Database Self-Management: Taming the Monster

Mohammed Abouzour *Ivan T. Bowman* *Peter Bumbulis* *David E. DeHaan*
Anil K. Goel *Anisoara Nica* *G. N. Paulley* *John Smirnios*
Sybase, An SAP Company

Abstract

We describe our experience in the development of self-managing technologies in Sybase SQL *Anywhere, a full-featured relational database system that is commonly embedded with applications developed by Independent Software Vendors (*ISV*s). We illustrate how* SQL *Anywhere's embeddability features work in concert to provide a robust data management solution in zero-administration environments.*

1 Introduction

Sybase SQL Anywhere is a full-function, ISO SQL standard-compliant relational database server designed to work in a variety of frontline environments, from traditional server-class back-office installations to handheld devices running the Windows CE operating system. SQL Anywhere includes features common to enterprise-class database management systems, such as intra-query parallelism, materialized views, OLAP functionality, stored procedures, triggers, full-text search, and hot failover. SQL Anywhere supports a variety of 32- and 64-bit hardware platforms, including Microsoft Windows; various flavours of UNIX including Linux, Solaris, AIX, HP-UX, and Apple OS/X; and Windows CE. However, the strength of SQL Anywhere is in its self-managing technologies, which allows the server to offer SQL data management, query processing, and synchronization capabilities in zero-administration environments.

In this paper, we describe our experience with some of the self-managing technologies present in SQL Anywhere. These self-managing technologies are fundamental components of the server, not merely administrative add-ons that assist a database administrator in configuring the server's operation—since in embedded environments there is often no trained administrator to do so. These technologies work in concert to offer the level of self-management and adaptiveness that embedded application software requires. It is, in our view, impossible to achieve effective self-management by considering these technologies in isolation.

2 Query Optimization

Certainly one of the most significant areas within the SQL Anywhere server that concerns self-managing behaviour is the SQL Anywhere query optimizer [1–3]. A significant portion of SQL Anywhere revenue stems from Independent Software Vendors (ISV's) who embed SQL Anywhere with their applications, some of which are deployed on thousands, and in some cases millions, of computers. These types of deployments typically share the following characteristics:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- There is no DBA available to 'tune' the application, to run workload analysis tools, perform capacity planning analysis, or monitor the system.
- The database workload is mixed, with both OLTP-like transactions and reporting queries mixed together. The nature of embedded applications renders workload partitioning, or sharding, moot.
- There is little correlation between the size of the database instance and the size of the schema, or the complexity of the application's queries. SQL queries with a join degree of 15-to-20 are routine. SQL queries with 50 or more total quantifiers (tables, views, or derived tables) are common enough to be unsurprising.
- In turn, there is often little correlation between a query's complexity and its execution time. In some cases, even complex queries can execute very quickly, particularly over small instances. In such cases, the optimization process must be very efficient. For these queries, the optimizer should not spend ten seconds (for example) optimizing a query that will run in under one second unless this time can be amortized over many executions.
- The database server executes concurrently with other applications on the same computer. Hence the server must co-operate with these applications for limited resources, particularly memory.
- The system hardware, operating system platform, and application workload characteristics, including peak-to-average ratios, may vary from installation to installation, making *a priori* application tuning unhelpful.

Moreover, a significant trend in application development over the past several years is the use of Object-Relational Mapping tools, such as Hibernate and NHibernate, and complex application development environments such as Microsoft Entity Framework, that map a relational schema into object-oriented abstractions within the application [4]. As a result, these frameworks can generate a mix of both simple and complex queries whose construction is largely outside the control of the application developer. Hence the ability to syntactically annotate SQL statements with tuning parameters is largely absent with the use of these frameworks.

As a result of the above constraints, we 'tamed the monster' by engineering the SQL Anywhere optimizer to be relatively inexpensive—in terms of both memory and CPU consumption—so that each SQL request can be optimized while taking into account the server context at that particular moment [5]. Server context includes the amount of available query memory for query execution, the number of available worker threads for intra-query parallelism, and the approximate contents of the server's buffer pool which directly influences I/O cost. One of SQL Anywhere's enumeration methods is a branch-and-bound algorithm [1], which is cheap due to careful engineering to minimize memory allocation and aggressively prune sub-optimal execution plans.

