints open up both potential blessing and curse, like Pandora's box.
- The "Sisyphus" approach aims to tune the cache manager by partitioning the overall cache into separate "pools" and assigns different page classes to different pools [22]. This approach works very well for partitioning index versus data pages or discriminating pages from different tables. However, determining appropriate pool sizes and a proper assignment of page classes to pools is a highly delicate tuning task, and when the workload characteristics change significantly after some time, one may have to start the tuning all over again. So this is a real Sisyphus work for the administration staff.
- Finally, the "Sphinx" approach takes a completely different approach in that it abandons LRU and rather adopts a replacement policy based on access frequencies. Such an LFU (Least Frequently Used) policy would indeed be optimal for a static workload where pages have independent reference probabilities that do not vary over time. With evolving access patterns, however, one crucially needs some kind of aging scheme for reference counters to recognize when previously hot pages become cold and vice versa. For example, in practically used variants such as LRD (Least Reference Density) replacement [7], the reference counter of a page is decremented by a specific value once every so many references; so the aging is controlled by two parameters. Now the crux is to find appropriate values for such fine-tuning parameters, which may be as hard as it was for Oedipus to solve the Sphinx's cryptic riddles.

The state-of-the-art solution for truly self-tuning cache management follows the Sphinx approach, but avoids the delicate aging parameters by using an appropriate form of online statistics which has the aging automatically built-in. We coin this approach the "Nike" approach (to stay in Greek mythology); concrete implementations are the LRU-k algorithm [15] or its variant 2Q [11]. The LRU-k method is organized into three steps according to an *observe-predict-react* paradigm [24]:

- *Observation:* The method is based on keeping a limited form of each relevant page's reference history: the k last reference timepoints. Here "relevant" means all pages that are currently in the cache plus some more pages that are potential caching candidates. Determining the latter kind of relevant pages may be seen as a tricky fine-tuning problem again, but one can easily make a pragmatic and practically robust choice based on the five-minute rule [10]: the history of pages that have not been referenced within the last five minutes can be safely discarded from the method's bookkeeping.
- *Prediction:* The time interval between the k-th last reference of a page p and the current time gives us a statistical estimator for the reference probability of the page: having observed k, $k \geq 2$, references to p in the time period between t_k and now and assuming independent "arrivals" of page accessses, we can infer a page-specific access rate, also known as the page's *heat*, $heat(p) = \frac{k}{now - t_k}$, and the probability for accessing the page (at least once) within the next T time units is given by $1 - e^{-heat(p)T}$. It can be shown that it is optimal to rank pages according to these near-future access probabilities [16].
- *Reaction:* When space needs to be freed up in the cache, LRU-k replaces the page with the smallest value for the above (estimated) probability. For low-overhead implementation, it is sufficient to simply maintain the heat of the pages or merely the timepoints of the k-th last references, and determine the page with

the lowest heat or, equivalently, the oldest k-th last reference as a replacement victim. Such an implementation needs a priority queue for the relevant pages; so it requires logarithmic time for page-replacement decisions. An approximative variant, the 2Q algorithm, makes its decisions even in constant time [11].

This family of cache replacement algorithms can be easily generalized to cope also with variable-size caching granules, say documents rather than pages, as one would have to deal with, for example, in a Web server. Now the benefit of caching a document, as measured by the document heat, needs to be normalized by the occupied cache space. So we derive a benefit/cost ratio for each document as the quotient of document heat and document size, which has been coined the *temperature* of the document. Then, for caching decisions, documents are simply ranked by their temperature.

LRU-k and its variants cope well with a wide spectrum of page access patterns including mixes of sequential and random references and also patterns that evolve over time. They are completely self-tuning in that they neither need administrator help for fine-tuning nor hints from programs. Their core assets, *online statistics for tracking access patterns* and *probabilistic reasoning for predicting near-future accesses*, are indeed fundamental for memory management in general, as the discussion of the following sections will show.

3 Automatic Tuning of Cache and Working Memory

In addition to caching disk-resident data in memory, a data server needs to manage also working memory that is needed by long-running operations such as hash- or sort-based joins or video/audio streams. With a highly diverse workload, memory management should no longer focus on a single global performance metric such as the mean response time of all operations. Rather differences in the resouce consumption and response-time-criticality of different *workload classes* should be taken into account. As the system has no way of automatically inferring the relative importance of the various classes, a human administrator must explicitly specify class-specific performance goals, which can then be monitored and enforced automatically. For example, all medium-sized join operations of standard users could be one class, large joins issued on behalf of business executives could be another class, and the most common class would be "rifle-shot" exact-match queries; the first two classes need working memory and the last one operates on the server's shared page cache. Performance goals could then be upper bounds for the per-class mean response time or the 90th percentile of the class's response time distribution for a given throughput (i.e., class-specific arrival rates) to be sustained. The number of such classes would typically be small, maybe, five or ten, and it often makes sense to have an additional "free" class with all operations for which no goal is specified. The objective function of the server should then be to minimize the mean response time of the free class under the constraint that the response-time goals of all classes are met.

The mechanism for handling multiple workload classes is a conceptual partitioning of the available memory into *class-specific memory areas*; these include operation-type-specific working memory (e.g., hash tables) as well as a shared page cache or even multiple page caches dedicated to specific data partitions. The partitioning is merely conceptual, not physical; so a memory area can be shared by multiple workload classes. For ease of explanation, however, we assume in what follows that each class accesses only one memory area; for example, m-1 classes of decision-support queries operate on private working memory, and one class of exact-match queries makes use of the shared page cache. Further note that we still need a replacement algorithm like LRU-k within an area if the area serves as a page cache (as opposed to working memory). Approaches for automatically maintaining performance goals within this memory model can generally be described as a *feedback loop*, again consisting of three steps for observation, prediction, and reaction [4, 3, 20].

Observation: This step involves collecting online statistics about the response time of the various workload classes, e.g., using moving time-window averaging. The length of the observation window must be carefully chosen, for too small intervals may result in the system overreacting to statistical fluctuations, whereas with overly long intervals, the system is not sufficiently responsive to evolving changes in the workload characteristics. Therefore, the observation phase needs some form of calibration, but this can be accomplished automatically based on the ratio of the observed metric's mean value and variance.

Prediction: Whenever the performance goal of one class of operations is not satisfied, the server has to determine an appropriate re-adjustment of its memory usage, for example, increasing the working memory for a certain class of join operations and dynamically decreasing the size of the shared page cache. Assuming algorithms for the various operations (e.g., joins or sorting) that can cope with time-varying memory resources, this calls for predictions of how performance would change if the memory assignment were modified in a certain way. So we are interested in predicting the response time R_i of class i as a function of the sizes of different memory areas $M_1, \ldots M_m$ assigned to the various classes.

Unfortunately, deriving or even adequately approximating the function $R_i(M_1, \ldots, M_m)$ is very difficult. Prior approaches that aimed at a completely analytical, accurate solution have not had much practical impact. Rather a simpler and practically viable approach [4, 3, 20] is to model each R_i function approximatively as a linear function of its memory-area size M_i. Although this is a fairly crude approximation as far as absolute hit-ratio or response-time predictions are concerned, it is sufficient to drive the feedback loop for gradually approaching the optimum memory assignment. Based on this approximate model one can determine the minimum amount of memory M_i such that the predicted performance for class i will be acceptable.

Reaction: After the desirable sizes of memory areas have been determined, the appropriate adjustments have to be enforced by the server, by increasing the area of the class whose response-time goal has been violated and taking away memory from the "free" class. If no such excess memory can be reclaimed without violating another class's goal, the server is obviously in an overload situation. In this case, it would re-initiate the prediction phase under the new goal of finding a memory assignment that mimimizes the maximum goal violation among all classes, $max\{\frac{R_i}{G_i} \mid 1 \leq i \leq m\}$, where R_i is the response time and G_i the response-time goal of class i. In addition, one could introduce an admission control (i.e., a limitation of the multiprogramming level) for preventing such situations or making them unlikely; methods along these lines again build crucially on a prediction step similar to what we have sketched above (see, e.g., [8, 3]).

If it turns out, after another observation cycle, that the re-adjusted memory area for the goal-violating class is still not sufficiently large for achieving the goal, the entire feedback loop is iterated. This feedback loop is guaranteed to converge (quickly) to a stable state if the actual response-time functions (i.e., not the approximative linear ones) are concave [3]. Even for non-concave functions, the approach typically reaches a new stable state fairly quickly, as new feedback is incorporated in each iteration of the feedback loop.

4 Exploiting Distributed Memory

High-end data services are often implemented on server clusters with a (small) number of computers, each with its own memory, sharing disks through a high-speed network (e.g., fibre channel). A functionally equivalent architecture would be a collection of independent servers (each with its own disks) with all data replicated across all of them; this is attractive for read-only applications such as Web information services. In both cases, a data request arrives at one of the servers and will primarily be executed there. During the execution it is possible that the server has to fetch some data from disk although another server holds that data in its memory. This is the situation where data shipping can be beneficial: with current low-latency interconnect technology accessing the remote memory of another server is often significantly faster than performing the disk access.

Exploiting this option in a systematic manner amounts to a distributed caching algorithm, which essentially controls the dynamic replication of data objects (i.e., fixed-size pages or variable-size documents) in the caches of the underlying computers [6, 9, 14]. Two simple heuristics for this purpose are "egoistic" caching, where each server simply runs a local cache replacement algorithm such as LRU or LRU-k, and the other extreme, called "altruistic" caching. Egoistic caching views remotely cached data that is not locally cached as an opportunity, but does not actively aim to make this situation more likely. In fact, it may easily end up with situations where the hottest data is fully replicated in all caches with only little space left for the rest of the data. Altruistic caching, on the other hand, aims exactly at minimizing this replication by giving preference in the local cache replacement to data that is not already cache-resident at a different server. Obviously, this method needs some distributed bookkeeping about the global cache contents. With a high-bandwidth network this is an affordable overhead, and altruistic caching outperforms the egoistic heuristics. However, even the fastest interconnect may become congested under high load and then turns into an effectively-low-bandwidth network. So one needs some careful tuning for reconciling the two distributed caching approaches.

The key to such an online tuning is, once again, a mathematical cost model for assessing whether egoistic or altruistic behavior is preferable under the current workload and system setting. Such a model has been developed in [21] (see also [23] for a closely related approach), driven by online statistics about the local and global heat of data objects, and taking into account memory-access costs and detailed costs of both the network and the computers' disks including queueing effects. Then cache replacement decisions are essentially optimization problems: drop the page or document that contributes least to the system-wide caching benefit where benefit is proportional to the mean response time of data requests over all servers. It is crucial, however, that this optimization model can be evaluated online with low overhead, so that it can drive the online decisions of the local cache managers. To this end, the cost model is simplified and itself "tuned" in [21] by various approximations and lazy, threshold-driven evaluations of cost formulas, as well as lazy estimation of load parameters and dissemination of cache-contents bookkeeping data in the network.

The resulting algorithm always performs at least as good as the better one of the two simple heuristics and clearly outperforms the other one. Most importantly, this property holds even for evolving workloads and time-varying network utilization. As the underlying cost model includes disk queueing, the entire approach can even contribute to disk load balancing, for example, by caching a higher fraction of an overloaded disk's data in remote caches. As a further extension of this method and a generalization of the algorithm of Section 3, it is even possible to accomodate workload-class-specific response-time goals in a server-cluster architecture [20].

5 Integrating Speculative Prefetching with Caching

Caching reduces the overall disk I/O load and thus helps throughput, but since caching can usually not eliminate disk I/O completely, data-intensive operations may still exhibit long response times. To reduce response time, prefetching can be employed as an additional measure to mask the latency of the disk accesses by bringing the relevant data into memory already before it is explicitly requested. Prefetching pays off particularly well for high-latency data requests, for example, for large images or video fragments that reside on disk, or when the data resides in a tertiary-storage archive or needs to be fetched from another server over a high-latency WAN.

Prefetching generally leverages some anticipation of near-future data requests. In special cases such as sequential scans, exact knowledge about what and when to prefetch can be obtained. In full generality, however, a server would have to speculate about the near-future access patterns of ongoing operations or client sessions. Such speculation is necessarily stochastic in nature, so that prefetching is beneficial only with a certain (hopefully high) probability. The server has to be careful so as not to initiate too much speculative prefetching, as this could possibly replace other, more worthy data from the cache and affect performance adversely. Reconciling prefetching and caching therefore involves quantitative benefit-cost considerations to throttle overly aggressive prefetching (see, e.g., [17]). To this end, a self-tuning method is once again called for.

A natural idea is to estimate near-future access probabilities on the basis of the stationary heat statistics that an LRU-k-like cache manager would keep anyway (see Section 2), or the corresponding temperature values when data units have variable size. Such a method, coined *temperature-based vertical data migration* in [12, 13], keeps a list of the top-temperature non-cached data units and considers their prefetching in descending order of temperature, but the prefetching is initiated only when the corresponding document's temperature exceeds the temperature of the document(s) that would need to be dropped from the cache instead.

When the latencies of fetching non-cached documents vary significantly, especially when tertiary storage is involved where "offline volumes" (i.e., tapes or (magneto-)optical platters) may first have to be mechanically loaded into a drive, the above benefit-cost consideration should be further refined by explicitly considering the fetch costs of non-cached documents (see also [18, 19] for similar considerations regarding cache replacement alone in settings like data warehouses and Web proxy servers). With a planning horizon of length T (e.g., the next second for prefetching from secondary storage or the next minute for tertiary storage), the expected number of accesses to document d within time T is $N_{spec}(d) = heat(d) \times T$. Then the *benefit* of prefetching document d is the product $\frac{N_{spec}(d)}{size(d)} \times fetch\_time(d, v)$, where $fetch\_time(d, v)$ is the estimated time for accessing d on its "home location" v where v can be secondary storage, an online volume in tertiary storage, or an offline volume, and the division by $size(d)$ is a normalization per cache space unit. The *cost* of this prefetching, on the other hand, is essentially the cost for accessing the corresponding cache replacement victim d' on its slower home location: $\frac{N_{spec}(d')}{size(d')} \times fetch\_time(d', v')$. Prefetching is considered beneficial only if its benefit exceeds its cost.

A cache manager that operates under this regime is self-tuning as it can automatically infer when aggressive prefetching should be throttled and when speculative prefetching pays off. The formulas for estimating the benefit and cost of a potential prefetching step can be made even more accurate by incorporating scheduling and queueing effects for the involved storage devices [13]. The method has been shown to be highly effective in experiments, especially for tertiary-storage archives, but it is attractive also for speculative prefetching from secondary-storage disks. Note that its overhead is low, comparable to the LRU-k bookkeeping.

