SQL Anywhere: A Holistic Approach to Database Self-management*

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

Sybase iAnywhere
Waterloo, Ontario, Canada
E-mail: paulley@ianywhere.com

Abstract

In this paper we present an overview of the self-management features of SQL *Anywhere, a full-function relational database system designed for frontline business environments with minimal administration.* 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 the various self-management 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. This 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, a commonplace circumstance in frontline business environments.

SQL Anywhere[1] is a full-function, ANSI SQL-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 supports features common to enterprise-class database management systems, such as intra-query parallelism, materialized views, OLAP functionality, stored procedures, triggers, and hot failover, and does so on a variety of 32- and 64-bit hardware platforms, including Microsoft Windows, various flavours of UNIX including Linux, Solaris, and AIX, Novell Netware, Apple Macintosh, and Windows CE. However, the strengh of SQL Anywhere is in its ability to offer SQL data management, query processing, and synchronization capabilities in zero-administration environments.

SQL Anywhere was designed from the outset to offer self-management features permitting its deployment as an embedded database system. As an example, a SQL Anywhere database can be started by a simple client API call from the application, and can shut down automatically when the last connection disconnects. As a second example, 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 database 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 database-specific copy utility 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 flexibil-

* ©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.

1 Sybase and SQL Anywhere are trademarks of Sybase Inc. Other company or product names referenced herein are trademarks and/or servicemarks of their respective companies.

ity makes database deployment, application development, and problem determination significantly easier.

In this paper, we describe a wide range of technologies that permit SQL Anywhere to be used in embedded and/or zero-administration environments. These technologies are fundamental components of the server, not merely administrative add-ons that assist a database administrator in configuring the server's operation. 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.

The paper is structured as follows. In Section 2 we describe how SQL Anywhere manages memory, specifically how it permits the buffer pool to grow and shrink dynamically. Section 3 contains an overview of the server's self-managing statistics implementation. Section 4 describes the server's adaptive query optimization and execution features, especially those that pertain to memory usage. Section 5 describes tools provided to assist with physical database and application design. Section 6 concludes the paper.

2. 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. It may be possible to perform workload analysis of the application to determine database configuration parameters such as the server's multiprogramming level [3]. However, it is difficult to predict the system load or the amount of memory that will be available at any point in time.

Therefore, 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 with the OS working set size as one input (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-

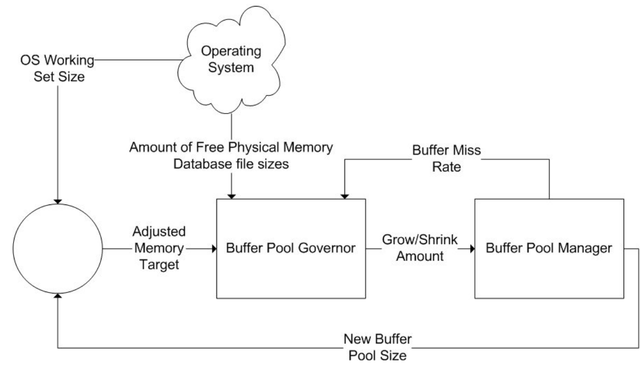


Figure 1. Cache sizing feedback control

time changes in the amount of available physical memory (see Section 4).

The database server attempts to set its buffer pool size to the current OS working set size plus the amount of unused physical memory, keeping a small amount (5 MB) in reserve for use by the OS. If the target size differs from the current size by less than 64 KB, then the buffer pool size remains unchanged. The target buffer pool size is constrained within a lower bound and an upper bound. These bounds do not change during the lifetime of the server and their default values can be overridden at server start. The upper bound determined at startup is a hard limit, but SQL Anywhere also imposes a soft limit by ensuring that the target buffer pool size is between the lower bound and

$$min(database\ size + main\ heap\ size,\ upper\ bound). \quad (1)$$

Database size includes the size of the temporary files used for intermediate results. Hence, larger temporary files will automatically unconstrain the maximum buffer pool size.

