

SQL Anywhere: An Embeddable DBMS

Ivan T. Bowman Peter Bumbulis Dan Farrar Anil K. Goel Brendan Lucier
Anisoara Nica G. N. Paulley John Smirnios Matthew Young-Lai
Sybase iAnywhere

Abstract

We present an overview of the embeddability features of SQL Anywhere, a full-function relational database system designed for frontline business environments with minimal administration. SQL Anywhere supports features common to enterprise-class database management systems, such as intra-query parallelism, materialized views, OLAP functionality, stored procedures, triggers, and hot failover. SQL Anywhere can serve as a high-performance workgroup server, an embedded database that is installed along with an application, or as a mobile database installed on a handheld device that provides full database services, including two-way synchronization, to applications when the device is disconnected from the corporate intranet. We illustrate how SQL Anywhere's embeddability features work in concert to provide a robust data management solution in zero-administration environments.

1 Introduction

Database systems have become ubiquitous across the computing landscape. This is partly because of the basic facilities offered by database management systems: physical data independence, ACID transaction properties, a high-level query language, stored procedures, and triggers. These facilities permits sophisticated applications to 'push' much of their complexity into the database itself. The proliferation of database systems in the mobile and embedded market segments is due, in addition to the features above, to the support for two-way database replication and synchronization offered by most commercial database management systems. Data synchronization technology makes it possible for remote users to both access *and update* corporate data at a remote, off-site location. With local (database) storage, this can be accomplished even when disconnected from the corporate network.

In this paper, we describe a wide range of technologies that permit SQL Anywhere [2] to be used in embedded and/or zero-administration environments. In Section 2, we outline architectural features of SQL Anywhere that are helpful, if not essential, in embedding the database with a specific application to provide data management services. In Section 3, we briefly describe self-management technologies within the SQL Anywhere server to enable its operation in zero-administration environments. 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. It is important to note that 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.

Copyright 2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Embeddability-enabling technologies

SQL Anywhere was designed from the outset to offer self-management features permitting its deployment as an embedded database system. The deployment of a SQL Anywhere server and database instance can be integrated with the application's installation procedure; both database files and executable program files can be renamed to fit the application's requirements. Moreover, the server can be configured so that its operation is transparent. Below, we highlight some of the embeddability features that enable SQL Anywhere to be used as an embedded database.

Autostart and autostop. A SQL Anywhere server instance can be started by a simple client API call from the application over a shared-memory connection on the same physical hardware. Once the server is started, additional databases can be started or stopped via SQL statements sent from any application. In turn, a server or database can be configured to shut down automatically when the last connection disconnects.

Database storage utilizes the native file system. SQL Anywhere databases are stored as ordinary OS files and can be managed with the file utilities provided by the operating system. Each database consists of a main database file, a separate transaction log file, and up to 12 additional files that can be placed on the same filesystem or spread across several. Raw partitions are not supported. The benefit of this simplicity is *deployment flexibility*. To image copy a database, one simply copies all of its associated files; execution of a copy utility program is not required. Database files are portable amongst all supported platforms, including Windows CE devices, and even if the machines are of different CPU architectures. This flexibility makes database deployment, application development, and problem determination, particularly to handheld devices and/or remote locations, significantly easier.

The transaction log contains logical operations. Unlike write-ahead logging approaches such as Aries [6] that log physical operations, SQL Anywhere's transaction log contains logical operations. In addition to providing the basis for database synchronization operations, the forward transaction log can be translated into equivalent SQL statements, which can then be applied to any other database instance with the same schema. This architecture offers application vendors considerable flexibility when performing problem determination and, if necessary, database reconstruction at remote locations.

Integrated security. SQL Anywhere provides full end-to-end security with 128-bit strong encryption of database tables, files, and communications streams between the application and the database, as well as the MobiLink synchronization stream. SQL Anywhere offers built-in user authentication (including support for Kerberos), and can integrate with third-party authentication systems. Client applications can verify the identity of a server using digital signatures or signed certificates. In addition, application vendors can *authenticate* a server, which can prevent usage of the server by other (unauthorized) applications entirely, or restrict their capabilities. SQL Anywhere also offers FIPS-certified encryption via a separately licensed security option. Further, SQL Anywhere supports encryption of business logic stored in the database to prevent reverse engineering of the OEM application logic (stored procedures, views, triggers and so on).