6 Self-tuning Caching and Prefetching for Web-based Systems

The temperature-based prefetching method of the previous section has striven for a practically viable compromise between achieving a high prefetching benefit and keeping its bookkeeping overhead acceptably low. In particular, the bookkeeping metadata must not consume more than a trivial amount of cache space itself, and the time for making prefetching and cache replacement decisions should be at least an order of magnitude less than the time for a non-cached data access. When servers are accessed over the Web or use tertiary storage for infrequently requested but still "alive" data, the WAN or the tertiary storage incur a very high latency (and possibly also data transfer time, especially when the WAN is congested). This may call for an even higher fraction of data for which near-future accesses are anticipated to be downloaded to the client or a proxy server near the client in a timely manner (i.e., before the client explicitly asks for that data). So the stochastic prediction for near-future requests should be more "aggressive" but also needs to be more "accurate", as the cost of speculative wrong decisions also becomes higher. On the other hand, with a latency of several seconds that one aims to mask, one can afford both more space and more computation time for the stochastic predictions themselves.

A "richer" class of models for capturing reference patterns and predicting near-future accesses are Markov chains. Each active client connected to a proxy server can be viewed as a stochastic process that proceeds through states such that the current state corresponds to the document that the client has accessed last. State transitions then reflect the co-access patterns induced by the client. In a (first-order) Markov chain the probability for accessing document j right after document i does not depend on the prior history of the client's session and can thus be viewed as constant, unconditional probability p_{ij} that can be estimated by keeping online statistics. Then, knowing the current state of a client, one can compute the probability of a given document being requested within the next so many steps by a standard mathematical apparatus for the transient analysis of the Markov chain. These predictions would then drive the proxy server's prefetching and caching algorithm and/or the management of the data server's hierarchical storage (assuming tertiary storage is used).

Markov-chain-based algorithms have been intensively investigated for prefetching and caching (e.g., [5, 2]), but have previously overlooked the critical interdependencies between these two issues. Moreover, the prior methods have focused on the reference pattern of a single client and assumed a discrete time (i.e., simply counted access steps). In contrast, the recently developed *McMIN* method (Markov-chain based Migration for Near-line

Storage) [12, 13] takes into account the different interaction speeds of clients by using a continuous-time Markov chain (CTMC) for each client, where the residence times of states reflect the time that the client needs to issue its next request once it has obtained the document that corresponds to the current state. In Web-based access to a digital library, for example, bibliographies, abstracts, contents tables, results from search engines, or images may be browsed quickly whereas the user may spend much longer time on a hypertext document with a mathematical formula before proceeding along one of the outgoing links. The CTMC captures this variability; its transient analysis is a bit more involved than that of a discrete-time Markov chain. As shown in [12], it is possible to approximatively compute both the expected number of near-future accesses to a document d, $N_{spec}(d)$ (see Section 5), by appropriate precomputations and truncating negligible paths in the underlying traversal of the Markov chain. Moreover, both of these values can be aggregated over multiple CTMC models, one for each active client session, and the "arrivals" of new client sessions as well as "departures" of terminating sessions can be incorporated, too. Once the near-future access prediction is enhanced by the CTMC-based stochastic analysis, the same kind of benefit/cost assessment that we outlined in the previous section can be applied again. By taking into account the current states of sessions and the context-dependency of reference patterns, the effectivity of the prefetching is significantly increased, though [13].

7 Lessons Learned

This overview paper has sketched a suite of self-tuning methods for memory management, geared for centralized, high-speed interconnected, and widely distributed data servers. These methods are, to a large extent, complementary and may co-exist in a concrete system setting; for example, LRU-k as page replacement algorithm is a building block in goal-oriented distributed caching or temperature-based prefetching. The various tuning problems have been viewed as online optimization problems with dynamically evolving, statistically fluctuating input parameters. Consequently, the common paradigm that all approaches follow is that of an observe-predict-react cycle. Observation requires *online statistics*, prediction needs *mathematical models*, and reaction can often be organized as a *feedback loop*. These are the fundamental assets towards self-tuning memory management. For practically viable methods, however, one needs additional careful considerations on the following tradeoffs:

- The space needed for online statistics must be carefully controlled. In most situations, it is mandatory that the statistics are kept in memory for fast lookup. All methods presented in this paper can therefore be calibrated to adjust the amount of memory used for statistical load tracking. The actual setting of this calibration parameter has been driven by pragmatic considerations, however. More insight is needed on how much memory should be invested in the bookkeeping in order to auto-tune the remaining memory most intelligently, optimizing the overall use of memory.
- The complexity of the mathematical models used for predicting near-future accesses has ranged from lookups in the online statistics (e.g., in LRU-k) all the way to the transient analysis of Markov chains. Also, the metrics to be predicted have progressed from access probabilities to more complex notions of benefit and cost. So the CPU time overhead of predictions may be a critical factor, too. All methods have been devised so that this overhead can be controlled in a flexible manner, but one would also wish to have design guidelines for when it pays off to use more advanced prediction models.
- The reaction phase needs to adjust the memory usage according to the predicted reference patterns. This bears the risk of overreacting and ultimately ending up with oscillating, unstable adjustments of cache contents, memory-area sizes, etc. The presented algorithms have been designed so as to be robust in this regard. In particular, feedback-driven iterative adjustments have been shown to be well-behaved in this regard. Most strikingly, feedback-based methods work well even though the dependency of response time on the memory adjustment is merely approximated in a crude manner based on prior observations, without an explicit causal model. It would be desirable to obtain better insight in this issue from a control-theoretic viewpoint and be able to *guarantee* (fast) convergence after shifts in the workload patterns.

Much of this work is mature enough to be adopted by commercial data servers, and we believe that self-tuning algorithms will penetrate products and contribute towards zero-admin, trouble-free servers. With the explosion of data collection and generation on the Internet and the advent of mobile "gizmos" equipped with digital camera, speech recorder etc., distributed memory management and storage management in general will remain as great challenges, though. Individuals may record everything they hear or see [1] (e.g., during business meetings or vacation trips), most of which would be uploaded and ultimately archived on servers via wireless communication. One would ultimately expect that all data is virtually ubiquitous at acceptable speed and cost.

References

[1] Bernstein, P.A., et al., The Asilomar Report on Database Research, ACM SIGMOD Record 27(4), 1998.

[2] Bestavros, A., Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time in Distributed Information Systems, Data Engineering Conf., 1996.

[3] Brown, K., Carey, M., Livny, M., Goal Oriented Buffer Management Revisited, SIGMOD Conf., 1996.

[4] Chen, C.M., Roussopoulos, N., Adaptive Database Buffer Allocation Using Query Feedback, VLDB Conf., 1993.

[5] Curewitz, K.M., Krishnan, P., J.S. Vitter, Practical Prefetching Via Data Compression, SIGMOD Conf., 1993.

[6] Dahlin, M.D., Wang, R.Y., Anderson, T.E., Patterson, D.A., Cooperative Caching: Using Remote Client Memory to Improve File System Performance, Symp. on Operating Systems Design and Impl., 1994.

[7] Effelsberg, W., Haerder, T., Principles of Database Buffer Management, ACM TODS 9(4), 1984.

[8] Faloutsos, C., Ng, R.T., Sellis, T., Flexbile and Adaptable Buffer Management Techniques for Database Management Systems, IEEE Transactions on Computers 44(4), 1995.

[9] Franklin, M.J., Client Data Caching, Kluwer Academic Publishers, 1996.

[10] Gray, J., Graefe, G., The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb, ACM SIGMOD Record 26(4), 1997.

[11] Johnson, T., Shasha, D., 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, VLDB Conf., 1994.

[12] Kraiss, A., Weikum, G., Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions, VLDB Conf., 1997.

[13] Kraiss, A., Weikum, G., Integrated Document Caching and Prefetching in Storage Hierarchies Based on Markov-Chain Predictions, VLDB Journal 7(3), 1998.

[14] Leff, A., Wolf, J.L., Yu, P.S., Efficient LRU-based Buffering in a LAN Remote Caching Architecture, IEEE Transactions on Parallel and Distributed Systems 7(2), 1996.

[15] O'Neil, E.J., O'Neil, P.E., Weikum, G., The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD Conf., 1993.

[16] O'Neil, E.J., O'Neil, P.E., Weikum, G., An Optimality Proof of the LRU-K Page Replacement Algorithm, Journal of the ACM 46(1), 1999.

[17] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J., Informed Prefetching and Caching, ACM Symp. on Principles of Operating Systems, 1995.

[18] Scheuermann, P., Shim, J., Vingralek, R., WATCHMAN: A Data Warehouse Intelligent Cache Manager, VLDB Conf., 1996.

[19] Shim, J., Scheuermann, P., Vingralek, R., Proxy Cache Design: Algorithms, Implementation, and Performance, IEEE Transactions on Knowledge and Data Engineering, 1999.

[20] Sinnwell, M., König, A.C., Managing Distributed Memory to Meet Multiclass Workload Response Time Goals, Data Engineering Conf., 1999.

[21] Sinnwell, M., Weikum, G., A Cost-Model-Based Online Method for Distributed Caching, Data Engineering Conf., 1997.

[22] Teng, J.Z., Gumaer, R.A., Managing IBM Database 2 Buffers to Maximize Performance, IBM Systems Journal 23(2), 1984.

[23] Venkataraman, S., Livny, M., Naughton, J.F., Remote Load-Sensitive Caching for Multi-Server Database Systems, Data Engineering Conf., 1998.

[24] Weikum, G., Hasse, C., Moenkeberg, A., Zabback, P., The COMFORT Automatic Tuning Project, Information Systems 19(5), 1994.

DB2 Universal Database Performance Tuning

Berni Schiefer
IBM Toronto Lab
1150 Eglinton Avenue East
Toronto, Ontario, Canada M3C-1H7
schiefer@ca.ibm.com

Gary Valentin
IBM Toronto Lab
1150 Eglinton Avenue East
Toronto, Ontario, Canada M3C-1H7
valentin@ca.ibm.com

Abstract

DB2 Universal Database is the flagship Relational Database from IBM. By building on decades of ground-breaking IBM Database Research projects, DB2 Universal Database benefits from a pervasive and continuous infusion of advanced technology ranging from first-class optimizer and compiler technology to advanced locking, logging and recovery mechanisms, all designed with a common thread of high performance in both OLTP and complex query environments. This article will take the reader on a whirlwind tour of DB2 Universal Database, with a focus on the new auto-configuration tools. Find out how its open platform support exploits a rich repertoire of data storage techniques and how the latest Java-based administration tools can be used to quickly build and configure a DB2 Universal Database - for maximum performance.

1 Introduction

DB2 Universal Database is the flagship Relational Database from IBM. By building on the ground-breaking IBM Database Research projects, such as the relational model[CODD70], System R [ASTR76], R* [LMH+85], Starburst [HCL+90], and the SQL language itself [CHAM76], (see [DB299d]) DB2 Universal Database benefits from a pervasive infusion of advanced technology ranging from first-class optimizer and compiler technology to advanced locking, logging and recovery mechanisms, all designed with a common thread of high performance in both OLTP and complex query environments. Today DB2 Universal Database counts performance as one of its strengths.

This article will take the reader on a whirlwind tour of the tuning process in DB2 Universal Database, how its open platform support exploits a rich repertoire of data storage techniques and how the latest Java-based administration tools can be used to easily build and configure a DB2 Universal Database for maximum performance.

Also we include an introduction to the auto-tuning tools in DB2 Universal Database. These tools are geared towards simpler database administration and quick tuning - bringing forth high-performance database technology while simultaneously reducing the total cost of ownership.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Configuring the hardware

2.1 Disk striping and RAID

RAID 0 is the simplest of all disk layout schemes. In RAID 0, the data is not mirrored, but simply laid out in stripes across the available disks. But often RAID 0 is not an option. This happens when the data volume is very large (100 GB+), and disk failures are prone to happen frequently. Otherwise - RAID 0 gives good read performance and optimal write performance. For databases with fast recovery time, RAID 0 can be the most cost-effective solution.

RAID 1 keeps 2 copies of each page. This means slower writes/faster reads. In DSS workloads this is acceptable because they rely heavily on read performance. In OLTP workloads with heavy write/update activity, this will cause a performance degradation.

RAID 5 is used when disk redundancy is a must - but the budget is low. RAID 5 can be slower than RAID 1 because a write requires more processing for parity calculations.

RAID 10 is a combination of RAID 1 and RAID 0. It preserves the disk reliability of RAID 1 while approaching the performance of RAID 0. Note however, that the financial implications become significant. This configuration is the most expensive of the four described here, and is often deployed in database projects which are both mission-critical and performance sensitive.

Striping is done on two levels. The first level is the striping done by the disk arrays such as RAID, the second level is the striping of data onto separate disk arrays by DB2 UDB. On the RAID level, we make the distinction between a strip size[1] and a stripe size. On the database level, we make the distinction between an extent size[2]. and a prefetch size. A good rule of thumb is to make one stripe size equal to one extent size, so that the RAID striping matches the database striping. To better grasp the concepts behind striping, we will work through a disk layout example. In this example, we take an OLTP database, and place it on 8 RAID arrays, each RAID array containing 12 disks.

The smallest element is a strip, and it represents one disk on a RAID array. A large strip size (for example, 4 KB per disk) means that a single table update will only cause writing to one disk. So this is good for OLTP. Alternatively, a thin stripe would cause the disks to fill up the bus bandwidth more readily, pushing the bus to its full potential. Continuing with our OLTP example, we make the following calculation for RAID parameters:

- One strip size for OLTP = 4 KB
- One stripe size = (number of disks) * (strip size) = 12 * 4 KB = 48 KB

Next we deal with the striping on the database level. We have at our disposal 8 such arrays, which DB2 will see as 8 distinct containers. To take full advantage of the disk arms, we will spread the data across all 8 arrays.

- One extent size = One stripe size = 48 KB
- One prefetch size = multiple of ((extent size) * (number of containers))
- One prefetch size = multiple of (48 KB * 8)
- One prefetch size = 384 KB

[1] A strip size refers to the number of bytes in a stripe which are written to a single disk before continuing to the next disk on the stripe

[2] An extent size is the number of database pages which are written out to a single container before moving on to the next container in the table space

Here is a summary of the results for an OLTP system with 8 RAID arrays of 12 disks:

RAID strip 4 KB

RAID stripe 48 KB

DB2 extent 48 KB

DB2 prefetch size 384 KB

2.2 Memory Layout

In the simplest sense, memory is about caching parts of your database in memory to avoid disk I/O. DB2 Univeral Database supports the concept of buffer pools, which are large memory-caches of data.