In addition to the two reference inputs in the feedback control loop—the OS working set size and the amount of free physical memory—the buffer pool manager also monitors the buffer miss rate. If there are no buffer pool misses between polling times, the buffer pool governor will not permit the buffer pool to grow. A lack of buffer pool page replacements may mean the server is largely idle, or that the working set of database pages is entirely resident in the buffer pool; either situation makes it unnecessary to increase its size. However, the buffer pool is always allowed to shrink, regardless of buffer pool activity, if the new target buffer pool size is smaller than the current size.

To avoid undesirable fluctuations, the server applies a damping factor to size changes by resizing the pool to

$$0.9 * new\ ideal\ size + 0.1 * current\ size. \quad (2)$$

Our general experience is that the SQL Anywhere dynamic buffer pool sizing heuristics are stable when the overall system load is not wildly fluctuating. Since the sampling period is nominally one minute, the buffer pool size does not respond rapidly to sudden changes to memory demands within the system. However, the server will decrease its sampling period to 20 seconds at startup and when the database grows significantly, to respond more quickly when it is expected that the demands on memory by the database server may be changing rapidly. The sampling period is not affected by sudden memory usage fluctuations elsewhere in the system.

A slightly modified version of this feedback control algorithm is used on Windows CE platforms because the Windows CE operating system resource manager lacks the ability to report the current working set size for an application. Instead, the feedback control algorithm utilizes as its reference input the *current* buffer pool size. In this case, the buffer pool will grow only when there is an increase in the amount of free memory available on the device. However, and more importantly, the buffer pool may shrink when other applications allocate additional memory.

2.1. 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.

To ensure that SQL Anywhere maintains a consistent memory footprint, in-memory data structures created and utilized for query processing, including hash tables, prepared statements, cursors, and similar structures, are allocated within heaps. 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 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 request, the heap is re-locked, pinning its pages in physical memory. A pointer swizzling technique (cf. reference [7]) is used to reset pointers in pages relocated during re-locking.

For a number of key data structures, SQL Anywhere uses disk-based implementations to eliminate or reduce the need for limits that would require tuning, automatically or otherwise. For example, to reduce stack pressure, SQL Anywhere copies and resets the stack, and restores it afterwards, before executing a stored procedure. As well, long-term locks are stored in a disk-based extensible hash table, avoiding the need for specifying a lock table size or lock escalation thresholds. Similarly, strings longer than a page have disk-based implementations. These techniques allow SQL Anywhere to eliminate restrictions on what data types can be indexed.

SQL Anywhere uses lightweight fibers rather than OS threads on systems where they are available. One advantage of fibers is that the server can schedule tasks itself, rather than accepting the decisions made by the OS thread scheduling algorithm. However, the main adaptive advantage is that fibers contribute to an overall framework for flexible reallocation of memory resources. If a request running on a fiber blocks or is suspended, and its heaps are swapped out, then its memory and address space requirements are very small. This, as well as techniques described below such as low-memory fallbacks for query plans, combine to give the server much flexibility in reducing its memory footprint on demand.

2.2. Page replacement strategy

As described above, all page frames in the buffer pool are the same size, and can be used for table and index pages, undo or redo log pages, bitmaps, 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 must be aware of differences in reference locality. 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 [18] 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. Conceptually, the clock algorithm uses a moving 'window' over the entire buffer pool, where pages are ordered by their time of last reference. The entire buffer pool is divided into eight segments based on the page reference time series. The 'score' of a page is incremented as it moves from segment to segment. Pages with lower scores are candidates for replacement. Page scores are decayed exponentially to ensure that all pages can eventually become candidates for replacement if they are not re-referenced.

One modification to this basic algorithm is the addition of a lookaside queue of pages that can be reused immediately. Typically, pages in this queue are heap and temporary table pages. The queue is implemented using a lock-free array that allows a fast decision whether a page is reusable. If the queue is empty, then the clock algorithm is used to choose a page to replace. It is important that the queue be lock-free to avoid the use of semaphores, which are expensive to implement on most hardware platforms.

3. Self-managing statistics

SQL Anywhere has used query execution feedback techniques since 1992 to gather statistics automatically during query processing. Rather than require explicit statistics generation by scanning or sampling persistent data, the server automatically collects statistics as part of query execution. More recent work by other researchers [1, 24] has also exploited this notion of collecting statistics as a side effect of query execution.