Application profiling. To support problem determination and analysis of application performance problems in the field, SQL Anywhere contains built-in application profiling that can obtain a detailed trace of all server activity, including SQL statements processed, performance counters, and contention for rows in the database. This trace information is captured as an application runs, and is transferred via a TCP/IP link into any SQL Anywhere database, where it can be analyzed. This flexible architecture allows for the trace to be captured with a focus on convenience (by storing the trace in the same database that generated it) or on performance (by storing the trace data on a database on a separate physical machine). The architecture also permits the Application Profiling tool to analyze and make recommendations, including index recommendations, for databases on mobile devices running Windows CE.

Quiet operation. Both the personal and the network SQL Anywhere server can be configured to operate in 'quiet' mode. During quiet operation, no console log is displayed, and the server does not issue startup messages or issue prompts for service. This permits the OEM application to be in complete control of the user interface for

the application.

Active database support. SQL Anywhere supports events, using either timer-based or event-based scheduling of event handlers (stored procedures), which can permit automatic handling of a variety of conditions, from ‘disk full’ errors to application disconnection. The server’s built-in SMTP support permits event handlers to issue email notification of events where desirable.

Support for web services. SQL Anywhere contains a built-in HTTP server that allows the server to function as a web server, as well as to permit access to web services in other SQL Anywhere databases and standard web services available over the Internet. SOAP is the standard used for this purpose, but the built-in HTTP server in SQL Anywhere also supports standard HTTP and HTTPS requests from client applications. As a consequence, it is possible to embed an entire application within a database instance, using the active database components supported by the server, along with the server’s built-in XML capabilities. If packaged in this way, deploying the ‘application’, with its database, is as simple as deploying the database instance itself; all that is required for a user interface is a web browser.

3 Self-management technologies

3.1 Dynamic buffer pool management

When a database system is embedded in an application as part of an installable package, it cannot normally use all the machine’s resources. Rather, it must co-exist with other software and system tools whose configuration and memory usage vary from installation to installation, and from moment to moment. In this environment, it is difficult to predict the system load or the amount of memory that will be available at any point in time.

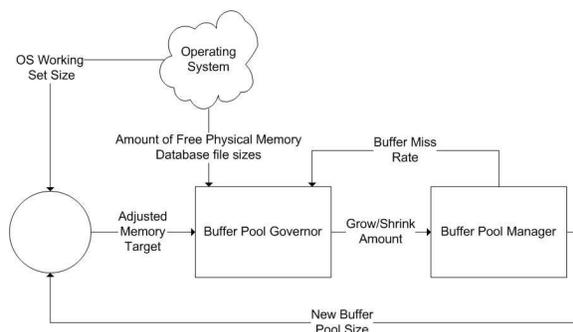


Figure 1: Cache sizing feedback control

Consequently, SQL Anywhere uses the following approach to buffer pool management: rather than attempting to ‘tune’ buffer pool memory in isolation, the server tunes buffer pool allocation to fit the overall system requirements. It does this by using a feedback control mechanism, using the OS working set size, the amount of free physical memory, and the buffer miss rate as the inputs (see Figure 1). The OS working set size, which is polled every minute, is the operating system’s amount of real memory in use by the process. Using this feedback control loop, the server’s buffer pool grows or shrinks on demand, depending on system-wide resource usage and the memory management policy of the operating system.

This adjustment can be done very efficiently on some operating systems that permit address space to be allocated to a process independent of backing physical memory. The variability of the buffer pool size has implications for query processing. Queries must adapt to execution-time changes in the amount of available physical memory (see Section 3.3).

Heaps. A novel feature of SQL Anywhere is that the buffer pool is a single heterogeneous pool of all types of pages: table pages, index pages, undo and redo log pages, bitmaps, free pages, and heap pages. To support efficient buffer pool management, all page frames are the same size.