- Non-leaf pages of indexes should be in memory. This means that an index-search can be completed using just one I/O.
- DB2 uses more memory than just for buffer pools. The two most important elements are sort memory and utility memory.

  Sort memory is controlled by the Sort Heap Size and Sort Heap Threshold configuration parameters.

  Memory used by utilitites is controlled by the Utility Heap Size configuration parameter, and is important for utilities such as LOAD and BACKUP.
- The remaining free memory can be allocated for buffer pools.

2.3 Disk Read/Write Cache

Disks have carried Read caches for a long time, and the purpose of these caches was to take advantage of locality in read patterns in the system behavior. Recently, with technology advancements, disks have begun sporting write caches as well. These caches are safe because they are static, and the information is preserved even when the power is cut off to the disks. These caches can be used to great advantage.

An obvious solution is to always enable read and write caches on all disks. But some systems will require the system administrator to make a tradeoff between read and write caches, by enabling the cache as a pure read cache, a pure write cache, or some measure in between. Also, a tradeoff may be required when purchasing the disks. Here are some recommendations to take full advantage of Read/Write caches.

Write caches will help log disks, especially for workloads with a high transaction rate. Write caches are useful for workloads containing occasional updates/inserts/deletes. Read caches are not as useful because the buffer pool does a better job of determining locality in data access. A good combination is to enable maximum read cache for OS and DB2 installation image, while enabling the write cache for data and log disks.

3 Building the Database

3.1 Table Space Layout

Data layout in table spaces is a key determining factor in database performance. The first rule is to take advantage of all the disks in every case. All table spaces should be split across all disks. DB2 Universal Database supports several methods for the layout of table spaces. The first such method is called DMS (Database Managed Space). DMS is a good choice for extra performance, but reduces usability somewhat. DMS table spaces let the database bypass the filesystem function supplied by the operating system and let the database control the disks directly. The net result is faster access to the data on the disk, and no interference from the operating system for journaling, caching, re-ordering, and other tasks. SMS (System Managed Space) is a strong alternative with good performance and high managability. It gives the user the freedom to take advantage of the filesystem functionality, which means that the database will be able to share disk resources with other applications.

3.2 Loading the Data

Loading the data is often a resource-intensive operation. Load is self-tuning and will adapt the loading process for those cases where large amounts of data are involved.

LOAD is a demanding high-volume I/O operation, it is recommended to take advantage of disk parallelism. If the source files can be placed across many disks (striped or otherwise) and/or there are several CPUs on the target system, then LOAD will recognize the available resources and use them.

Another method to improve LOAD performance is to streamline the LOAD operation with the subsequent gathering of statistics and index creation. DB2 Universal Database offers the opportunity of re-building the indexes and gathering statistics during the LOAD.

By telling the LOAD operation more about the nature and requirements of the task, the performance can be improved even further. For example, LOADs can be performed in a non-recoverable fashion, or more memory can be assigned for the database utility processing. All added information provides potential for increasing LOAD performance.

3.3 Creating Indexes and Gathering Statistics

The main indexes are built, when possible, together with the LOAD. However, circumstances may require that additional indexes be created. This may happen if new, and unexpected queries are submitted for processing. When considering indexes, the database administrator is often concerned with the tradeoff of disk space versus query performance, and query performance speedup versus potential insert/update/delete slowdown.

Creation of the largest indexes might be slow if not enough resources are available to perform the task. There are several things which can help to make it faster. The most obvious of these improvements is to give the index creation process more memory to work with by increasing the buffer pool, increasing the sort heap, enabling the SMP parallelism, or ensuring that the temporary table space is large enough and spread out across several disks.

In order to function optimally, DB2 Universal Database uses reliable and up-to-date statistics. There are several levels of statistics which can be collected, beginning with general information to detailed distribution statistics. For example, when the data values are skewed, the quality of the statistics will benefit greatly from the added information on distribution.

In some cases, the tables and indexes which are subject to this operation do not contain the data which they will contain during normal operation, and as a result produce misleading statistics. These tables are often referred to as ”volatile tables” simply because their contents change quickly in terms of size and values. Any table can be marked as having these properties by using the VOLATILE option on the ALTER TABLE command. It is essential that statistics are gathered at a time when the table contains typical values and typical amounts of data. Lastly, statistics must be gathered regularly, and the data can be reorganized to ensure its clustering. It is best to set up these operations on a scheduled basis so as not to forget.

4 Taking Advantage of the Auto-Tuning Tools

The configuration of DB2 Universal Database is substantially simplified and accelerated due in part to the auto-configuration and auto-tuning tools. Most of these algorithms and tools have been integrated into the database engine and are invisible to the database user.

This section covers two auto-tuning tools which have been externalized in DB2 Universal Database in the form of SmartGuides. By making these tools open, the user has the opportunity to supply more information about the workload, and become aware of the configuration changes brought on by the tools.

4.1 The Database Configurator Tool

One auto-configuration tool included in DB2 Universal Database is the Database Configurator Tool [3]. The purpose of this tool is to relieve the daunting task of learning about each configuration parameter and performing measurements to find each optimal value.

The auto-configurator tool does rely on some basic questions which it asks the user in order to better tune the underlying database.

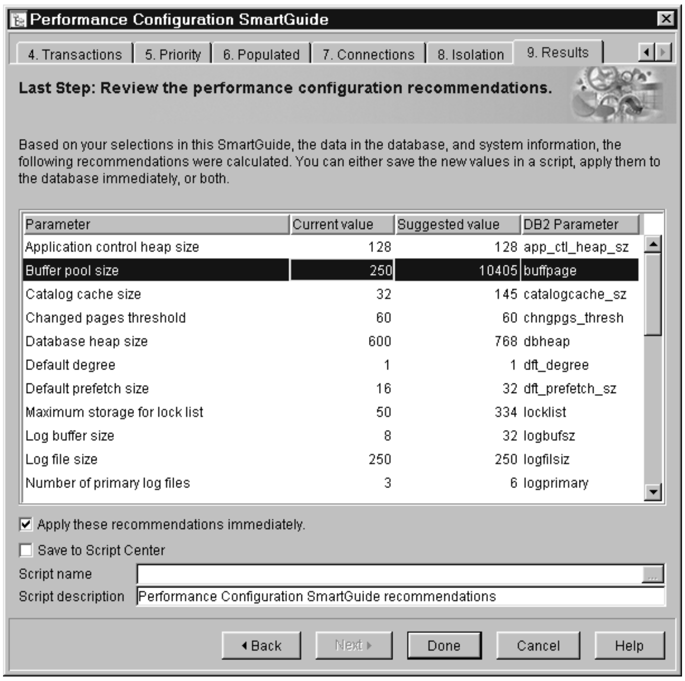


Figure 1: Analyzing the results

Figure 1 displays the resulting configuration parameters. This window reveals two things. The first is the number of configuration parameters, as you can see from the scrollbar, a modern commercial database is extremely configurable, and for many users it is a little too configurable.
The second thing made obvious in this picture is the recommendation for buffer pool size. As you can see, the default value of 500 pages (1 MB) greatly underutilizes the available resources! This machine had 128 MB of memory, and the tool decided to allocate 40 MB for buffer pool memory.

4.2 The Index Advisor

An important and time-consuming question for every database administrator is determining the right set of indexes. It requires an understanding of the dimensions of the data, of the workload exercised against the data, and

[3]To start the Performance Configuration SmartGuide, from the Control Center, click with mouse button 2 on the database for which you want to configure performance. Select 'Configure Performance' from the pop-up menu. The Performance Configuration SmartGuide opens. Follow the steps in the SmartGuide.

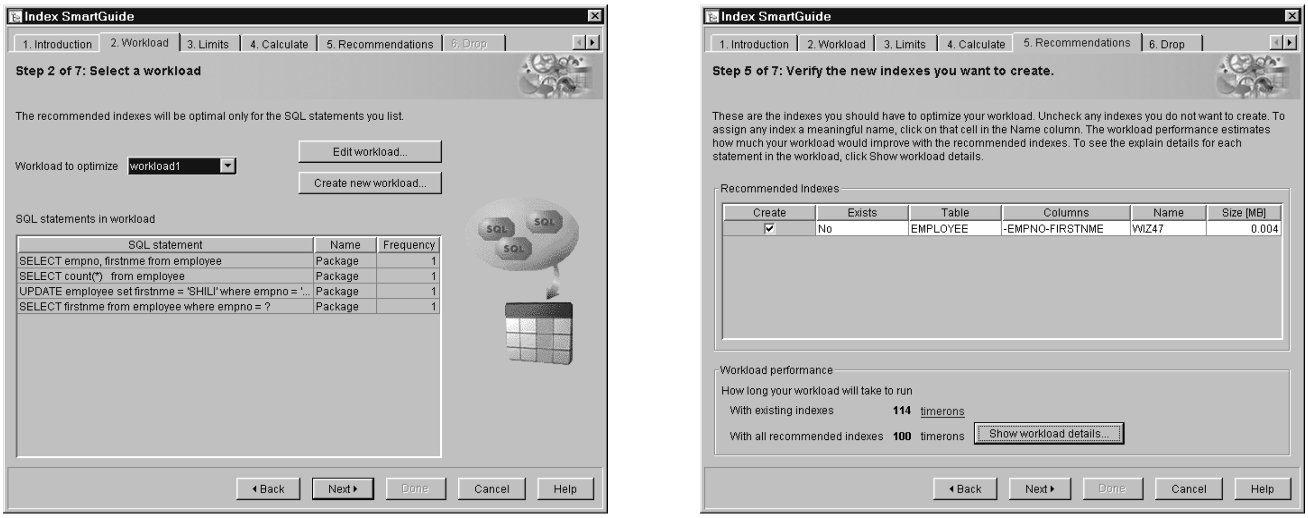
of the optimizer's methods in choosing an access plan. When the number of tables and columns runs into the thousands, no single administrator can handle this task without making a mistake along the way.

The Index SmartGuide [4] is a visual tool in DB2 Universal Database which helps the administrator choose indexes, and in some cases it relieves the administrator of the task altogether.

The main concept behind index advising is to suggest indexes to the administrator based on a representative SQL workload, and the underlying statistics of the data. The recommendation engine will take advantage of the optimizer and use a cost-based approach instead of heuristics or rules.

The index recommendation algorithm also considers the cost of maintaining indexes. In other words, if the workload contains many insert/update/deletes then some indexes can have a negative impact on performance because they need to be maintained. These impacts are taken into account. So it is better to include insert/update/delete statements as part of the workload description, to guarantee optimal recommendations.

The following snapshots show the two basic steps in the Index SmartGuide:



(a) (b)

Figure 2: Index Advisor

Figure 2(a)illustrates collecting the workload. The Index SmartGuide, when started, already knows about a workload by searching through the SQL cache, which stores recently executed dynamic statements. Other sources of SQL may be tapped, such as SQL stored in application packages and SQL stored in event monitor files. SQL can also be entered manually.

Figure 2(b) illustrates making recommendations. The recommendations are shown with details on tables, columns, size in megabytes, and direct improvement on total workload time. In this example, the workload was improved by only 10%.

5 Advanced Topics

As we have seen, some basic principles for the physical machine and physical database design as well as the use of intelligent tools to guide the tuning of the database can go a long way to providing an efficient well-tuned DBMS system. However, in certain application environments with unique requirements, or to achieve an even

[4] To start the Index SmartGuide click mouse button 2 on the Indexes folder and select Create - Index using SmartGuide from the pop-up menu. The Index SmartGuide opens. Follow the steps in the SmartGuide.

more highly tuned system, the DBA can use a wide range of additional functionality to control the behavior of the database very precisely. In this section we will explore some of this additonal functionality.

We begin with the physical database design. The use of multiple table spaces can give fine-grained control over the placement of data. For example, separating data and indexes into different table spaces or placing table data on only a subset of the disks available. The use of multiple table spaces also allows the database to be managed at a more granular level since many of the database utilities operate at a concurrency granularity of a table space (e.g. load, backup/restore). The characteristics of each table space can also be controlled, including defining a page size ranging from 4 KB to 32 KB, the size of an extent placed on each disk, the number of pages to prefetch from the table space, and a summary of the performance characteristics of the underlying disk devices.

Once multiple table spaces are in place, multiple buffer pools can be created to specifiy the caching behavior. Groups of tables representing one workload can be given priority over another group by allocating a proportionately larger percentage of available memory to one group. Alternatively, certain large tables can be restricted to a small buffer pool. In this case, large table scans will not compete with the buffer pool memory used by indexes, which is needed to avoid expensive random accesses. The result is better concurrency and more reliable performance characteristics.

The ALTER TABLE statement can also be used to more precisely define the role of each table and the access to it. For example, read only tables can choose to use a less granular level of locking using the LOCKSIZE option. Tables which log changes via continuous INSERT activity can benefit from specifying this using APPEND MODE. Space for future update activity can be reserved by using PCTFREE which controls how densely the LOAD utility populates the table pages.

Once the tables are defined and an initial set of indexes exist, the behavior of the indexes can also be specified using options on CREATE INDEX. For example, the freespace on each index page can increase or decrease depending on table usage using the PCTFREE option. Also, the set of columns present in unique indexes can be partitioned between those required for uniqueness and those columns included to obtain index only access to the table for certain queries by using the INCLUDE clause.

After the physical database design is complete, the execution behavior can be controlled both at the database manager (all databases in an instance) and the database level using a combination of database manager configuration parameters, database configuration parameters and registry settings. All of these are documented in the Administration Guide and allow you to override many of the default behaviors of DB2 Universal Database. In many cases DB2 offers a hierarchy of ways in which the behavior can be controlled. One example is the query optimization class which controls the intensity and scope with which the query optimizer explores the search space and considers different query rewrite and access path strategies. Prior to the execution of any individual DML statement the query optimization class can be specified using the SET CURRENT QUERY OPTIMIZATION statement. However, session level defaults for ODBC/CLI users can be set in the .ini file. Finally, a global database level default can be provided with the default query optimization class (dft_queryopt) database configuration parameter.

SQL statements come in different shapes and sizes, and sometimes contain idiosyncracies. For example, some statements are aggregations which return a small amount of data. Other queries return large volumes of data, but only the first few results are actually needed. Some queries are read-only, and others will be updated using WHERE CURRENT OF CURSOR processing. The optimizer can exploit this additional knowledge if the intent is signalled within the syntax of th SQL Query. For example, defining a cursor as FOR READ ONLY allows the optimizer to select access plans that do not have the ability to keep position on the base row. Similarly the FETCH FIRST N ROWS ONLY clause permits the optimizer to optimize sorts and result set processing for a specified and limited number of rows. Similarly the OPTIMIZE FOR N ROWS clause indicates that priority should be give to retrieving the first N rows as opposed to the entire answer set.