In its early releases, SQL Anywhere computed frequent-value statistics from the evaluation of equality and Is Null predicates in a manner similar to that described by Lynch [11], 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. These frequent-value statistics were also used to compute a density measure for each base table column that was used to compute join selectivity.

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

3.1. Histogram implementation

SQL Anywhere histograms are equi-depth histograms [10] 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 cardinality of the table and the column's data distribution, but lies in the range [0,100]. A 'density' value is also computed for each column, which represents the average selectivity of a single value that is not saved as a singleton bucket. Density values are used by the query optimizer as a guide for estimating the cardinality of intermediate results and for interpolating within a bucket. When appropriate, a histogram may consist entirely of singleton buckets, in which case a 'compressed' representation is employed that maintains only one domain value for each bucket.

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 *numeric* data types including date and time, the hash function is simply a conversion of the underlying value to a double-precision floating point value. For short strings the hash is obtained by constructing an integer value representing the binary values of characters in the string. In order to maintain discreteness of domains, each data type is also assigned a *value width* which represents the difference between two consecutive values in the domain, e.g., the value widths for INT and REAL are 1 and 1e-35, respectively.

For longer string and binary data types, SQL Anywhere uses a different infrastructure that dynamically maintains a list of observed predicates and their selectivities. Rather than save individual string values for bucket boundaries—which can potentially be very long—a non order-preserving hash function value is used. Each bucket is represented by a hash value, a relational predicate (equality, non-equality, BETWEEN, IS NULL, or LIKE) and the associated selectivity for the specified predicate on the histogram column. When collecting statistics on these string columns, not only are buckets created for entire string values, but buckets are also created for 'words' in the string, where a 'word' is loosely defined as any sequence of characters separated by any amount of white space. These buckets are useful in estimating the selectivity of LIKE predicates since we have found, in our experience, that many applications perform string searches using a LIKE pattern intended to match a 'word' somewhere in the string.

3.2. 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 [8] is used to create the cumulative distribution function for each table 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. In addition, INSERT, UPDATE, and DELETE statements also update the histograms for the modified columns.

Join histograms are computed on-the-fly during query optimization to determine the cardinality of intermediate results. As with column histograms, join histograms are over a single attribute. In cases where the join condition is over multiple columns, a combination of existing referential in-

tegrity constraints, index statistics, and density values is used to compute and/or constrain join selectivity estimates.

A variety of statistics, other than column histograms, are automatically maintained for other database objects. For stored procedures used in a FROM clause, 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 optimization of 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 the cost of an access method.

4. 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 2, the operating characteristics of the server can change from moment to moment. In simple OLTP workloads with uniform data distributions, the effect of changes in buffer pool size can be easily approximated [20]. However, 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.

4.1. Query optimization

SQL Anywhere (re)optimizes a query[2] at each invocation. There are two broad exceptions to this. The first class of exceptions are 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 are statements within stored procedures, user-defined functions, 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 [16] over left-deep processing trees[3]. 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. As an example, a 100-way join query against a small TPCH database can be optimized and executed by SQL Anywhere on a Dell Axim device, running Windows Mobile 5, with as little as 3 MB of buffer pool, with only 1 MB needed for optimization.

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 complex subqueries by converting them into joins on a cost basis [15]. 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.

There are several interesting characteristics of the branch-and-bound enumeration algorithm. The first is how 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 [2].

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

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

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 [9, 12, 19]. However, measures such as buffer hit ratios can be accurately estimated only for a *complete* execution strategy, because 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 cost intermediate results accurately, but to prune grossly inefficient strategies from the search space quickly.

A second notable characteristic of the join enumeration involves the control strategy for the search algorithm. As described in reference [16], the SQL Anywhere optimizer's search space is a tree. Conceptually, the root of the tree represents an 'empty' join strategy, with no quantifiers placed in the plan. Each of the root's children represents a $\langle$ quantifier, index, join method $\rangle$ 3-tuple that can be the first quantifier in the plan. Level 1 has at most n quantifiers, but may have many 3-tuples depending on how many different index and join method combinations can be used with each quantifier. This 'first' quantifier corresponds to the left-most, and therefore deepest, plan node in the left-deep strategy. Each of these 3-tuples at level 1 in turn has m 3-tuple children formed from the other $n-1$ quantifiers that can be placed at that position. The 3-tuple children are heuristically ordered by the enumeration algorithm such that the most 'promising' ones are enumerated first. Note that the search space tree may be unbalanced due to inherent constraints of processing the particular query. For example, if the query contains a LEFT OUTER JOIN, the preserved side of the join must precede the null-supplied side in the left-deep join strategy tree.