Data structures created and utilized for query processing, including hash tables, prepared statements, cursors, and similar structures are allocated inside pages in a heap. When a heap is not in use—for example, when the server is awaiting the next FETCH request from the application—the heap is ‘unlocked’. Pages in unlocked heaps can be stolen and used by the buffer pool manager for other purposes, such as caching table or index pages, as required. When this happens, the stolen pages are swapped out to the temporary file. To resume the processing of the next request when it arrives, the heap is re-locked, pinning its pages in physical memory. A

pointer swizzling technique (cf. reference [4]) is used to reset pointers in pages forced to be relocated during re-locking.

Page replacement strategy. As described above, all page frames in the buffer pool are the same size, and can be used for various purposes. Each type of page has different usage: heap pages, for example, are local to a connection, while table pages are shared. The buffer pool's page replacement algorithm must be aware of the differences in usage as well as those in reference locality in page access patterns. For example, it must recognize that adjacent references to a single page during a table scan are different from other reference patterns. SQL Anywhere uses a modified generalized 'clock' algorithm [9] for page replacement. In addition, we have implemented techniques to reduce the overhead of the clock algorithm, yet quickly recognize page frames that can be reused.

3.2 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. Subsequent work by other researchers [1, 12] 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.

3.2.1 Histogram implementation

SQL Anywhere uses self-tuning histograms whose bucket pools expand and contract dynamically as column value distribution changes are detected. The structure of bucket boundaries shifts over time to minimize error and allocate more resolution where it is needed. 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'. Singletons are implemented as buckets ranging over single values, and are interleaved with traditional buckets in the histogram structure. A value that represents at least 1% of the column data or appears in the 'top N' most frequent 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].

For efficiency and simplicity, the same infrastructure is used for almost all data types. 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. The exceptions are longer string and binary data types, for which 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 data is updated using INSERT, UPDATE, and DELETE statements, when an index is created, or when an explicit CREATE STATISTICS statement is executed. A modified version of Greenwald's algorithm [5] 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 data change 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.

Join histograms are computed on-the-fly during query optimization to determine the cardinality and data distribution of intermediate results. Join histograms are created over single attributes. In cases where the join condition is over multiple columns, a combination of existing referential integrity constraints, index statistics, and density values are used to compute and/or constrain join selectivity estimates.

A variety of additional statistics are automatically maintained for other database objects. For stored procedures used in table functions, the server maintains a summary of statistics for previous invocations, including total CPU time and result cardinality. A moving average of these statistics is saved persistently in the database for use by the optimizer for subsequent queries. In addition, statistics specific to certain values of the procedure's input parameters are saved and managed separately if they differ sufficiently from the moving average. Index statistics, such as the number of distinct values, number of leaf pages, and clustering statistics, are maintained in real time during server operation. Table statistics, in particular the percentage of a table resident in the buffer pool, are also maintained in real time and used by the cost model when computing a table's access cost.

3.3 Query processing

In our experience, there is little correlation between application or schema complexity, and the database size or deployment platform. Developers tend to complicate, rather than simplify, application design when they migrate applications to business front-lines, even when targeting platforms like hand-held devices with few computing resources. It is usually only the user-interface that is re-architected because of the input mode differences on such devices.

This complexity makes sophisticated query processing an essential component of SQL Anywhere. As described in Section 3.1, the operating characteristics of the server can change from moment to moment. In mixed-workload systems with complex queries, this flexibility demands that query processing algorithms adapt to changes in the amount of memory they can use. It also means that the query optimizer must take the server state into account when choosing access plans.

3.3.1 Query optimization

SQL Anywhere (re)optimizes a query¹ at each invocation. There are two broad exceptions to this. The first class of exceptions is simple DML statements, restricted to a single table, where the cost of optimization approaches the cost of statement execution. In such cases, these statements bypass the cost-based optimizer, and are optimized heuristically. The second class is statements within stored procedures and triggers. For these statements, access plans are cached on an LRU basis for each connection. A statement's plan is only cached, however, if the access plans obtained by successive optimizations of that statement during a 'training period' are identical. After the training period is over, the cached plan is used for subsequent invocations. However, to ensure the plan remains 'fresh', the statement is periodically verified at intervals taken from a decaying logarithmic scale.