Finally, an important but often neglected area of performance tuning is the actual application design. There are many interfaces available for sending SQL statements to the database engine, ranging from static embedded SQL, through ODBC and JDBC, all the way to command interfaces like the command line interface and db2batch.

For further information on the ways in which access to DB2 Universal Database can be optimized for each of these interfaces refer to the DB2 Application Programming Guide and Reference.

6 Conclusion

Tuning DB2 is not difficult. Features have been added to make the database administrator's job easier. For the sizing and configuration phases, there are rules which help to provide a good starting point. Configuring and building the database is also made easier with the presence of both hidden auto-tuning and visual SmartGuides. Both help the user reach optimal database performance earlier in the deployment cycle, on all sizes of machines, from a personal database on a workstation to an enterprise solution on a parallel cluster. DB2 UDB offers these advantages without compromizing its configurability and flexibility, which have allowed expert administrators to implement advanced database layout and configuration. The auto-management technology has benefits for all levels of administrators. It lets the database user concentrate on his original concern: the safe storage and fast processing of enterprise data.

References

[ASTR76] Astrahan, M., M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson, System R: Relational approach to database management, ACM Transactions on Database Systems 1:2 (june 1976).

[CHAM76] "SEQUEL2: a unified approach to data definition, manipulation, and control," IBM J. Research and Development 20:6, pp 560-575.

[CODD70] "A Relational model for large shared data banks," Comm, ACM 13:6, pp 377-387.

[DB299a] DB2 IBM DB2 Universal Database Administration Guide, Design and Implementation (SC09-2839), IBM Corp, 1999.

[DB299b] DB2 IBM DB2 Universal Database Administration Guide, Performance (SC09-2840), IBM Corp, 1999.

[DB299c] DB2 IBM DB2 Universal Database Command Reference (SC09-2844), IBM Corp, 1999.

[DB299d] DB2 IBM DB2 Universal Database SQL Reference, Volume 1 and Volume 2, (SBOF-8923), IBM Corp, 1999.

[HCL+90] L.Haas, W.Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita Starburst mid-flight: As the dust clears. IEEE Trasactions on Knowledge and Data Engineering, March 1990.

[LMH+85] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger and P. F. Wilms. Query processing in R*. In Query Processing in Database Systems, (W. Kim, D. S. Reiner, and D., S. Batory, eds.), Springer-Verlag, 30-47, 1985.

Self-Tuning Technology in Microsoft SQL Server

Surajit Chaudhuri, Eric Christensen, Goetz Graefe, Vivek Narasayya, Michael Zwilling
Microsoft Corporation
One Microsoft Way, Redmond, WA, 98052
Email Contact: surajitc@microsoft.com

1 Introduction

Today's databases require database administrators (DBAs) who are responsible for performance tuning. However, as usage of databases becomes pervasive, it is important that the databases are able to automatically tune themselves to application needs and hardware capabilities rather than require external intervention by DBAs or applications. Thus self-tuning databases would result in significant reduction in the cost of ownership of databases and would encourage more widespread use in many nontraditional applications. However, making database management systems self-tuning require significant understanding of DBMS components and relationships among many of the "tuning knobs" that are exposed to the application/user. Microsoft SQL Server is committed to the vision of self-tuning databases. In this short paper, we review some of the recent advances in making Microsoft SQL Server self-tuning. Section 2 describes a physical database design tool, available with SQL Server 7.0, that automatically picks indexes appropriate for a SQL Server database. Section 3 and Section 4 describe the self-tuning advances in the query engine and the storage engine respectively for Microsoft SQL Server 7.0.

2 Physical Database Design Tool

One important task of a database administrator is selecting a physical database design appropriate for the workload of the system. An important component of physical database design is selecting indexes since picking the right set of indexes is crucial for performance. In this section, we describe an Index Tuning Wizard for Microsoft SQL Server 7.0 that automates this challenging administrative task while achieving performance competitive with that of systems cared for by DBAs. The tool was conceived in the AutoAdmin research project [1] at Microsoft the goal of which is to develop technology to ensure that database systems have sharply reduced administration overhead and total cost of ownership. The index-tuning wizard is the result of collaborative work between Microsoft Research and SQL Server.

When viewed as a search problem, the space of alternatives for indexes is very large. A database may have many tables and columns, indexes may be clustered or non-clustered, single-column or multi-column. The "textbook solution" of taking semantic information such as uniqueness, reference constraints and rudimentary statistics ("small" vs. "big" tables) to produce a database design performs poorly because they ignore valuable workload information. Even when index selection tools have taken the workload into account, they suffer from being disconnected from the query optimizer, e.g., [6]. These tools adopt an expert system like approach, where the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

knowledge of "good" designs are encoded as rules and are used to come up with a design. These tools operate on their private model of the query optimizer's index usage. Modern query optimizers use complex strategies such as index intersection and index-only access. Thus, making an accurate model of the optimizer and keeping it consistent as the optimizer evolves is an extremely difficult task and will likely result in selection of indexes that the optimizer may not use as desired by the index selection tool in generating a plan.

2.1 Functionality of the Index Tuning Wizard

For a given database, the index tuning wizard requires a workload file (or table) as its input. The workload file is used to capture the queries and updates that are executed against the database system. Any SQL Server Profiler trace can be used as a workload file. SQL Server Profiler is a graphical SQL Server client tool that makes it possible to monitor and record DBMS events in a workload file. A typical entry in such a workload file may consist of a variety of fields: event-class, text of the event (for example, text of the Transact-SQL query), start-time, and duration of the event. The Index Tuning Wizard can extract engine relevant events (such as SQL statements) and fields from a SQL Server Profiler trace automatically.

The most important output from the Index Tuning Wizard is a set of recommended indexes. The wizard also produces an estimate of expected improvement in the execution of the workload compared to the existing configuration. The Index Tuning Wizard uses the optimizer component of the query processor to project the above estimate. The Index Tuning Wizard recommendations are augmented by a variety of reports that provide further analysis of the recommendations and their quantitative impact. These reports affect the decision about whether the recommendations should be accepted or rejected. For example, the index usage report presents information about the expected relative usage of the recommended indexes and their estimated sizes.

The Index Tuning Wizard can be customized in several ways depending on the requirements of the user. It is possible to configure it to tune choice of indexes over selected tables only. It can also be used in a mode where it only considers adding indexes to the existing set of indexes. Finally, an upper limit on available space may be specified. Further details of the tool are available from [3].

2.2 Architectural Overview

Figure 1 provides an architectural overview of the Index Tuning Wizard and its interaction with SQL Server. The Index Tuning Wizard takes as input a workload of queries on a specified database. The tool iterates through several alternative *configurations* (a configuration is a set of indexes), and picks the configuration that results in lowest cost for the given workload. Clearly, the obvious approach of evaluating a configuration by physically materializing the configuration is not practical since materializing a configuration requires adding and dropping indexes that can be extremely resource intensive and can affect operational queries on the system. Therefore, the Index Tuning Wizard needs to simulate a configuration without materializing it. The Index Tuning Wizard uses the optimzers cost estimate for the workload for comparing alternative configurations. Therefore, we have extended Microsoft SQL Server 7.0 to support the (a) ability to *simulate a configuration* and (b) *optimize a query for the simulated configuration* [4].

The Index Tuning Wizard may have to evaluate the cost of many alternative configurations by calling the optimizer. Invoking the optimizer can be a relatively expensive operation since it involves crossing process boundaries from the tool to the server. The *cost evaluation* module significantly reduces the number of optimizer invocations by exploiting the commonality among possible configurations.

We now describe the strategy used by the Index Tuning Wizard to pick single column indexes. As mentioned earlier, the tool also considers multi-column indexes, and we discuss later in this section how we deal with the large space of multi-column indexes. An enterprise database may have many tables, and queries in the workload may reference many columns of these tables. Therefore, if the Index Tuning Wizard were to consider an index on each column referenced in the workload, the search would quickly become intractable since the tool has to

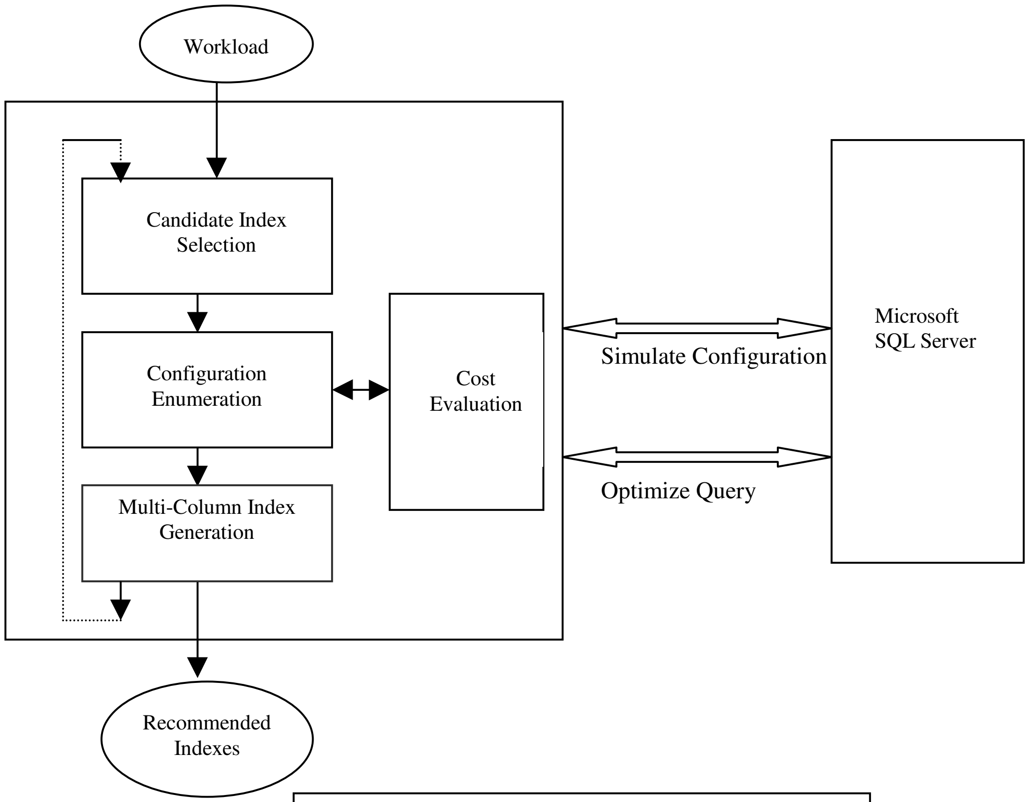


Figure 1. Architecture of Index Tuning Wizard

consider combinations of these indexes. The *candidate index selection* module addresses this problem by examining each query in the workload one by one and then eliminating from further consideration, a large number of indexes that provide no tangible benefit for any query in the workload. The indexes that survive candidate index selection are those that potentially provide significant improvement to one or more queries in the workload (referred to as candidate indexes). The *configuration enumeration* module intelligently searches the space of subsets of candidate indexes and picks a configuration with low total cost. The tool uses a novel search algorithm that is a hybrid of greedy and exhaustive enumeration. By being selectively exhaustive, the algorithm captures important interactions among indexes. However, the basic greedy nature of the algorithm results in a computationally tractable solution.

Multi-column indexes are an important part of the physical design of a database; increasingly so with modern query processors. A multi-column index can be used in a query that has conditions on more than one column. Moreover, for a query where all the required columns of a table are available in a multi-column index, scanning the index can be more attractive than scanning the entire table (index-only access). However, the number of multi-column indexes that need to be considered grows exponentially with the number of columns. The Index Tuning Wizard deals with this complexity by iteratively picking multi-column indexes of increasing width. In the first iteration, the tool considers only single-column indexes; in the second iteration it considers single and two-column indexes, and so on. The *multi-column index generation module* determines the multi-column indexes to be considered in the next iteration based on indexes chosen by the configuration enumeration module in the current iteration. The index selection algorithms are described in details in [2].

3 Relational Engine and Query Processor

In the relational engine, there are two particularly interesting instances of self-tuning. They manage memory and statistics.

3.1 Memory Management

The memory allocated by the server process is shared among multiple uses, including the procedure cache (which retains compiled SQL statement ready to run), query processing (sort and hash operations), and database utilities (in particular index creation and consistency checks). The query processor has to allocate its memory not only between competing queries but also within queries, i.e., among parallel threads of a single query as well as among query evaluation operators (sort and hash) that produce and consume data within a single query phase.

The procedure cache retains plans for reuse, just as the I/O buffer retains plans for reuse. Therefore, the procedure cache participates in the replacement policy of the buffer manager. Obviously, the size of a cached plan as well as the cost of recreating it (which is typically much higher than replacing a disk page) must be taken into consideration.

Multiple concurrent complex queries create memory contention among queries. If asynchronous I/O works well, including asynchronous I/O for resolving record pointers to full records, very few threads each executing a complex query will load a CPU such that additional threads will not increase system throughput. Therefore, we focused on enabling and exploiting asynchronous I/O and then limited the number of concurrent complex queries, thus giving each query, once admitted to execution, more memory. The number of queries admitted at any time depends on, among other considerations, the size of the data manipulated and therefore the memory used by the queries already executing and the next one being considered for admission.

Within each query, operator phases and plan phases are key considerations for memory allocation. For example, if a merge join obtains its data from two sort operations, the first sort should use all memory until only a final merge step is left, then the second sort uses all memory, and then the two sort operations compete for memory while both produce data for the merge join out of the two final merge steps. Of course, there are several exceptions to this scheme if one or both sort inputs fit into the available memory.

We implemented this behavior using the notions of waiting operators and waiting memory. While an operator waits, it may retain its working memory, but it has to register the fact with a per-query memory manager. Other, active operators can request such memory, and the waiting operator is forced to free its memory. For example, if the first sort in the example above can fit its data into memory, it will not write its data to a run file unless it is forced to do so when the second sort requests that memory. On the other hand, if both sort inputs fit in memory at the same time, no I/O at all will occur. If multiple operators are waiting, the longest-waiting operator is the first to free its memory. In a way, this is a LRU scheme among active and waiting query operators. More about the operation of hash operations and this memory management scheme can be found elsewhere [5].