Over time, and due in part to the ever-increasing sophistication of SQL Anywhere applications—particularly those that involve synchronization—along with the ever-expanding SQL language and the explosion in the use of Object-Relational mapping tools, SQL Anywhere's query optimizer has become increasingly more sophisticated—and self-adapting. Included in the optimization process are sophisticated static and cost-based query rewrite optimizations [6, 7], including those for optimizing queries over materialized views [3]; the detailed analysis of a wide range of physical query execution operators; sophisticated selectivity and cardinality estimation; and an exploration of a space of possible plans that include parallelization. Our server software has been spectacularly successful at meeting the needs of our customers, whether they deploy SQL Anywhere server software on 64-bit installations of Windows 2008 Server, or the identical server software on Windows CE handheld devices.

Experience. However, despite the fact that the SQL Anywhere optimizer is relatively cheap, with this greater sophistication comes longer code paths, and consequently greater overhead, with each new SQL Anywhere release. `OPEN` time for a request is the critical measure for many applications, which includes the time to

construct the statement's necessary abstractions within the server in addition to optimization time *per se.* As with other commercial relational database systems, we too have tried to mitigate this additional overhead, in largely two ways. The first technique is execution plan caching, which amortizes the optimization cost over multiple invocations of the same SQL statement. The second is 'bypass' optimization for 'simple' SQL statements where the best execution strategy can be determined heuristically.

These mitigation techniques have been difficult to 'get right' in the sense that errors in selecting which plans to cache (or which cached plans to abandon), or precisely which queries will be optimized heuristically, lead to inconsistencies in query execution times. This inconsistency, coupled with a cost model that takes server context into account, yields another 'monster' that we term the 'dancing optimizer', named after the stereotypical dancing Russian circus bear.

We are actively researching new optimization techniques to tame this monster. The usage patterns of SQL Anywhere described above dictate that the optimization process continue to adapt to available resources, query complexity, and estimated query cost. Join enumeration algorithms can be extremely expensive for complex queries—for example, the exhaustive enumeration of the space of bushy trees—regardless of how small the estimated runtime is. Moreover, the memory consumption of these algorithms may be prohibitive. One important lesson learned from designing the SQL Anywhere optimizer is that there is no ideal join enumeration algorithm which will perform well—that is, generate good-quality access plans in a CPU- and memory-efficient way—for all types of queries, or available system resources. A robust design is to have a set of join enumeration algorithms to choose from when optimizing a particular query, based on the available system resources, query complexity, and its estimated runtime [2]. Query complexity, for example, can be used to predict the memory needed for the memoization process as well as the CPU required for enumeration. Efficient cost-based and heuristic pruning should be employed to adjust the optimization time to the expected query cost.

In addition, we are researching diagnostic tools with which to support the analysis of performance issues in the field. One novel capability of the SQL Anywhere optimizer is to log the set of *rejected* plans for a given SQL statement, so that these may be compared with the plan that was eventually chosen [8]. We are also continuing to research techniques that will permit the diagnosis of performance issues, due to sub-optimal execution plans chosen by the optimizer, when the customer's database is unavailable (for example, due to regulatory restrictions). We have augmented the execution plan information output by the query optimizer to contain information ranging from the cost-model computations to the cost-based and heuristic decisions made by the optimizer during join enumeration. These traces of the exploration of the search space can be analyzed in isolation, without the customer database, and also can be used to generate synthetic data to simulate the original database instance.

3 Self-managing Statistics

SQL Anywhere has used query feedback techniques since 1992 to automatically gather statistics during query processing. Rather than require explicit generation of statistics, by scanning or sampling persistent data, the server automatically collects statistics as part of query execution. More recent work by other researchers [9–13] has also exploited this notion of collecting statistics as a by-product of query execution.

In its early releases, SQL Anywhere computed frequent-value statistics from the evaluation of equality and `IS NULL` predicates, and stored these statistics persistently in the database for use in the optimization of subsequent queries. Later, support was added for inequality and `LIKE` conditions. An underlying assumption of this model is that the data distribution is skewed; values that do not appear as a frequent-value statistic are assumed to be in the 'tail' of the distribution.