A problem with traversing the above search tree 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 [21] 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. The amount of effort spent in optimization is measured by the number of visits to each node in the search tree. A quota of visits is used to limit the search effort in any subtree. This quota is unevenly distributed across siblings so that more is given to 'promising' 3-tuples at each level based on their heuristic ordering. Conceptually, each node assigns $1/2$ of its quota to its first child, $1/2$ of whatever remains after visiting that child to the second child, and so on. The initial quota can be specified within the application, if desired, allowing fine-grained tuning of the optimization effort spent on each statement.

Implementation of the above algorithm is slightly different from the conceptual view since the route taken by the search is not known in advance. Potential 3-tuples for each remaining unplaced quantifier are determined for each join strategy prefix. During enumeration, if a set of strategies is pruned at level m in the tree, then any unused quota is returned to the node above it at level $m-1$. As a further refinement, when a new optimal plan is discovered that improves the overall estimated cost by at least 20%, any remaining quota for that search path is completely redistributed, starting at the root, which represents the lowest-level node in the left-deep processing tree. This has the benefit of concentrating additional quota in a portion of the search space that has yielded at least one good plan, in anticipation of possibly finding other efficient plans in that portion of the space.

4.2. Disk transfer time model

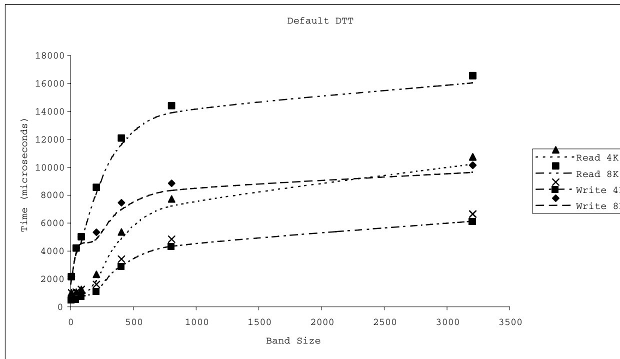
SQL Anywhere uses a Disk Transfer Time (DTT) model [7] to estimate a query's expected I/O cost. A 'generic' model (see Figure 2(a)) is used by default. It has been validated using a systematic testing framework over a variety of machine architectures and disk subsystems. While complete accuracy of any cost model is a worthwhile goal, in practice such accuracy is not necessary. The primary objective for the cost model is to ensure that for any query plans P_1 and P_2, we have

$$\begin{aligned} &Cost_E(P_1) > Cost_E(P_2) \;\; \textit{iff} \\ &\quad Cost_A(P_1) > Cost_A(P_2). \end{aligned} \tag{3}$$

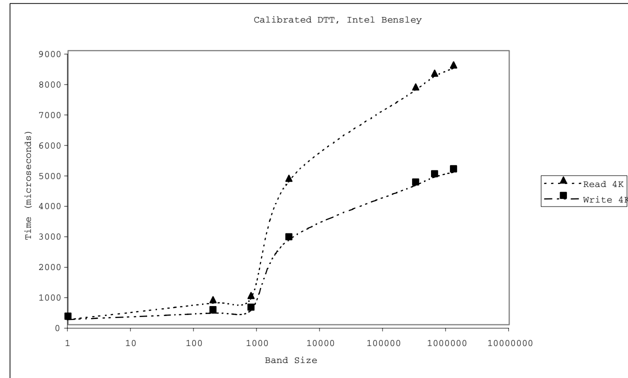
That is, the relationship of expected costs of the two plans is the same as that of the actual costs at run time.

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. The two read curves in Figure 2(a) illustrate better random retrieval times when the band size is smaller. 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.

Note that each write curve in the model DTT (Figure 2(a)) illustrates a lower amortized cost than its corresponding read curve for larger band sizes. On the surface, this is counterintuitive: disk subsystems are typically optimized for read operations, particularly sequential reads [17, pp. 44–7] and write operations usually require addi-



(a) Default DTT model.