3.2 Distribution statistics

A non-procedural persistent programming language such as SQL as well as physical data independence create great opportunities for query optimization, but they also require careful query optimization to achieve acceptable performance. Query optimization depends on compile-time estimates of run-time results, e.g., the cardinality of intermediate tables. The more sophisticated the query processor, the more choices it can and must make in response to logical and physical database state as well as the request being optimized. However, the more choices, the more opportunities there are for mistakes. Therefore, modern query optimizers depend on database statistics even more than their predecessors. Typical database statistics include minimum, maximum, histograms, frequent values, and counts of distinct values for individual and combinations of database columns.

Creating and updating database statistics can be onerous, for two reasons. First, someone has to determine what statistics ought to exist or ought to be refreshed. Second, those statistics must be gathered from the database.

In SQL Server 7.0, both reasons are greatly alleviated. Unless this feature is explicitly disabled, whenever the query optimizer would benefit from statistics that currently dont exist, it creates those statistics. If statistics are found to be out-of-date, they are refreshed. Note that statistics are created or refreshed only if they are truly useful, not simply because data have been created or updated, or because a given time has gone by since the last refresh. Statistics are presumed out-of-date if a given fraction of a table's rows have been inserted, updated, or deleted since the last statistics refresh. In order to ensure that OLTP overhead is minimal, a count of updates is used that is probabilistic, very inexpensive, and sufficiently accurate.

When statistics are gathered, small tables are inspected in their entirety, whereas large tables are sampled. The absolute sample size grows with the table size, whereas the relative sample size shrinks with a growing table size. For all but the very largest tables, creating or refreshing statistics takes only a few seconds. In other words, compilations might be held back for a few seconds in order to enable accurate, reliable, and repeatable query optimization. Concurrent execution of other queries and updates is not held back, because statistics gathering does not require transaction consistent database data.

4 Storage Engine

The SQL Server Storage Engine (SE) provides facilities such as access methods, concurrency control, recovery, sorting, buffering, disk space allocation, I/O, and overall memory management. Previous versions of the SE had 35 server-wide tuning knobs controlling various aspects of its operation. Microsoft SQL Server 7.0 eliminates many of the knobs by making them self-tuning or by using new algorithms that make them unnecessary. The result is that the SE has only 16 server-wide knobs, 6 of which are self-tuning with the remaining knobs providing extra control for extreme conditions.

The self-tuning features of the SE are critical for low-end of the market where the DBMS resides on desktops/laptops where it shares resources with applications or on small business server machines where it shares resources with other servers for email and web pages. Users may not even be aware that there is a DBMS on the machine as it is just a piece of an application. In this environment the SE not only needs to tune for its optimal performance, but for the applications on the machine as a whole. Self-tuning is also important for the high-end where workloads change throughout the day and no static configuration is optimal. Also, in this environment DBAs are under pressure to bring new DB applications into production without time to fine-tune performance.

4.1 Dynamic Resource Allocation

Many DBMS systems use knobs to set the maximum values for certain resources and often the resources are statically allocated thus taking up valuable memory or disk space even when unused and often require restarting the server to change them, which is not tolerable in a 24x7 environment. Even if not statically allocated, once the resources are allocated they frequently are not deallocated. Some examples of these knobs from the previous version of the SE are memory for user connections and lock structures and disk space managed by the server. Without dynamic allocation, the knobs must be set for worst-case conditions.

Dynamic disk space allocation is an especially important feature for users on desktop machines where the expectation is that files grow and shrink depending on the amount of data in them. The SE supports specifying initial/maximum sizes and growth rates for database files. Growth rates are important for efficient disk space allocation and are typically set to 10should grow 10database files in a desktop environment. Shrinking a file can be quite expensive as it involves moving data from the end of the file to the unused portions in the middle in order to truncate the end and free space to the file system, thus a knob still exists to control whether file shrinking is attempted [7].

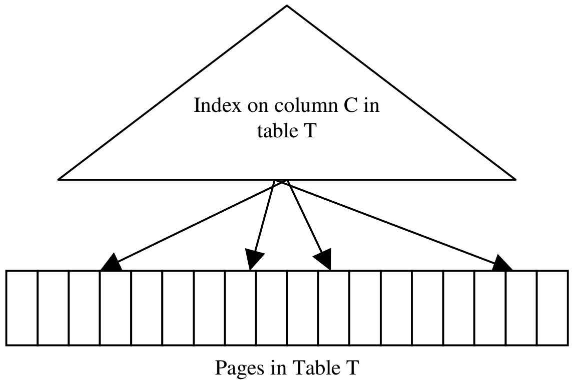


Figure 2. Base Table Access through an Index

4.2 Resource Optimization Strategies

Another type of self-tuning is gathering data about an operation and current resource availability and then producing an optimized plan for the use of those resources. The SE uses several optimization techniques, two that we highlight here: determining locking granularity for a query and directing read-ahead.

The dynamic locking feature optimizes locking granularity, per-index, for a query, deciding between table, page, and row locks. Although fine-granularity rowlocks provide the best concurrency, they are expensive in terms of the CPU cost to acquire and memory to hold them. The optimization takes into account degree of concurrency, CPU time to acquire a lock, memory to hold the lock, row size, row density (number of rows accessed per page accessed), number of pages, transaction isolation level (i.e. lock duration). Much of this information comes from the statistics already used for query optimization.

For example, consider the locking strategy for a query on table T with an index I on column C where the query selects a range on column C and returns other columns from the rows. Figure 2 shows how the row locators from a small range of pages in C can point to rows in physically disparate pages in the table. For this query it is desirable to use page level locking for the index lookups since every row is examined on the relevant pages . On the base table, row-level locks are desirable since only one row out of each page is accessed. However, if each row generally takes up the entire page, then page locks are sufficient.

Previous SE versions performed read-ahead by detecting requests for contiguous disk pages and had many parameters to tune the detection. SQL Server 7.0 eliminates these parameters using two new strategies [7]. In index-directed read-ahead, a check is made during the initial descent of an index to see if multiple leaf pages may need to be scanned to match the range of the scan. If so, read-ahead is started on each relevant page, with contiguous pages being coalesced into a single read request. In bitmap-directed read-ahead, the bitmaps tracking allocated pages are used to drive physically ordered reads across multiple disk devices at once. SQL Server still has read-ahead depth knob limiting the number of outstanding asynchronous I/Os on a device. We plan to use feedback techniques discussed in the next section to eliminate even this knob.

4.3 Feedback-based Techniques

Feedback-based tuning can work in concert with optimization techniques to compensate for inaccurate planning data, approximations in rules and cost functions, and changing execution conditions. Currently only a few feedback techniques are used in the SE, but we see the potential for many more. We outline memory management and read-ahead techniques here.

The buffer manager controls the caching of database pages. To be effective, it must make as many database pages as possible available to the rest of the server without driving the entire system into demand paging. This is accomplished in 7.0 by recognizing and responding to varying memory demands upon the entire system. Once a second, the buffer manager queries the OS for the amount of free physical memory available. If the amount falls below a certain threshold, the buffer manager will free up some of its buffers in an attempt to keep the OS from having to page it or other processes out to disk. If the amount of free physical memory rises above a certain threshold, the buffer manager can allocate additional buffers, thereby increasing the cache hit rate.

Thus the OS provides indirect feedback to the buffer pool through the free physical memory value, causing the SQL Server process to grow and shrink dynamically[1]. This mechanism allows the server not only to utilize all available memory to increase system throughput (which has heretofore been difficult for the DBA to achieve), but to reduce its memory usage when other applications on the machine need memory and to reclaim memory when an application ends as well.

We are developing feedback techniques to throttle the read-ahead mechanism for a future version. One technique is to monitor whether the consumer of the reads is able to keep up with the current I/O rate and adjust read-ahead depth (i.e. number of asynchronous requests) accordingly. Another technique is to notice if increasing the depth actually increases I/O throughput and backing off if it does not. This mechanism can respond to system I/O load caused by other applications using the disk, by reducing read-ahead to optimize overall system performance.

References

[1] AutoAdmin Home Page. http://www.research.microsoft.com/db/AutoAdmin

[2] Chaudhuri S., Narasayya V., "An Efficient, Cost-driven Index Tuning Wizard for Microsoft SQL Server." Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece. pp. 146-155.

[3] Index Tuning Wizard for Microsoft SQL Server 7.0: http://www.microsoft.com/SQL/70/whpprs/indexwp.htm

[4] Chaudhuri S., Narasayya V., "AutoAdmin "What-If" Index Analysis Utility." Proceedings of ACM SIGMOD 1998, Seattle, USA.

[5] Graefe G., Bunker R., Cooper S., Hash join and hash teams in Microsoft SQL Server, Proc. VLDB Conf. 1998, pp. 86-97.

[6] Hobbs L., England K., "Rdb/VMS A Comprehensive Guide", Digital Press, 1991.

[7] Soukup S., Delaney K., Inside SQL Server 7.0, Microsoft Press, Redmond, WA, 1999.

[1] This feedback mechanism and all other SQL Server features rely only on publicly documented features of Windows NT.

Performance Challenges in Object-Relational DBMSs

Muralidhar Subramanian, Vishu Krishnamurthy
Oracle Corporation
Redwood Shores
CA 94065

Abstract

Traditionally, relational database systems have excelled in handling relatively simple data, such as numbers, characters and dates. With the advent of the Internet, new programming languages such as Java, and new interchange languages such as XML, developers deal with complex data that need to be efficiently stored and managed. Extending Relational systems with Object capabilities has the potential to meet this challenge, but the handling of the complex data introduces new performance challenges that must be met by the resulting ORDBMSs. The principal performance challenges are found in the following areas: (i) handling collections of objects; (ii) handling domain specific data (e.g. Text, Spatial); (iii) handling Object References (e.g. Navigational Queries); (iv) handling Object Behavior; (v) mapping and handling objects on the client-side. In this paper, we will elaborate on the performance challenges in these various areas and discuss ways to address them.

1 Introduction

The tsunami of internet applications has brought newer and more complex kinds of data to hundreds of users around the globe. Object-relational technology has evolved relational database systems enabling them to store complex data: structured, unstructured or semi-structured. Once your objects live inside the database, you get to persist your object model, query without loss of encapsulation, and build robust and reusable database code.

Object-relational systems reduce the impedance mismatch between the application and the data stored in the database. Implementing the object abstraction layer on top of the relational infrastructure poses certain performance challenges.

Consider for example a typical e-commerce application that implements the purchase order system for a personal computer vendor. The data model for this system consists of:

- A purchase order object that contains a set of line items, a reference to a customer object, the shipping address, the order date and ship date.
- Each line-item contains a reference to the item, the quantity of order, any discount amount
- A customer object that contains customer name, customer address, customer credit information

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- An item object that captures the specification of a computer part (note that this may form an aggregation hierarchy of other parts/subparts), an image of the item, a text of the item description and warranty information, and the price of the item

An object-relational database management system (ORDBMS) supports the structure and complex relationships of the above data model such that one can query the database semantically and fetch objects from the database preserving their structure. For example, retrieving the purchase order for a given customer must return the order entry including all the line-items information. Or, for example, one should be able to query the purchase-order table to navigate the customer reference to obtain those orders that reference customers who live in California.

In addition to structure, a typical ORDBMS supports the ability to capture the behavior of objects as methods. For example, the purchase-order object may have a method that computes the total value of the purchase order. Capturing complex business logic as methods in the database enables applications to query and retrieve objects more meaningfully and efficiently. For example, the total value method enables the application to retrieve those orders whose value exceeds $1000. Contrast this to fetching the purchase order objects from the database to the client and performing the total value computation on the client-side to identify the purchase orders. Clearly, running data intensive application logic in the server has inherent performance advantages. [1] discusses the benefits of ORDBMSs and highlights the performance issues. [2, 3] are the benchmarks that exist server-side and client-side performance of ORDBMSs. In this paper, in the context of the above example we will highlight certain performance challenges that an object-relational system faces in support of the above modeling abstraction and offer some solutions.

2 Handling Collections of Object Data

Collections are essential to the object-relational type system to model multi-valued properties and relationships that are common in business applications. For example, the set of line-items in the purchase-order can be modeled using a collection type.

There are different ways of storing collections: as binary data or as rows in a table in an ORDBMS. Each method has its performance trade-offs. Storing collections by mapping its elements to rows of a table provides for better querying of the collection data as indexes may be created on these storage tables. For example, the line-items of purchase-order objects may be mapped to rows of a storage table by maintaining some referential keys between the purchase-order table and the line-items storage table. This storage model allows for efficient querying across line-items of all purchase-orders, for example, to retrieve those orders which contain a specific item.

However, if the collection has to be retrieved as an atomic value, then the ORDBMS has to construct the collections by aggregating the relevant rows of the storage table that constitute the collection value. Representing the collection as binary data, on the other hand, facilitates retrieval of the collection as an atomic unit since no construction needs to be done; however, querying of the collection is hampered since the binary data needs to be interpreted by the ORDBMS to read the collection elements. For example, the customer's list of phone numbers may be better represented as a collection stored as a binary value if the application always retrieves them as a single value and never queries its content.

It should also be noted that greater concurrency and finer level of logging is realized when the collection elements are mapped to rows of a storage table.

Representing large collections, such as time-series data, is another challenge faced by an ORDBMS. When an object containing such a collection is retrieved, it would be prohibitively expensive or nearly impossible to retrieve the collection data as part of the object's state. For these situations, an ORDBMS can provide the facility to return a locator to the collection instead of its value. Collection locators allow application to retrieve large collections without materializing the collections in memory. This allows for efficient transfer of such collections across interfaces. In such situations, the ORDBMS must also support querying of these collections returned as locators to enable the application to retrieve a subset of the collection data.

3 Handling Domain Specific Data

Multimedia data types like images, maps, video clips, and audio clips were once rarely seen outside of specialty software. Today, many web-based applications need their database server to manage such data and provide the ability to query the content of such data and correlate it with other data in the database. Typically, database systems provide the way to store such content-rich data as binary streams in the database. However, efficient access and querying of such data pose a significant performance challenge.

Typical DBMSs support a few types of access methods (B+Trees, Hash Indexes) on some limited set of data types such as numbers, strings, etc. There is a growing need for indexing content-rich, domain-specific data, and also specialized indexing techniques. For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips and other complex data types that require content-based retrieval. Complex data types have application specific formats, indexing requirements, and selection predicates. For example, there are many different encodings (e.g., ODA, SGML, plain text) and information retrieval techniques (e.g., keyword, full-text boolean, similarity, probabilistic, and so on) for documents. Similarly, R-trees are an efficient method of indexing spatial data. No database server can be built with support for all possible kinds of complex data and their indexing requirements.