Today, SQL Anywhere uses a variety of techniques to gather statistics and maintain them automatically. These include index statistics, index probing, analysis of referential integrity constraints, three types of single-column self-managing histograms, and join histograms.

Histogram implementation. SQL Anywhere histograms are equi-depth histograms whose number of buckets expand and contract dynamically as column distribution changes are detected. As is typical, we use the *uniform distribution assumption* when interpolating within a bucket. SQL Anywhere histograms combine traditional buckets with frequent-value statistics, known as 'singleton buckets'. A value that constitutes at least 1% or 'top N' of the values is saved as a singleton bucket. The number of singletons retained in any histogram depends on the size of the table and the column's distribution, but lies in the range [0,100]. An additional metric, which we term *histogram density,* is maintained for each histogram. Histogram density is an estimate of the average selectivity of all values in the column that are not covered by singleton buckets. In other words, the histogram density is the average selectivity of individual values in the tail of the distribution. Thus density gives an approximate selectivity estimate for an equality predicate whose value falls within a non-singleton bucket. An advantage of our definition of histogram density is that it does not reflect the high-frequency outliers in the data distribution, so a single average value will serve as an accurate estimate for many data distributions.

For efficiency and simplicity, the same infrastructure is used for all data types except longer string and binary data. An order-preserving hash function, whose range is a double-precision floating point value, is used to derive the bucket boundaries on these data types. For longer string and binary data types, SQL Anywhere uses a different infrastructure that dynamically maintains a list of observed predicates and their selectivities. In addition, not only are buckets created for entire string values, but also for `LIKE` patterns intended to match a 'word' somewhere in the string.

Statistics collection during query processing. Histograms are created automatically when data is loaded into the database using a `LOAD TABLE` statement, when an index is created, or when an explicit `CREATE STATISTICS` statement is executed. A modified version of Greenwald's algorithm [14] is used to create the cumulative distribution function for each column. Our modifications significantly reduce the overhead of statistics collection with a marginal reduction in quality.

In addition to these bulk operations, histograms are automatically updated during query processing. During normal operation, the evaluation of (almost) any predicate over a base column can lead to an update of the histogram for this column. `INSERT`, `UPDATE`, and `DELETE` statements also update the histograms for the modified columns. Each histogram's density metric is updated using a moving average with exponential smoothing. That is, given histogram density d and an observed value selectivity v, we take the new histogram density to be $d+\lambda(v-d)$ where λ is a small constant. We apply this procedure to observed selectivities due to query feedback from equality predicates, and also for selectivities of singleton buckets that are removed from the histogram.

Experience. Techniques such as query feedback enable SQL Anywhere's ability to perform well in zero-administration environments. However, a monster lurks: these self-managing techniques would occasionally fail to satisfactorily maintain high-quality statistics. Part of the issue is that statistics are modified on a per-connection basis, so concurrent transactions can result in incorrect histogram modifications. Another issue is that, to avoid concurrency bottlenecks, histogram maintenance is not transactional.

These issues have prompted the development of 'self-healing' measures to identify inaccurate statistics, determine appropriate remedies to correct the problem, and schedule those remedies for execution, all without DBA intervention. The SQL Anywhere server automatically tracks estimated predicate selectivities to actual values, and when the amount or frequency of error is considered high enough, determines how the error should be rectified. One technique is to replace outright the current histogram with a reconstructed one based on the query's experience feedback. If the error is more widespread, however, the server may schedule one of several different piggybacking mechanisms (cf. reference [13]) to reconstruct portions of the histogram through (a) piggybacking off a user query, (b) index sampling, or (c) stratified table scan sampling. In addition, the server tracks the frequency with which specific histograms are used, and concentrates its maintenance efforts on those histograms most critical to the user's workload.