(b) Calibrated DTT for an Intel Bensley processor with a single Seagate 7200 RPM 'Barracuda' disk (note the logarithmic scale).

Figure 2. DTT models

tional overhead [4]. The explanation of the DTT curves is that in a database system, if we ignore issues such as prefetching, read requests are typically synchronous; a join algorithm, for example, cannot proceed until the required page is present in the buffer pool. However, writes are typically asynchronous, since it is the buffer pool manager that will decide at what point any dirty pages will be flushed to disk, and these deferred I/O requests can take advantage of parallelism, shortest seek-time scheduling, and other optimizations [7, 17].

If the default DTT model is unsatisfactory, a DBA can substitute a different one. For specialized hardware, a CALIBRATE DATABASE statement can determine the read DTT curve from the actual system. The write DTT curve is approximated using the read curve as a baseline. Figure 2(b) illustrates an actual read DTT and an approximate write DTT taken from a dual-core, multi-CPU Intel Bensley processor with a 'Barracuda' 7200 RPM disk. Calibration can also be important for Windows CE platforms, particularly because persistent storage, often compact flash memory, has substantially different characteristics than disk media. Figure 3 illustrates the DTT curve of a 512 MB SD card on a Pocket PC 2003 handheld device—note the uniform random access times. In SQL Anywhere, the DTT model is stored in the catalog and can be altered or loaded with the execution of a DDL statement. Consequently, it is straightforward to deploy hundreds or thousands of databases to CE devices with a cost model derived from a representative device.

4.3. Adaptive query execution

SQL Anywhere's query optimizer makes choices based on estimates and predictions made at optimization time, but there is necessarily a level of uncertainty in these predictions due to estimation error and concurrent activity. Individual query execution operators are able to detect particular prediction problems and adapt locally to the current conditions.

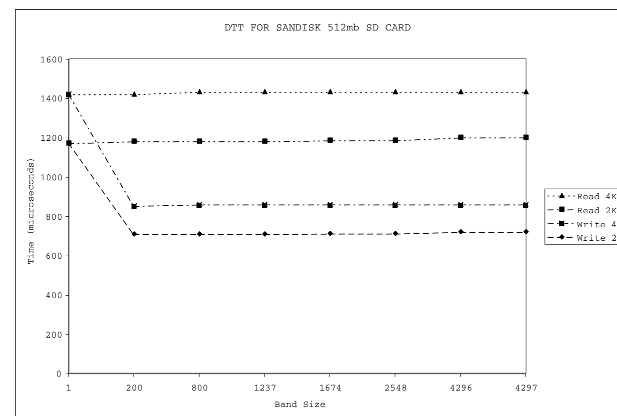


Figure 3. DTT for an SD storage card

One example of this adaptivity is the hash join operator. The optimizer may favour a hash join over an index nested-loops strategy based, in part, on the estimated number of rows in the build input. After building the hash table on the build input, the hash join operator knows the precise number of build rows, and can determine if the optimizer was wrong and an index nested-loops strategy would in fact have been cheaper. The query optimizer annotates the hash join operator with an alternate strategy so that an index nested-loops strategy can be employed after reading the build input if the number of rows is low enough to make it preferable. In a similar way, a special operator for exe-

cution of RECURSIVE UNION is able to switch between several alternative strategies, possibly using a different one for each recursive iteration, and also possibly sharing work from iteration to iteration.

In addition to adapting to errors in cardinality estimation, the SQL Anywhere query execution operators adapt to changes in the amount of buffer pool available and to fluctuations in the number of concurrent, active requests. Each task, or unit-of-work, within the SQL Anywhere server has a memory governor that controls memory usage for both fixed-sized data structures and variable-sized structures used for memory-intensive operators such as sorting or hashing. The memory governor provides two quotas to each task:

- a *hard* memory limit: if exceeded, the statement is terminated with an error. This limit is

$$\frac{(\frac{3}{4}\ maximum\ buffer\ pool\ size)}{number\ of\ active\ requests.} \quad (4)$$

- a *soft* memory limit, which the statement's query processing algorithms should not exceed. When this limit is reached, the memory governor requests query execution operators to free memory. The soft limit is computed as

$$\frac{current\ buffer\ pool\ size}{server\ multiprogramming\ level.} \quad (5)$$