An ORDBMS should be an extensible server that enables the type definer to define new index types and operators in support of these complex data domains. The extensible framework of the ORDBMS should support the ability for the definer of the index type to specify its cost model so that the cost-based optimizer of the ORDBMS can evaluate the cost of the various access paths for a cross-domain query and select the optimal one.

In our sample application database, creating such domain indexes on the warranty text and the image of the item enables one to query the purchase orders to select those orders with item lists containing an item whose image matches a given input image and the warranty text contains a specific clause.

4 Handling Object References (Navigational Queries)

References in object-relational systems are the equivalents of primary key-foreign key relationships in relational systems. In addition to providing a more natural modelling of relationships between objects, references also allow users to specify more concise queries with path expressions (avoiding explicit specifications of multiple joins).

Navigational queries involve traversals through references. For example, to obtain the set of purchase orders of customers who live in California, the purchase order table is scanned, the target object of the customer reference is looked up - a process referred to as de-referencing the reference - and the predicate is applied on the address attribute of the customer object.

This, however, is not necessarily the most optimal solution. In many cases, the target objects corresponding to a set of references reside in the same table. Potentially better query plans can be generated if the optimizer is aware of this information.

A scoped reference is a reference whose target object resides in one well-known table - the scope table. Dereferences through scoped references can be converted into (outer) joins (on the object identifier columns) with the scope table(s). This allows the optimizer to pick alternate access paths for the query. For instance, it is possible that the predicates on the scope table are more selective than the predicates on the driving table - as is the case in our example above. Converting the operation into a join allows the optimizer to consider more optimal plans.

Scoped references offer storage benefits as well. An unscoped reference must contain an identifier for the table containing the target object of the reference. (This is not completely true. Systems that support global database-wide indexes on object identifiers do not need table identifiers to be part of references.) A scoped reference does not need to, since the table containing the target object is well-known.

5 Handling Object Behavior

In an object-relational database, methods are the natural way to model object behavior. Regardless of what language the method is written in, method invocation is usually an expensive operation.

With the support for object types, methods run inside the server. Method performance in queries can be addressed by employing functional indexes and in-lining of method expressions

Functional indexes provide one way to improve method performance. A functional index, as the name suggests, is an index on the result of a function. During query analysis, the optimizer can identify functions that have indexes defined on them, and use the index access path for the function, rather than evaluating the function.

In many cases, object methods tend to be fairly simple. They are written as methods to provide encapsulated access to objects. For example, a *TimeForDelivery* method on the purchase-order object might simply compute the difference between the date the order was placed and the date it was delivered. Function in-lining is a useful mechanism to improve performance of such queries. In essence, with function in-lining, the body of the function is in-lined into the query, thus allowing the query optimizer to choose (potentially) more optimal plans. In addition, the method invocation overhead can also be avoided.

As we move from traditional business applications to newer domains, function logic tends to get more complicated. Optimizers in relational systems typically ignore the cost of evaluating functions. Object-relational databases should provide a way for users to register cost (and selectivity) information about functions, and then use this information to generate appropriate plans. Reordering predicates with expensive functions, and predicate migration are some strategies that can be employed in this regard.

6 Mapping and Handling Object data on the Client-side

Object data tend to be very structured and deeply nested. Applications fetch these objects from the database and navigate their relationships, in the process requiring other objects to be fetched from the database. Pure object-oriented database systems tend to be better at addressing this requirement since they support a memory persistence storage that inherently co-locates related objects on disk, and a page-based caching scheme that causes pages to be faulted in from disk to memory enabling several related objects to be fetched in one round-trip. Client-side management of objects pose a significant performance challenge for ORDBMSs whose storage is more relational.

In our purchase-order example, the application may wish to fetch the part hierarchy of a given item, make necessary changes, and persist the complex item in the database. Clearly, making separate roundtrips to the server to fetch every subpart of the item would be very inefficient.

Object-relational systems should provide a client-side object cache to aid in the memory management of the objects and provide a transparent persistency service. With the help of features like complex-object-retrieval (fetching and manipulation of a set of related objects as a single unit) into the object cache, clustering of related objects on disk, performance can be greatly improved.

With the help of an object cache, applications can "pin" an object reference to load the object from the server into the cache (if not already present in the cache) and "flush" to return and persist the modified object to the server. Storing the segment key or ROWID of the object as a hint in the object reference enables the ORDBMS to directly locate the object data on disk (without any index or table scans) in many cases. Updating the object in the client cache raises the issue of flushing the changes made by the client to the server. The object cache should provide an ability to flush all or selected updated objects to the server in one network roundtrip.

Acknowledgements

The authors wish to thank the following people for their valuable input and feedback: Jags Srinivasan, Nipun Agarwal, Ravi Kasamsetty, Ravi Murthy, Sandeepan Banerjee and Anil Nori.

References

[1] Object-relational DBMSs - Tracking the Next Great Wave, M. Stonebraker, P.Brown, Morgan Kaufmann, Second Edition, 1999.

[2] The Bucky Object Relation Benchmark (with M. Carey, J. Naughton, M. Asgarian, J. Gehrke, D. Shah), Proceedings of the 1997 SIGMOD Conference, Tucson, Arizon, May, 1997.

[3] The OO7 benchmark (with M. Carey and J. Naughton), Proceedings of the SIGMOD Conference, Washington, DC, May, 1993.

Performance Tuning for SAP R/3

Alfons Kemper *Donald Kossmann* *Bernhard Zeller*
University of Passau, Germany
`http://www.db.fmi.uni-passau.de`

1 Introduction

SAP R/3 is the most successful product for enterprise resource planning (ERP). It is used by most Fortune 500 companies and comprises modules for human resource management, accounting, logistics, etc. Like many other application systems, SAP R/3 is based on a commercial relational database system which is used to store all R/3 related data; e.g., a company's sales information. When installing R/3, it is possible to choose among several commercial systems; e.g., Adabas D, IBM UDB, Informix Adaptive Server, Microsoft SQL Server, or Oracle 8.

Obviously, very good performance is crucial for users of SAP R/3. In many companies, transactions of thousands of users must be processed concurrently by SAP R/3 and the underlying database system. In addition, the size of the database can become very large. Today, the largest SAP R/3 databases have about 1.5 terabytes; in a few years, these databases are likely to grow several times that size as a result of new R/3 releases and developments; e.g., component architecture or SAP's business information warehouse. Unfortunately, tuning the performance of an SAP R/3 system is very difficult because both the R/3 application system and the underlying database system must be tuned.

2 Overview of SAP R/3

SAP R/3 is the market leader for integrated business administration systems. It integrates all business processes of a company and provides modules for finance, human resources, material management, etc. SAP R/3 is based on a (second party) relational database system which serves as an integration platform for all components of SAP R/3. The database system manages the SAP database which stores all business data of a company (e.g., customer and supplier information, orders, . . .), all of SAP R/3-internal control data, an SAP R/3 data dictionary, and the code of all application programs. Virtually no data are stored outside this SAP database, thereby avoiding the use of a file system.

SAP R/3 [WHSH96, BEG96, Sch99, DHKK97] is based on a three-tier client/server-architecture:

1. The presentation layer. It provides a graphical user interface (GUI) usually running on PCs that are connected with the application servers via a local (LAN) or a wide-area network (WAN).

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2. The application layer. It comprises the business administration "know-how" of the system. It processes pre-defined and user-defined application programs such as OLTP and the implementation of decision support queries. Application servers are usually connected via a local area network (LAN) with the database server.

3. The database layer. It is implemented on top of a (second party) commercial database product that stores all data of the system, as described above.

2.1 Data Model and Schema of SAP R/3

SAP R/3 is a comprehensive and highly generic business application system that was designed for companies of various organizational structures and different lines of business (e.g., production, retailing, . . .). This genericity and comprehensiveness resulted in a very large company data model with over 13.000 database tables. To manage the meta data (e.g., types and interrelationships) of these tables, SAP R/3 maintains its own data dictionary which is (like all other data) stored in SAP's relational database and which can be used by SAP application programs.

2.2 ABAP/4

Applications of the SAP R/3 system are coded in the interpreted programming language ABAP/4 (**A**dvanced **B**usiness **A**pplication **P**rogramming Language) [MW97]. Except for a small kernel, actually the entire R/3 system is coded in ABAP/4. ABAP/4 is a so-called Fourth Generation Language (4GL) whose origins can be found in report/application generator languages. For this reason, ABAP/4 programs are often called reports. In the course of the R/3 evolution, ABAP/4 was augmented with procedural constructs in order to facilitate the coding of more complex business application programs. For example, ABAP/4 contains language constructs to program so-called "Dynpros" which are dialog programs with a graphical user interface including the logic for validating and processing user entries. Recently, SAP has extended its ABAP programming language with object-oriented features, sometimes referred to as ABAP Objects.

As sketched in Figure 1, ABAP/4 provides commands that allow to access the database via two different interfaces: *Native SQL* and *Open SQL*. The Native SQL interface can be addressed by so-called EXEC SQL commands. It allows the user to access the SAP database directly without using the SAP-internal data dictionary. The advantage is, that the database system-specific properties and services (e.g., non-standard SQL statements like optimizer hints or specialized operators like *cube*) can be fully exploited and additional overhead by SAP R/3 is avoided. However, using the Native SQL interface incurs some severe drawbacks: (1) The EXEC SQL commands may be database system-specific which renders non-portable ABAP/4 programs. (2) By circumventing the SAP-internal data dictionary, EXEC SQL commands cannot access encapsulated relations (containing encoded/clustered data that can only be interpreted using the transformations maintained in the data dictionary). (3) Native SQL reports are potentially unsafe because Native SQL directly reads database relations, and the implementor of a Native SQL report might overlook intrinsic business process interpretations which are otherwise carried out implicitly by SAP R/3's application programs; that is, bypassing SAP R/3's data dictionary requires expert knowledge about the rules and dependencies of the system.

Safe and portable ABAP/4 reports can be written by relying exclusively on ABAP/4's Open SQL commands. Consequently, with very few exceptions the entire R/3 system shipped by SAP is programmed using Open SQL.

2.3 Transaction Processing in SAP

SAP uses the term *Logical Unit of Work* (LUW) for transactions. Basically, an SAP LUW has the same ACID properties as database system transactions: an SAP LUW can span several dialog steps and an SAP LUW is either executed completely or not at all (i.e., atomicity). However, SAP LUWs are not mapped 1:1 to database transactions; rather a more complex SAP LUW may involve several database transactions. To synchronize LUWs

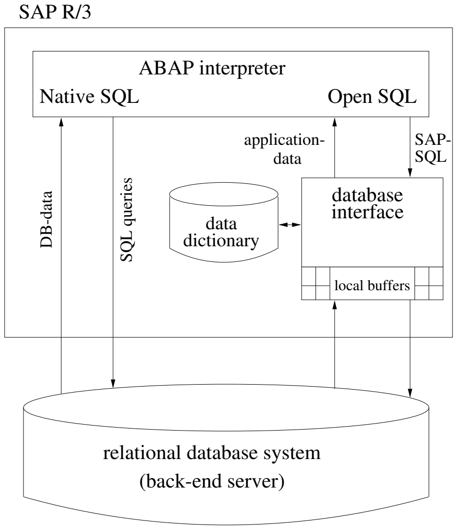


Figure 1: Database Interface of ABAP/4

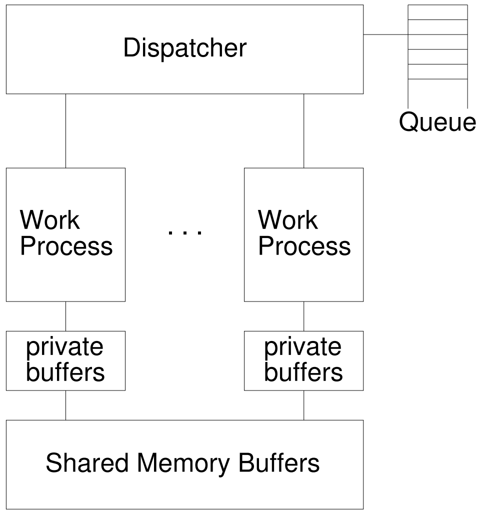


Figure 2: Architecture of the Application Server

SAP implemented its own locking scheme which is managed by a (centralized) *enqueue server* which runs in one of the application servers. Basically, SAP also implemented its own TP monitor consisting of a *message handler* and *request queue* in every application server. In comparison: PeopleSoft uses third-party TP monitors such as Tuxedo.

Figure 2 illustrates the processing of user dialog steps by the application server. When a user logs in, a message handler finds the application server with the smallest load (load balancing). This application server handles all of the requests of that user session. A user session consists of several transactions (LUWs), and every transaction consists of several dialog steps. Every application server has one dispatcher process and several work processes. The dispatcher queues requests until a work process is available to carry out the dialog step. For this, the relevant data is "rolled into" the private buffer and the ABAP program is interpreted. Thus, every user session is handled by a single application server, but dialog steps of the same LUW may be handled by different work processes. There is, however, an exception for transactions that involve large objects: They are assigned to exclusive work processes to avoid the cost of rolling data in and out.

3 Tuning SAP R/3 For Transaction Processing

3.1 Application Servers and Work Processes

Obviously, one crucial decision is to determine the hardware used for the application server machines and the database backend server. This decision impacts greatly how many concurrent users can be supported. The following is a brief overview of possible hardware configurations [BEG96, KKM98]:

1. *tiny* (1 user): the whole system is installed on a laptop; the golf club of one of SAP's founders, for example, is organized in this way. Such installations are particularly useful for teaching purposes.

2. *small* (10 users): application and database server run on one mid-range machine

3. *medium* (100 users): a few mid-range machines are used to run the application servers and a separate machine is used for the database server; the machines are connected by an Ethernet

4. *large* (1000 users): several machines are used for the application servers and a high-end (multi-processor) machine is used for the database server; the machines are connected by a very high-speed LAN.

Another decision concerns the number of work processes established at each application server. Clearly, the CPU of an application server machine will be under-utilized if too few work processes are established; on the other hand, many CPU cycles will be wasted due to context switches if too many work processes are established. Furthermore, the number of work processes impacts the use of main memory (Section 3.2). A good rule of thumb is to have one work process for five to ten users.

3.2 Main-memory Management

An SAP R/3 application server divides the available main memory into several segments. First of all, the memory is divided into *shared memory* which can be accessed by all working processes and into *local memory* for each individual working process. The shared memory is further subdivided into the *R/3 extended memory*, the *R/3 paging area*, and the *R/3 buffer/cache*. The local memory of a working process is subdivided into local *roll memory*, the *R/3 heap*, and memory for local data (e.g., application code). The size of each of these segments can (and must) be set by an R/3 system administrator. It should be noted that SAP R/3 main memory management is of course implemented on top of the virtual memory management of the operating system.