These feedback mechanisms, along with on-the-fly, real-time index and table statistics, such as the number of distinct values, number of leaf pages, and clustering statistics, have greatly improved the accuracy of the statistics used for query optimization in SQL Anywhere. Recent customer experience strongly suggests that these

techniques have significantly reduced the need for manual or scheduled intervention to correct statistical errors. Nonetheless, we are continuing to enhance our histogram maintenance techniques, including the development of analysis and reporting tools to enable diagnosis of histogram and other statistical anomalies without requiring access to the raw customer data.

4 Adaptive Query Execution

SQL Anywhere uses feedback control to 'tune' buffer pool memory to meet overall system requirements, including memory demands from other applications. The feedback control mechanism uses the OS working set size, the amount of free physical memory, and the buffer miss rate as its inputs, which are ordinarily polled once per minute but are polled much more frequently at server startup in anticipation of application activity. Since the buffer pool size can grow—or shrink—on demand, query execution must be able to adapt to execution-time changes in the amount of available physical memory.

All page frames in SQL Anywhere's heterogeneous buffer pool are the same size, and can be used for table and index pages, undo or redo log pages, bitmaps, query memory, or connection heaps. Each type of page has different usage patterns: heap pages, for example, are local to a connection, while table pages are shared. Therefore, the buffer pool's page replacement algorithm, a modified generalized 'clock' algorithm [15], must be aware of differences in reference locality. No attempt is made in SQL Anywhere to support buffer pool partitions. In our view, dynamic changes to the system's workload that occur frequently in SQL Anywhere deployments render static workload configuration tools ineffective.

SQL Anywhere's query optimizer can automatically annotate a chosen query execution strategy with alternative plan operators that offer a cheaper execution technique if either the optimizer's choices are found to be suboptimal at run-time, or the operator requires a low-memory strategy at the behest of the memory governor (see below). For example, the optimizer may construct an alternative index-nested loops strategy for a hash join, in case the size of the build input is considerably smaller than that expected at optimization time. Hash join, hash-based duplicate elimination, and sorting are examples of operators that have low-memory execution alternatives. A special operator for execution of `RECURSIVE UNION` is able to switch between several alternative strategies, possibly using a different one for each recursive iteration.

To ensure that SQL Anywhere maintains a consistent memory footprint, in-memory data structures used by query processing operators are allocated within heaps that are subject to page replacement within the buffer pool. Moreover, as the buffer pool can shrink during query execution, memory-intensive query execution operators must be able to adapt to changing memory availability.

The memory governor controls query execution by limiting memory consumption for a statement to not exceed its quota. Approaching this quota may result in the memory governor requesting query execution operators to free memory if possible. For example, hash-based operations in SQL Anywhere choose a number of buckets based on the expected number of distinct hash keys; the goal is to have a small number of distinct key values per bucket. In turn, buckets are divided uniformly into a small, fixed number of partitions. The number of partitions is selected to provide a balance between I/O behaviour and fanout. During the processing of the hash operation, the governor detects when the query plan needs to stop allocating more memory—that is, it has exceeded its quota. When this happens, the partition with the largest number of rows is evicted from memory. The in-memory rows are written out to a temporary table, and incoming rows that hash to the partition are also written to disk.

By selecting the partition with the most rows, the governor frees up the most memory for future processing, in the spirit of other documented approaches in the literature [16]. By paying attention to the memory quota while building the hash table on the smaller input, the server can exploit as much memory as possible, while degrading adaptively if the input does not fit completely in memory. In addition, the server also pays attention to the quota while processing the *probe* input. If the probe input uses memory-intensive operators, their execution

may compete with the hash join for memory. If an input operator needs more memory to execute, the memory governor evicts a partition from the consuming operator's hash table. This approach prevents an input operator from being starved for memory by a consumer operator.

Experience. Adaptive query execution techniques are essential in SQL Anywhere due to the other autonomic systems within it, such as dynamic cache sizing, that can negatively impact query execution strategies. Today, adaptive execution in SQL Anywhere involves trying to salvage an execution strategy when the server context at execution time was not anticipated by the optimizer. This conservative approach 'tames' the monster of catastrophically slow execution plans or memory panic situations at execution time. So far, we have not implemented potentially expensive mitigation techniques such as on-the-fly query re-optimization. However, to better manage the tradeoff between conservative resource allocation and optimal performance, we are considering other approaches, such as more sophisticated execution plan caching techniques.