Re-optimization of each query means that optimization cost cannot be amortized over many executions. Optimization must therefore be cheap. One of several techniques used to reduce optimization cost is to limit the size of the optimizer's search space. The SQL Anywhere optimizer uses a proprietary branch-and-bound, depth-first search enumeration algorithm [8] over left-deep processing trees². Depth-first search has the significant advantage of using very little memory; in fact, much of the state information required by the algorithm can be kept on the processor stack. In this approach, enumeration and cost estimation are interleaved.

¹In this context we use the term 'query' to refer to not only queries but also to INSERT, UPDATE, and DELETE statements.

²Left-deep trees are used except for cases involving complex derived tables or table expressions containing outer joins.

The enumeration algorithm first determines a heuristic ranking of the tables involved in the join strategy. In addition to enumerating tables or table functions, the algorithm also enumerates materialized views matching parts of the query, and complex subqueries by converting them into joins on a cost basis [7]. Join and index physical operators are also enumerated during join strategy generation which makes possible costing plans with intra-query parallelism. By considering tables in rank order, the enumeration algorithm initially (and automatically) defers Cartesian products to as late in the strategy as possible. Hence it is likely that the first join strategy generated, though not necessarily optimal, will be one with a ‘reasonable’ overall cost, relative to the entire search space. The algorithm is branch-and-bound in the sense that a partial join strategy is retained only if its cost is provably less than the cost of the best complete join strategy discovered thus far.

An interesting characteristic of the branch-and-bound enumeration algorithm is the method by which the search space is pruned during join strategy generation. The algorithm incrementally costs the prefix of a join strategy and backtracks as soon as the cost of an intermediate result exceeds that of the cheapest complete plan discovered thus far. Since any additional quantifiers can only add to the plan’s cost, no join strategy with this prefix of quantifiers can possibly be cheaper and the entire set of such strategies can be pruned outright. This pruning is the essence of the algorithm’s branch-and-bound paradigm. It is described in detail in reference [3].

Additional care must be taken when analyzing cost measures for an intermediate result. For example, a significant component of any plan’s cost concerns its buffer pool utilization. However, measures such as buffer hit ratios can be accurately estimated only with regard to a *complete* execution strategy, since in a fully-pipelined plan the most-recently used pages will be from those tables at the root of the processing tree. Nonetheless, it is possible to estimate the cost of computing an intermediate result based on a *very* optimistic metric: assume that half the buffer pool is available for each quantifier in the plan. Clearly this is nonsense with any join degree greater than 1. However, the point is not to accurately cost intermediate results, but to quickly prune grossly inefficient strategies from the search space.

Another notable characteristic of the join enumeration algorithm involves the control strategy used for the search [8]. A problem with join enumeration using a branch-and-bound approach with early halting is that the search effort is not well-distributed over the entire search space. If a small portion of the entire space is visited, most of the enumerated plans will be very similar. SQL Anywhere addresses this problem by employing an optimizer governor [10] to manage the search. The governor dynamically allocates a quota of search effort across sections of the search space to increase the likelihood that an efficient plan is found. This quota is unevenly distributed across similar join strategies so that more effort is spent on heuristically higher-ranked strategies. Initial quota can be specified within the application, if desired, allowing fine-grained tuning of the optimization effort spent on each statement.

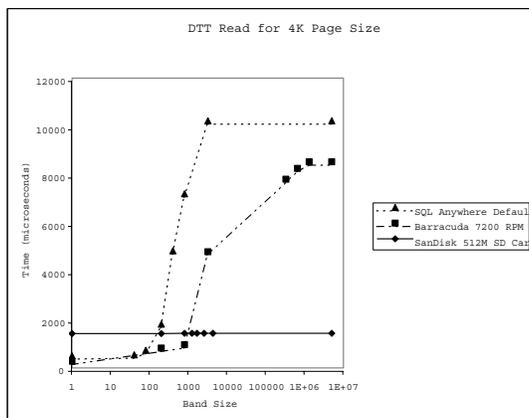


Figure 2: DTT models

Disk transfer time model. SQL Anywhere uses a Disk Transfer Time (DTT) model [4] to estimate a query’s expected I/O cost. A ‘generic’ model is used by default. It has been validated using a systematic testing framework over a variety of machine architectures and disk subsystems. While the default DTT model performs well for a range of hardware devices, SQL Anywhere also provides a mechanism to calibrate using a specific hardware device. Further, the calibrated model can be saved in a file and loaded into multiple other databases. This approach is very useful in cross-platform development where the development platform is significantly different from the deployment platform.