In order to demonstrate the importance of a right configuration, we will show how the roll memory, the R/3 extended memory, and the R/3 heap memory are used during transaction processing. A transaction is composed of several dialog steps; each dialog step may be processed by a different working process. To process a dialog step, a working process copies all the data of the transaction generated by previous dialog steps from the R/3 extended memory into its roll memory, carries out the dialog step, and copies the updated data back into the R/3 extended memory. If the data of a transaction grows larger than the size of the roll memory, then the working process carries out the dialog step directly in the R/3 extended memory; in this case, only pointers to the data are copied into the roll memory. As a result, the size of the roll memory determines whether work processes operate using private (local) or shared memory. In fact, starting with Release 4.0 SAP recommends to set the size of the roll memory to 1 in order to force the system to carry out all dialog steps directly in the R/3 extended memory because shared memory access has become cheap and a great deal of copying data can be saved this way.

In situations in which the R/3 extended memory is exhausted and cannot hold all data of all active transactions, the data of a transaction is copied into the R/3 heap. In the R/3 heap, data can only be read and updated by the same working process that wrote the data into the heap. As a result, all dialog steps of a transaction must be processed by the same work process once the data of the transaction has been written to the R/3 heap. Configuring the size of the R/3 extended memory, therefore, impacts the dispatching of dialog steps.

From this discussion it should have become clear that tuning the main memory allocation is very difficult for the R/3 application servers. (Buffer allocation for the database server is difficult, too, – for different reasons – and must be carried out in addition to the main memory management of the R/3 application servers.) A list of recommendations is given in [Sch99]. Furthermore, R/3 ships with a tool called *quick size* that helps R/3 administrators. Interestingly, Microsoft provides a special feature in the Windows NT operating system to help the administrator. The idea is to dynamically adapt the size of the R/3 extended memory so that the administrator need not set this parameter explicitly and need not consider the tradeoffs between executing dialog steps in the R/3 extended memory and the R/3 heap. This initiative is called *zero main memory administration.*

3.3 Caching

In order to reduce the load on the database server, a potential bottleneck, all R/3 application servers make use of caching. Any kind of data including application code, constraints, schema information, and user data such as *Customer* information, can be cached in the buffers of an application server. Hit rates of 90% and even higher are not unusual for OLTP workloads in practice. The decision of what and how to cache can again be made by the R/3 system administrator. For example, the administrator can determine that *Sales* information which is bulky and frequently updated should not be cached, whereas information about *Regions* should be cached because such data is small and rarely updated. The administrator can also determine whether whole tables or individual records of a table should be cached (i.e., the granularity of caching). For all (13004) tables that are part of the *standard vanilla* R/3 schema, R/3 provides default settings. For new, user-defined tables, no caching is the default.

4 Query Processing in SAP

4.1 SAP's Query Language Features

As shown in Figure 1, the SAP R/3 system provides the two query interfaces *Native SQL* and *Open SQL*. Since almost all built-in applications access the database via the *Open SQL* interface we will concentrate on its query facilities. The two basic query constructs of the Open SQL interface are the SELECT ... ENDSELECT and the SELECT SINGLE statements (the latter requiring a WHERE clause on *unique* fields of a table so that at most one tuple qualifies and is returned for further processing).

Up to recent R/3 releases, both SELECT commands were restricted to a single SAP table or view. That is, unless a (join-)view was defined, it was not possible to implicitly describe a join, as is possible in SQL by referencing several relations in the FROM-clause. Join views could only be formulated over transparent tables (i.e., non-encoded tables that can be interpreted by the RDBMS without the SAP data dictionary) and only along primary key/foreign key relationships.

To evaluate a general join within the Open SQL interface, the implementor had to code an ABAP/4 program with nested SELECT...ENDSELECT or SELECT SINGLE loops. In essence, this corresponds to an (index) nested loops join with the additional overhead of "crossing" the interface between database system and ABAP/4 program for every tuple of the outer relation in order to find the matching tuples of the inner relation.

Furthermore, groupings and aggregations could not be incorporated into the Open SQL SELECT statements of older SAP R/3 releases. As a consequence, all groupings and aggregations had to be performed by the SAP system, thereby possibly transferring huge amounts of data from the RDBMS to the SAP application server.

Recently, SAP has incorporated joins, groupings, and simple aggregations into the Open SQL SELECT statements. These operations can now be delegated to the underlying RDBMS and benefit from the RDBMS's advanced join and groupby algorithms. However, one has to keep in mind that it requires reprogramming the many (thousands of) built-in applications before this takes effect. Delegating these operations to the RDBMS will, of course, "strain" the underlying database server (in particular the query optimizer and the query engine) even more than current SAP installations already do.

4.2 SAP Query Optimization Features

To optimize query processing, SAP R/3 implements two techniques which take effect if the Open SQL interface is used: (1) *Cursor caching* which reduces the overhead of calls to the RDBMS by using the same cursor for, say, all the queries that retrieve the matching tuples of the inner relation in a nested SELECT statement. Cursor caching is possible because most database systems allow parameterized queries and provide a cursor REOPEN command in their API. (2) *Caching* data in SAP R/3 application servers in order to avoid calls to the RDBMS altogether (cf. Figure 1). For caching, the typical tradeoffs between read and update frequency apply; in addition, SAP R/3 does not fully guarantee cache coherency in a distributed environment as updates are only propagated periodically.

Cursor Caching In order to enable cursor caching, SAP is transforming queries containing literals (constants) into parameterized queries—in anticipation that the same query with a different parameter will be issued again in the near future. Due to this translation, the optimizer of the RDBMS cannot estimate the selectivity of the predicate of the translated query and thus *blindly* generates a plan. In some cases, the optimizer chooses bad access plans—due to the fact, that the selectivity could not be estimated correctly.

Data Caching If data is cached in the application server the Open SQL interface is exploiting this. SAP differentiates between two caching levels for a particular relation: complete table caching and individual tuple caching. As pointed out before, the SAP system "comes with" pre-configured caching rules for all its built-in tables which may, however, need to be adapted for particular workloads.

In addition to the implicit caching, ABAP/4 allows the materialization of query results in internal (i.e., temporary) tables in order to use this data for further processing. For example, it is possible to materialize the inner relation of an Open SQL nested-loops join report and avoid repeated calls to the RDBMS this way. Using such a materialized table, SAP provides a language construct called

$$\ldots \text{FOR ALL ENTRIES IN } \langle \text{materialized table} \rangle \ldots$$

to obtain the matching tuples of another table—i.e., to obtain the semijoin result. For this purpose, the SAP query processor generates a corresponding WHERE clause containing a disjunct for every tuple of the materialized table. This way, users can implement their own join routines; for example, they could implement a semijoin-supported sort/merge join executed in the application server—thereby taking load off the database system.

4.3 Tuning SAP Queries

SAP provides tracing tools for analyzing potential performance bottlenecks resulting from poor query evaluation plans. Applications with poor query performance could be reprogrammed using the advanced query features that allow to delegate complex operations to the underlying database server. If the RDBMS server is particularly loaded the opposite may be necessary: reprogram an application such that more data processing is performed in the application server, e.g., sorts, groupings, etc. In this case, intermediate results should be materialized in the application server and these snapshots should be reused whenever possible. Database-specific features, such as user hints and specialized (non-standard) operators, can only be exploited if an application is reprogrammed such that it accesses the database via the Native-SQL interface. The SAP built-in applications usually don't use this interface because, among other reasons, it violates portability between database vendors. Query processing can be supported by defining new indexes on the underlying database tables. This may, however, hurt the "mission-critical" OLTP performance. Database statistics should, of course, be updated periodically to ensure that the query optimizer has precise numbers. However, we have experienced that SAP users suppressed certain statistics in order to "trick" the optimizer into generating a desired plan that it wouldn't choose otherwise. Note that from the Open SQL interface it is not possible to pass *optimizer hints* to the RDBMS.

4.4 BW: SAP's Data Warehouse

Optimizing the SAP system for query performance may be hurting the OLTP performance. In most SAP installations, OLTP performance is more critical than OLAP performance because the OLTP operations are needed for the companies' operational business. Therefore, SAP recommends that companies with high OLAP requirements use its recently developped data warehouse product, called Business Information Warehouse (BW), in conjunction with the R/3 OLTP system. Using the BW the OLAP-relevant data is extracted from the operational R/3 OLTP system and stored in pre-defined star schemes [CD97], called *InfoCubes* in SAP terminology.

Like the R/3 system, SAP's business information warehouse BW runs on a variety of third party RDBMSs. The BW provides open Business Application Programming Interfaces (BAPIs) for data loading (even from non-SAP systems besides R/3 systems) and for OLAP processing (to facilitate the use of third party visualization tools). The SAP BW provides a pre-configured meta data repository consisting of InfoCube catalog, report catalog, information source catalog, etc. SAP's BW is shipped with many pre-defined InfoCubes for common business applications, e.g., market segment analyses, profitability analyses, stock inventory analyses, corporate indicator systems. To obtain the data (and subsequently the incremental updates) from R/3 OLTP systems predefined extraction routines are delivered.

5 Monitoring and Performance Evaluation

In the previous two sections we described different options for transaction and query processing in SAP R/3. In this section, we describe tools that can be used to monitor and assess the performance of SAP R/3 installations.

5.1 Monitoring

SAP R/3 ships with a very powerful monitoring tool. This tool measures, for instance, the lengths of the queues at every application server, the cache hit ratio, the running time of database operations and of ABAP/4 operations, the number of transactions that are committed and rolled back, the response time of dialog steps, and the CPU, disk, memory, and network utilization. All these performance statistics are stored in the relational database. In addition, alerters inform system administrators if the system performs poorly. Furthermore, SAP offers a special *early watch* service. If a company buys this service, then SAP specialists periodically assess the performance statistics of the company's R/3 installation and provide suggestions for performance improvements [BEG96].

5.2 SSQJ Tool

In order to detect performance problems, SAP has also developped a tool called SSQJ. This tool stores the code and running times of queries that have caused performance problems in at least one R/3 installation in the past. The queries recorded by SSQJ include simple select queries involving different kinds of indices as well as complex multi-way join queries; for example, the queries of the TPC-D benchmark for SAP R/3 are maintained by SSQJ (Section 5.3). Currently, this tool is mostly used by database vendors in order to test new releases. Before a new release of an RDBMS is shipped, the vendor runs all the queries stored in SSQJ and SSQJ indicates which queries showed good or bad performance compared to the running times obtained using an older release. SSQJ could also be used to assess the R/3 installation of a customer; that is, before working with the new R/3 installation, a customer could run all queries stored in SSQJ and see whether any queries show particularly poor performance.

5.3 Benchmarks

A number of benchmarks have been proposed for SAP R/3. The main characteristic of these benchmarks is that the whole system is tested; i.e., rather than testing the RDBMS in isolation, the benchmarks test the execution of typical R/3 operations (dialog steps and queries) that involve processing by the application and database servers. The most prominent benchmark is the SD benchmark for measuring the performance of processing sales and distribution transactions. Many hardware and database vendors have published results of this benchmark. SAP collects and certifies these benchmark results and publishes them, e.g., on the Web. In previous work, we adapted the TPC-D benchmark [TPC95] for SAP R/3 and published results in [DHKK97]. As stated earlier, the queries of the TPC-D benchmark are also part of SAP's SSQJ tool.

6 Summary

In this paper we gave an overview of various performance aspects of SAP R/3. After overviewing SAP R/3's three-tier architecture, we described tuning options and experiences for transaction processing (OLTP) and query processing (OLAP) and briefly surveyed the monitoring and benchmarking tools for an R/3 system.

References

[BEG96] R. Buck-Emden and J. Galimow. *SAP R/3 System, A Client/Server Technology*. Addison-Wesley, Reading, MA, USA, 1996.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, Mar 1997.

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 123–134, Tucson, AZ, USA, May 1997.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998. http://www.db.fmi.uni-passau.de/publications/tutorials/.

[MW97] B. Matzke and A. Weinland. *ABAP/4 - Programming the SAP R/3 System*. Addison-Wesley, Reading, MA, USA, 1997.

[Sch99] Thomas Schneider. *SAP R/3-Performanceoptimierung*. Addison-Wesley, Reading, MA, USA, 1999.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995. `http://www.tpc.org/`.

[WHSH96] L. Will, C. Hienger, F. Straßenburg, and R. Himmer. *R/3-Administration*. Addison-Wesley, Reading, MA, USA, 1996.

Tuning Time Series Queries in Finance: case studies and recommendations

Dennis Shasha
Courant Institute of Mathematical Sciences
Department of Computer Science
New York University
shasha@cs.nyu.edu
http://cs.nyu.edu/cs/faculty/shasha/index.html

Abstract

Financial databases are different from regular ones: they involve more money. Furthermore,

- *Database speed and reliability can make the difference between prosperity and ruin.*
- *Money for information systems (and for salaries) is no object.*
- *Data must be accessible from many points on the globe with subsecond response.*
- *Many calculations require computations on vectors, usually time series.*

In a previous tutorial article [10], I presented several case studies having to do with reliability, interoperability, data coherency, and tuning. In this complementary paper, I discuss examples of challenges that follow from the time series nature of financial data. I also suggest a way to attack this problem.

1 Cultural Context

As a frequent consultant to Wall Street, I am exposed to a technical atmosphere that is unrecognizable to most researchers and system programmers: time is short, tempers are shorter, and bonuses influence truth.

- New financial products are introduced within days of their conception.
- Traders routinely yell obscenities into their phones. Some break those same phones by banging the handset against its base in fury.[1]
- Technical people often bear the brunt of this wrath and usually must do so with good humor since their yearly bonuses depend largely on what the traders say about them.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] This behavior goes home with some. The director of technology of a major investment bank was once handcuffed to a fence by the police so he would stop harassing the umpire during his son's little league game.

In such an environment, planning horizons rarely exceed six months and technical purchase decisions often depend on what some trader used at his or (rarely) her last job. The result is a babel of systems that are held together by the sweat of highly paid programmers and administrators.

Like the traders, these technical workers change jobs frequently (once every two years or so) and so feel a need to tell a good story about the latest technology in the press. One group leader told me in 1994 that he used C++ in his group, partly because he knew it would attract people who wanted to put knowledge of that language on their resume *for their next job*. Replace C++ by Java Beans and you will find a similar story as of 1999.