5 Self-tuning Multiprogramming Level

Several commercial database servers (e.g., Microsoft SQL Server, Oracle, and SQL Anywhere) employ a worker-per-request server architecture. In this architecture, a single queue is used for all database server requests from all connections. A worker dequeues a request from the request queue, processes the request, and then returns to service the next available request in the queue or block on an idle queue. In this configuration, there are no guarantees that a single connection will be serviced by the same worker. This architecture has proved to be more effective in handling large numbers of connections and, if configured properly, has less overhead [17]. The difficult issue with this architecture is how to set the size of the worker pool to achieve good throughput levels [18]. This parameter effectively controls the server's multiprogramming level (MPL).

Setting the server's MPL is subject to tradeoffs. A large MPL permits the server to process a large number of requests concurrently, but with the risk of thrashing due to hardware or software resource contention (locks) or to excessive context switching between threads [19]. With a small MPL, the server is able to apportion a larger amount of server memory for query execution. The obvious drawback, however, is that the server's hardware resources are under-utilized. The issue in zero-administration or deployed environments is that setting the server's MPL *a priori* is largely impossible because of changes to both the server's execution environment and the database workload over time.

SQL Anywhere overcomes this problem by dynamically adjusting the server's multiprogramming level to maximize throughput during server operation [20]. To perform the MPL adjustment, the SQL Anywhere kernel uses a feedback controller that oversees two algorithms—a hill climbing algorithm and a global parabola approximation algorithm—that significantly extends prior work by Heiss and Wagner [21]. The feedback controller periodically switches between the two algorithms. The combined approach is effective at maintaining throughput over varying workloads without large oscillations.

Experience. Hand-tuning of a server's multiprogramming level is a notoriously difficult task even with well-understood workloads, making MPL tuning a monster to be tamed. In SQL Anywhere, hand-tuning is doubly difficult because SQL Anywhere can assign multiple workers to a single request to achieve intra-query parallelism. The degree of parallelization is determined by the query optimizer in a cost-based manner, made with respect to the characteristics of the system context at optimization time. There is no static partitioning of work amongst the workers assigned to a query. Rather, a parallel query plan is organized so that any active thread can grab and process a small unit of work from anywhere in the data flow tree. The unit of work is usually a page's worth of rows. It is not important which thread executes a given unit of work; a single thread can execute the entire query if additional threads are not available. This means that parallel queries adapt gracefully to fluctuations in server load. However, autonomic MPL tuning has not slayed the monster outright; in the field, automatic adjustment does not perform well with periodic, 'bursty' workloads. Additional research is ongoing to develop better feedback control techniques to solve this problem.

6 Indexing Spatial Data

SQL Anywhere offers support for spatial data, with functionality modelled after the SQL/MM spatial standard [22]. Spatial data is indexed by *linear quadtrees* [23], which use a Z-curve to map two-dimensional quadtree tiles into 64-bit integer keys suitable for storage in a B-tree index. Query performance over linear quadtrees is heavily impacted by the granularity with which spatial objects are tessellated [24]. Choosing a tessellation granularity requires navigating a trade-off between index filtering efficacy and index size, and entails knowledge of both the data distribution and the query workload. Other commercial systems specify the tessellation granularity at index creation time, and the same tessellation algorithm is applied to both data objects (upon insertion) and to query objects over the index [24, 25]. Unfortunately, choosing tessellation granularity up front becomes problematic in embedded environments where either the data or the query distributions may not be known at design time. The approach we have taken in SQL Anywhere is to separate the choice of tessellation granularity for data and query objects. In order to ensure robust index performance in all scenarios, data objects are tessellated with a fixed coarse granularity. This conservative approach favours index size over filtering efficacy and guarantees that index scans remain as cheap as possible in case the query workload includes large containment queries; any reduction in filtering quality at the index level is mitigated by various second-stage filters.