The DTT function summarizes disk subsystem behaviour with respect to an application (in this case, the SQL Anywhere server). The DTT function models the amortized cost of reading one page randomly over a *band size* area of the disk. If the band

size is 1, the I/O is sequential; otherwise, it is random. Significantly larger band sizes increase the average cost of each retrieval because of a higher probability of each retrieval requiring a seek, and the increase in the travel time of the disk arm to reach the correct cylinder. Figure 2 illustrates the default and calibrated DTT taken from two different hardware configurations.

3.3.2 Adaptive query execution

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 conditions.

Each task, or unit-of-work, within the SQL Anywhere server is given a quota of available memory by the server’s memory governor. There are two quotas computed for each task:

- a *hard* memory limit: if exceeded, the statement is terminated with an error.
- a *soft* memory limit, which the query processing algorithms used for the statement should not exceed, if at all possible. Approaching this limit may result in the memory governor requesting query execution operators to free memory if possible.

The memory governor controls query execution by limiting memory consumption for a statement to not exceed the soft limit. 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 the soft limit. 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 [11]. By paying attention to the soft limit 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 soft limit 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.

Adaptive intra-query parallelism. SQL Anywhere can assign multiple threads to a single request, thus achieving intra-query parallelism. There is no static partitioning of work amongst the threads 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.

4 Conclusions

Virtually every new feature implemented in SQL Anywhere is designed to be adaptive or self-managing. In addition to the embedded and self-management technologies described above, SQL Anywhere also includes a variety of tools and technologies that are useful during the development of a database application, including an index selection utility called the *Index Consultant*, graphical administration and modeling tools, and a full-function stored procedure debugger. Going forward, we will continue to develop autonomic features that provide greater degrees of self-tuning and self-management.

References

- [1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *ACM SIGMOD International Conference on Management of Data*, pages 181–192, Philadelphia, Pennsylvania, May 1999.
- [2] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. SQL Anywhere: A holistic approach to database self-management. In *Proceedings, ICDE Workshops (Self-Managing Database Systems)*, pages 414–423, Istanbul, Turkey, Apr. 2007. IEEE Computer Society Press.
- [3] I. T. Bowman and G. N. Paulley. Join enumeration in a memory-constrained environment. In *Proceedings, Sixteenth IEEE International Conference on Data Engineering*, pages 645–654, San Diego, California, Mar. 2000.
- [4] A. K. Goel. *Exact Positioning of Data Approach to Memory Mapped Persistent Stores: Design, Analysis and Modelling*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1996.
- [5] M. Greenwald. Practical algorithms for self-scaling histograms or better than average data collection. *Performance Evaluation*, 20(2):19–40, June 1996.
- [6] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.
- [7] A. Nica. System and methodology for cost-based subquery optimization using a left-deep tree join enumeration algorithm. US Patent 2004/0220923, Nov. 2004.
- [8] A. Nica. System and methodology for generating bushy trees using a left-deep tree join enumeration algorithm. US Patent 7,184,998, Feb. 2007.
- [9] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, Sept. 1978.
- [10] M. Young-Lai. Database system with methodology for distributing query optimization effort over large search spaces. US Patent 6,807,546, Oct. 2004.
- [11] H.-J. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 186–197, Brisbane, Australia, Aug. 1990.
- [12] Q. Zhu, B. Dunkel, W. Lau, S. Chen, and B. Schiefer. Piggyback statistics collection for query optimization: Towards a self-maintaining database management system. *Computer Journal*, 47(2):221–244, 2004.