Fortunately for the industry, the technical people are quite smart. So, while they follow fads, the fads usually answer a real need. Sybase enjoyed a huge success on Wall Street in the late 1980s, for example, because the system had light weight threads and triggers. I still use the word ‘fad’ however because technology selection is not done in a vacuum on Wall Street. Everyone knows what everybody else is doing. The more popular a technology becomes, the more people want to use it because it will give them job mobility. Conversely, vastly superior technology may have a hard time capturing interest until, well, it catches on. I will discuss a needed superior technology towards the end of this article.

2 Four Ways of Working with Time

Financial applications treat time in many ways, but the following four queries illustrate the main variants.

- Compute the value of my portfolio of bonds based on future payments and projected interest rates.
- Find correlations or other statistics among different stock prices over time or between option premiums and stock prices in order to help make better trading decisions.
- Find all events that are likely to precede a rise in the price of a security, regardless of precise timing constraints.
- Determine the data available at time X to trader T when he made the decision to buy Y for his own account.

Let us discuss each variant and its generalizations in turn.

3 Present Value

Contrary to what one reads in the popular press, bonds are far more important than stocks for most investment banks. Companies, municipalities, and nations issue debt and the markets buy them. Each debt note has an associated payment stream occurring at regular intervals, e.g. each half-year. These debt notes are then swapped for cash (I’ll borrow money and give some bonds I’m holding as collateral), tranched (sliced into payment streams of different risks), and otherwise manipulated. The basic computational issue is to value these payment streams.

But how should one value ten yearly one million dollar payments? The basic principle is to normalize all payments to their *present value*. The present value of a payment of x dollars in i years assuming an interest rate of r is $y = (x/((1+r)^i))$ dollars. (This definition is justified by the fact that putting y dollars in the bank at interest r would yield the same amount of money at the maturity of the bond as taking each of the payments and putting them in the bank at interest r.)

So, the basic problem is to compute expressions of the form $(1+r)^i$ for many different values of i. Since the value of r is never known precisely, many different r’s are tried in order to calculate the statistics of the value of a bond.

Notice that each such calculation involves computing the log of $1+r$, then multiplying by i and then taking the inverse log of the result. Since many expressions have the same value of r but differ on i, we’d like to avoid doing the log more than once for a given r.

Relational systems on Wall Street typically store each payment in the form: (amount, date, interest). The present value of each row is then computed independently of all others either through a trigger or an object-relational extension. This strategy misses the significant log optimization described above and the resulting performance is punishingly slow. For this reason, many applications take the data out of the relational database and perform this calculation in some other way.

Avoiding this exodous from the database requires the object-relational system to share computational work even among external functions. That is an undecidable task in general since the external functions are written in a Turing-complete language. Another possibility is to call external functions on bigger entities (e.g. arrays of payments and dates) so the function itself can do the necessary computational sharing.[2] We will explore this second possibility later.

3.1 Regular Time Series and Statistics

Trends are a popular abstraction for many average stock investors. Knowing that "the market is going up" is a powerful motivation to buy and conversely. Unfortunately, history and countless investment books suggest that this does not yield good results for two reasons. First, it is not clear when a trend has started. Second, trend trading ignores the different properties of different companies.

A far better strategy is to make use of correlations between different companies having similar properties. Take two similar banks B1 and B2. In the absence of differentiating external news (e.g. a default of a creditee of B1 alone), the prices of the two banks should track one another closely. So, if B1 starts going up while B2 does not, a "pairs trader" will sell B1 and buy B2.

In the FinTime benchmark[3] Kaippallimalil J. Jacob of Morgan Stanley and I have tried to capture the typical queries correlation and regression analysis requires.

The model consists of a few relational tables that typically contain infrequently changing information about stocks and a number of time-series tables about prices and stock splits.

Here are three typical queries:

1. Get the closing price of a set of 10 stocks for a 10-year period and group into weekly, monthly and yearly aggregates. For each aggregate period determine the low, high and average closing price value.

2. Find the 21-day and 5-day moving average price for a specified list of 1000 stocks during a 6-month period. (Use split adjusted prices).

3. Find the pair-wise coefficients of correlation in a set of 10 securities for a 2 year period. Sort the securities by the coefficient of correlation, indicating the pair of securities corresponding to that row.

Here are some features that relational systems require to answer such queries:

1. the special treatment of the semantics of time (e.g. the discovery of weekly boundaries and the ability to aggregate over such boundaries)

2. the ability to treat events in a time-ordered sequence (e.g. to perform moving averages)

3. the ability to perform time-based statistical operations on multiple sequences.

[2] It should go without saying that the overhead of each call should be low. Unfortunately, null external function calls appear to require thousands of instructions in some large well-known database management systems.

4 Irregular Time Series and Frequency Counting

Hurricanes, oil spills, earthquakes, and other natural and manmade disasters can influence the stock market in many ways. These events are not regular (one hopes) and so moving averages and complex signal processing techniques are inapplicable. On the other hand, the fusion of disaster events with price data may make the following queries meaningful and useful:

- (Specific) Which disaster event type affects the price of insurance company X?
- (General) Which event sequences affect which companies within 5 days?

We have used the example of disasters, but similar queries apply to large purchases prior to merger announcements and other suspected insider activity.[3] One way to conceptualize such problems is to discover patterns in timed event sequences[7]. Relational systems would model this by attaching a time attribute to events, but the discovery of sequence of events would require one or more joins. Most real-life systems would take some data from the database and do the real work outside.

5 Bitemporality

It is often necessary to know what someone knew at a certain date. For example, did trader Jeff believe that a merger was to take place between companies X and Y as of time Z when he bought stock in X? This might be grounds for legal concern. What was the best public guess as to the third quarter profits of company X at time Z? This might explain why trader Carol took such a large position in the company at that time.

Such queries are called bitemporal[4, 11, 12] because they involve two times: the time of the event in question (the merger or the third quarter profits in our examples) and the time of the query (time Z in both cases). The point is that our knowledge of the event in question may be better now, but what we are interested in is what was known at some time in the past based on the transactions that had committed up to that point.

Whereas most queries having to do with compliance or personnel review occur rarely enough to be irrelevant for performance tuning, the same does not hold for other queries involving time. For example, consider a portfolio accounting application with queries like this:

- As of last June, what was the anticipated revenue stream of this portfolio over the next ten years.

Since a portfolio consists of many securities, this involves combining the streams from many different securities after determining which securities were in the database last June.

As Snodgrass and Jensen show in their book[12], such queries can be implemented in relational databases, but may require pages of code and may in fact be very slow. Indeed, the bank officer who described the portfolio accounting application put the problem this way: “The functionality excites the users but makes the [standard relational] server glow a dull cherry red...”

At least two solutions are possible to speed up such actions:

- Implement the bitemporal model inside the database system. The risk is that this may create overly specialized implementations.
- Treat the data as ordered and see where that leads. Please read on.

[3] Of course, such event sequence queries appear elsewhere, e.g. in web data mining, A typical problem there is to infer the probability that a person will buy an item given that he has visited some sequence of pages. Sequence plays a role but not necessarily wall clock time.

6 Recommendations

My recommendations follow from a simple observation: *Sequences are more powerful than sets.*

Membership, intersection, union, and difference can be easily applied to sequences. On the other hand, order, nth best, moving averages, and other statistics require a tortured and inefficient encoding on sets, but are linear time on sequences.

Now let us consider two ways in which sequences might extend the relational model:

- Tables might be maintained in a specific order, as **arrables** (array tables). Selections, joins, and projections can all carry their multiset-theoretic meaning to arrables (SQL is a multiset language when all is said and done), but new operations can be defined that exploit order. These new operations may include nth best operations, moving averages on certain columns, correlations between columns and so on.

  Ordering tables into arrables has certain disadvantages however. For example, an arrable cannot be stored using hashing. Ordering does not however preclude a secondary hash index. The extra overhead for ordering is therefore one more access in most cases[9]. This extra access may be significant in disk-resident data.

- Another approach to order is to break a relational commandment. In a normal SQL statement group by statement, e.g.

  ```
  select id, avg(price)
  from stock
  group by id
  ```

  each group corresponds to one id and there are one or more scalar aggregates associated with that group (here, the average price in the group). Suppose instead that that the following were possible:

  ```
  select id, price, date
  from stock
  group by id
  ```

  The net result here would be to cause a *vector* of prices and dates to be associated with each stock. Now that we have vectors, we can apply vector style operations that may be offered by a system or that an application program may write. For example, we may sort the prices in each group by date and then construct moving averages. Regrettably, allowing vector elements in rows violates first normal form. Maybe this is ok.

These two ways of incorporating order are separable. If introduced together, then the stock arrable might be ordered by date and then the query becomes:

```
select id, price
from trade
group by id
```

Prices will then be ordered by date within each date, provided the grouping preserves the sort order of the trade arrable. Even better, we can construct the 5 day moving average in one shot, given a function that computes the moving average of a sequence, say "movavg" and takes a time parameter:

```
select id, movavg[5,price]
from trade
group by id
```

Several systems treat sequences as first class objects including Fame[1], SAS[8], S-Plus[6], and KDB[2]. KDB supports a language called KSQL which is a semantic extension to SQL supporting order and time[4] in a few ways:[5]

- Tables are ordered, often but not necessarily by time.
- Special functions are offered to exploit this order, e.g. first, last, nth, and moving aggregate statistics. For example, if the trade table is ordered in descending order by price, then the following produces the 17th highest price of each stock

  ```
  select last 17 first price
   from trade
   group by stock
  ```

- Other functions can be added that apply to scalars, vectors or entire tables by the user, just as in conventional object-relational systems. For example, suppose we want to compute the dot product of prices per stock offset by 10 days. The dot product function is first defined (in this case, in the language K) and then used

  ```
  dotprod:{[x;y] +/(x*y)}
  select dotprod[(10 drop price), (-10 drop price)]
   from trade
   group by stock
  ```

- Date is a data type and the system is aware of operations on that type. In this example, the trade table is grouped by date and month.

  ```
  select volume, price
   from trade
   group by date.month
  ```

Looking back on the time series queries we face in finance, let us see what order does for us:

- Computing the present value and similar operations are natural on sequences. If the external functions to compute these functions are vector-aware, then optimizations, such as performing the logarithm once instead of many times, can easily be incorporated.
- Correlations, moving averages, grouping by time periods such as weeks and months, and special purpose interpolations require an awareness of date as a special data type and the ability to add arbitrary functions in an object-relational style. Event data mining requires a subset of this functionality.
- Bitemporality remains a challenge, but less of one. Because bitemporal queries involve unchanging historical data, one can afford to store redundant data, e.g. the entire set of portfolio entries each time a portfolio changes. So, the query that makes a current server glow a cherry red, can perhaps become red hot.

7 Conclusion

Time as used in finance presents a performance challenge to conventional databases. After reviewing the main query variants, this paper recommends a simple but radical change: embrace order.

[4] KSQL also shortens some SQL statements, e.g. "group by" becomes "by" and some joins are implicit. I have rendered the examples without those shortcuts here.

[5] SRQL shares many of these ideas and adds a few other twists[5]

8 Acknowledgments

Many thanks to Marc Donner for introducing me to financial databases and to Arthur Whitney for profound scientific insight.

References

[1] Information about Fame can be found at www.fame.com.

[2] KDB and KSQL can be downloaded on a trial basis from www.kx.com.

[3] K. Jacob and D. Shasha "FinTime, a financial time series benchmark" http://cs.nyu.edu/cs/faculty/shasha/fintime.html

[4] Christian S. Jensen and Richard T. Snodgrass "Semantics of Time-Varying Information *Information Systems, 21(4)* pp. 311-352 (1996)

[5] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Keven S. Beyer, and Muralidhar Krishnaprasad, "SRQL: sorted relational query language" *SSDBM 98*

[6] B.D. Ripley and W.N. Venables, *Modern Applied Statistics with S-Plus* Springer Verlag (1994) http://www.stats.ox.ac.uk/ ripley/ has a lot of useful functions.

[7] John F. Roddick and Myra Spiliopoulou. "A bibliography of temporal, spatial and spatio-temporal data mining research." SIGKDD Newsletter, 1999.

[8] Information about SAS (the statistical package) can be found at www.sas.com

[9] Dennis Shasha *Database Tuning — a principled approach*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[10] Dennis Shasha "Lessons from Wall Street: case studies in database tuning, configuration, and replication" *Proc. 1997 ACM Sigmod Conf.*, pp. 498-501. Slides are available from the research link of Shasha's web page.

[11] Richard T. Snodgrass (editor), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Kifer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada, *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, 1995.

[12] Richard T. Snodgrass and Christian S. Jensen *Developing Time-Oriented Database Applications* Morgan-Kaufman, 1999.

25th International Conference on Very Large Databases

Edinburgh - Scotland - UK

7th - 10th September 1999

Invitation to attend VLDB'99

VLDB'99, Edinburgh, UK marks the 25th anniversary of the leading international conference on databases. VLDB'87 was also held in UK (at Brighton) and was one of the largest VLDB conferences and produced some of the most cited VLDB papers. We expect VLDB'99 to be no less successful. For database technology practitioners and theorists VLDB'99 is not to be missed.

The conference has an excellent programme of Tutorials, Panels and Invited Talks, and the selection of papers to be presented from the many submitted will ensure - as usual for VLDB - the highest quality. A programme of demonstrations ensures awareness of the leading edge R&D and the exhibition provides an opportunity to see and discuss products. Furthermore, there are workshops and events organised before and after the conference, so making the travel to attend VLDB'99 even more worthwhile.

VLDB'99 is being held in Edinburgh, a city world-renowned for its beauty, style, culture and nightlife. VLDB'99 is being held just after the Edinburgh Festival - a truly amazing collection of art, drama, music, revue and cinema. Edinburgh can be reached easily - see the VLDB99 website for details. VLDB'99 attendees will find themselves not only at an interesting spatial location; the temporal location is also important as Scotland gains some level of independence from the rest of the UK through devolution. This is the time to visit Edinburgh, Scotland's capital city.

The VLDB Social programme has also been carefully arranged to provide attendees not only with a taste of Edinburgh and Scotland, but also those opportunities for discussion and conversation so necessary to the advancement of database technology and to the maintenance and further building of the community.

So, on behalf of the Programme Committee led by Malcolm Atkinson and the Organising Committee led by Jessie Kennedy it is my pleasure to invite you to attend VLDB'99. Registration details, and details of travel, accommodation, programme and social programme are to be found in this Call for Attendance or on the VLDB'99 Website http://www.dcs.napier.ac.uk/~vldb99/

I look forward to meeting you in Edinburgh

Keith G Jeffery
General Conference Chair VLDB'99

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

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