Tessellation of spatial query objects is a great example of how several self-tuning technologies in SQL Anywhere work together. Query objects are tessellated dynamically during query execution by a cost-based algorithm that examines both the data distribution and the current server state. Within sparse data regions a query object is tessellated coarsely, thereby reducing the number of key range probes against the index; as data density increases the tessellation becomes more fine-grained, yielding more but tighter range probes. The query execution plan includes alternatives for both the spatial index or a table scan, allowing the tessellation algorithm to make a cost-based decision to revert to a table scan for large query objects whose index scanning cost is prohibitive. Given a spatial join where objects from one table form the queries against the spatial index of a second table, changes in buffer pool contents during execution of the join is taken into account by the cost function when tessellating successive query objects. Furthermore, because the knowledge of the data distribution is obtained from the server's self-tuning histograms, query tessellation adapts dynamically to both changes in the underlying data and improvements in histogram accuracy resulting from query feedback.

7 Conclusions and Future Work

Taming the various self-management monsters is exceedingly difficult. System and regression testing of autonomic systems like SQL Anywhere remain unsolved problems. Yet as challenging as these problems are, diagnosing performance problems experienced by customers can be even more difficult. Increasingly, privacy and legislative constraints prevent access to customer databases. In these situations, the existence of diagnostic tools that the customer can execute is essential. As one example, graphical representations of SQL Anywhere execution plans can contain information related to the search space considered by the query optimizer. This information ranges from the various cost model computations to the cost-based and heuristic decisions made by the optimizer during join enumeration. These traces of optimization search spaces can be analyzed in isolation, and also can be used to generate synthetic data to match the original customer database.

Database self-management does have the potential to exacerbate its own monster: unpredictable query execution times. However, our twenty years of customer experience has shown that SQL Anywhere's capacity to adapt to changing execution contexts largely trumps those infrequent situations when a specific query's execution time has become intolerable.

In the main, the term 'database self-management' typically applies to performance characteristics. However, an often-neglected aspect of self-management is with error handling, increasingly relevant due to the use of commodity hardware that will fail in predictable ways [26].

Finally, we note that with the proliferation of database systems in many different environments, backwards compatibility has become increasingly important. While such constraints are occasionally inconvenient, backwards compatibility is critical when an ISV has millions of deployed seats, and can't possibly upgrade all software instantaneously.

References

[1] Ivan T. Bowman and G. N. Paulley, "Join enumeration in a memory-constrained environment", in *Proceedings, Sixteenth* IEEE *International Conference on Data Engineering*, San Diego, California, Mar. 2000, pp. 645–654.

[2] Anisoara Nica, "A call for order in search space generation process of query optimization", in *Proceedings,* ICDE *Workshops (Self-Managing Database Systems)*, Hanover, Germany, Apr. 2011, IEEE Computer Society Press.

[3] Anisoara Nica, "Immediate materialized views with outerjoins", in *Proceedings,* ACM *Thirteenth International Workshop on Data Warehousing and* OLAP (DOLAP)*, in* ACM *Ninetheenth Conference on Information and Knowledge Management* (CIKM), Toronto, Canada, Oct. 2010, pp. 45–52.

[4] Craig Russell, "Bridging the object-relational divide", ACM *Queue*, vol. 6, no. 3, pp. 16–27, May 2008.

[5] Ivan T. Bowman, Peter Bumbulis, Dan Farrar, Anil K. Goel, Brendan Lucier, Anisoara Nica, G. N. Paulley, John Smirnios, and Matthew Young-Lai, "SQL Anywhere: A holistic approach to database self-management", in *Proceedings,* ICDE *Workshops (Self-Managing Database Systems)*, Istanbul, Turkey, Apr. 2007, pp. 414–423, IEEE Computer Society Press.

[6] G. N. Paulley and Per-Åke Larson, "Exploiting uniqueness in query optimization", in *Proceedings, Tenth* IEEE *International Conference on Data Engineering*, Houston, Texas, Feb. 1994, pp. 68–79, IEEE Computer Society Press.

[7] Kristofer Vorwerk and G. N. Paulley, "On implicate discovery and query optimization", in *Proceedings,* IEEE *International Data Engineering and Applications Symposium*, Edmonton, Alberta, July 2002, pp. 2–11, IEEE Computer Society Press.

[8] Anisoara Nica, Daniel Scott Brotherston, and David William Hillis, "Extreme visualisation of the query optimizer search spaces", in ACM SIGMOD *International Conference on Management of Data*, Providence, Rhode Island, June 2009, pp. 1067–1070.

[9] Ashraf Aboulnaga and Surajit Chaudhuri, "Self-tuning histograms: Building histograms without looking at data", in ACM SIGMOD *International Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999, pp. 181–192.

[10] Nicolas Bruno and Surajit Chaudhuri, "Efficient creation of statistics over query expressions", in *Proceedings, Nineteenth* IEEE *International Conference on Data Engineering*, Bangalore, India, Mar. 2003, pp. 201–212.

[11] Surajit Chaudhuri and Vivek Narasayya, "Automating statistics management for query optimizers", in *Proceedings, Sixteenth* IEEE *International Conference on Data Engineering*, San Diego, California, Mar. 2000, pp. 339–348, IEEE Computer Society Press.

[12] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga, "CORDS: Automatic discovery of correlations and soft functional dependencies", in ACM SIGMOD *International Conference on Management of Data*, Paris, France, June 2004, pp. 647–658.

[13] Qiang Zhu, Brian Dunkel, Wing Lau, Suyun Chen, and Berni Schiefer, "Piggyback statistics collection for query optimization: Towards a self-maintaining database management system", *Computer Journal*, vol. 47, no. 2, pp. 221–244, 2004.

[14] Michael Greenwald, "Practical algorithms for self-scaling histograms or better than average data collection", *Performance Evaluation*, vol. 20, no. 2, pp. 19–40, June 1996.

[15] Alan Jay Smith, "Sequentiality and prefetching in database systems", ACM *Transactions on Database Systems*, vol. 3, no. 3, pp. 223–247, Sept. 1978.

[16] Hans-Jörg Zeller and Jim Gray, "An adaptive hash join algorithm for multiuser environments", in *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, Aug. 1990, pp. 186–197.

[17] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman, "How to determine a good multi-programming level for external scheduling", in *Proceedings, 22nd* IEEE *International Conference on Data Engineering*, Washington, DC, Apr. 2006, p. 60, IEEE Computer Society.

[18] Stavros Harizopoulos, *Staged Database Systems*, PhD thesis, Carnegie Mellon University, 2005.

[19] Shimin Chen, Phillip B. Gibbons, Tood C. Mowry, and Gary Valentin, "Fractal prefetching B^+-trees: Optimizing both cache and disk performance", in ACM SIGMOD *International Conference on Management of Data*, Madison, Wisconsin, June 2002, pp. 157–168, Association for Computing Machinery.

[20] Mohammed Abouzour, Kenneth Salem, and Peter Bumbulis, "Automatic tuning of the multiprogramming level in Sybase SQL Anywhere", in *Proceedings, 26th* IEEE *International Conference on Data Engineering: Workshops*, Long Beach, California, Mar. 2010, pp. 99–104, IEEE Computer Society Press.

[21] Hans-Ulrich Heiss and Roger Wagner, "Adaptive load control in transaction processing systems", in *Proceedings, 17th International Conference on Very Large Data Bases*, Barcelona, Spain, Sept. 1991, pp. 47–54, Morgan-Kaufmann.

[22] International Standards Organization, ISO/IEC *13249-3:2011, Information technology—Database languages—SQL Multimedia and application packages: Part 3, Spatial*, 2011.

[23] Irene Gargantini, "An effective way to represent quadtrees", *Communications of the* ACM, vol. 25, no. 12, pp. 905–910, Dec. 1982.

[24] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov, "Quadtree and R-tree indexes in Oracle spatial: a comparison using GIS data", in ACM SIGMOD *International Conference on Management of Data*, Madison, Wisconsin, 2002, pp. 546–557, Association for Computing Machinery.

[25] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid, "Spatial indexing in Microsoft SQL Server 2008", in ACM SIGMOD *International Conference on Management of Data*, Vancouver, Canada, 2008, pp. 1207–1216, Association for Computing Machinery.

[26] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "An analysis of data corruption in the storage stack", in *FAST*, Mary Baker and Erik Riedel, Eds. 2008, pp. 223–238, USENIX.