The memory governor controls query execution by limiting memory consumption for a statement to the soft limit. The query optimizer uses the predicted soft limit to estimate execution costs, and annotates memory-intensive operators with a maximum number of pages to use in their processing. Memory-intensive operators include hash join, hash group by, hash distinct, and sort. If the actual memory available at execution time is much lower than the optimizer expected, then a memory-intensive operator may not be able to complete in a reasonable time. Memory-intensive operators detect this condition at execution time, and change locally to a low-memory fallback strategy. For example, the low-memory fallback for hash group by uses a temporary table containing partially computed groups with an index on the grouping columns.

Low-memory fallback strategies are only used in extraordinary cases. In normal operation, the concern is how to share memory effectively between operators within a single statement's execution plan and between concurrent requests [5, 14]. This sharing is accomplished using the memory governor's soft limit and annotations from the query optimizer.

For example, a hash join operation in SQL Anywhere chooses 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 processing of the hash operation, memory usage of the hash table is monitored. When the hash table reaches the memory governor's soft limit, 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 [23]. 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 some cases, multiple memory-intensive operators within a query execution plan may be using memory concurrently. For example, the probe input to a hash join may use memory for hash group by. When the soft limit is reached for the statement, the memory governor begins requesting that memory be relinquished starting at the 'highest' consuming operator and moving down the execution tree. This approach prevents an input operator from being starved for memory by a consumer operator, while still allowing individual operators to proceed with as much memory as possible.

4.4. Adaptive intra-query parallelism

SQL Anywhere can assign multiple threads or fibers to a single request, thus achieving intra-query parallelism. Manegold et al. describe a technique that has very good automatic load balancing characteristics [13]. They construct a right-deep pipeline of hash joins and execute the probe phase with multiple threads. A thread fetches a row in a first-come, first-serve manner from the single scan feeding the pipeline and then executes the probe against all hash tables in the pipeline. Any number of threads can do this simultaneously, independent of the number of joins in the plan. First-come, first-serve partitioning also has the advantage that it preserves a sequential disk access pattern for table scans. This is particularly important for parallelizing queries on SMP or multi-core machines where the database is on a single disk.

We use this technique with several extensions. One is that the build phase of the hash joins is also parallelized. The threads assigned to perform the probe phase are first used to execute the build phase by fetching rows first-come, first-serve from the scans and building separate hash tables. The hash tables are then merged into a single hash table for each join and the probe phase proceeds. Thus, a plan involving an arbitrary number of joins composed in an arbitrary way

(not just right-deep) can be executed with much the same load balancing characteristics on the build phases as on the probe phases. We also extend the technique to support complex plans containing both hash joins and additional operators while maintaining the load balancing characteristics. The operators include nested loop joins, Bloom filters, and hash group bys.

In addition to load balancing, an adaptive advantage of the technique is that the number of threads assigned to a plan can very easily be changed during execution. In particular, if the number of threads is dynamically reduced to one, then the total cost of the query is only slightly worse than if it was never set up to use parallelism. This gives the server a lot of flexibility to adapt gracefully to fluctuations in load. It also makes query optimization relatively easy since costing of a parallel plan can be done as a slight modification of costing the corresponding non-parallel plan.

5. Database design and workload analysis

The goals that drive self-management features on database systems, such as reduced total cost of ownership, also extend to other phases of the design and implementation database applications. Database application design, physical database design, and database setup are phases of the complete database application lifecycle that have traditionally required a level of user expertise similar to that needed to tune an active database system. SQL Anywhere provides an integrated toolset, known as Application Profiling, to advise DBAs and application developers on each of these tasks.

To support this analysis, as much detail as possible is collected about a database application and a database instance. This is accomplished by obtaining 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.

The Application Profiling tool contains a database of commonly seen design flaws. It is able to detect incorrect database option settings. It can also detect suboptimal query patterns coming from an application. For instance, it can detect the presence of a client-side join, in which many identical statements arrive from an application, differing only by some constant value used in a predicate. The tool can then point out to the user that such a loop in the application would be more efficiently carried out as a single statement issued to the server.

In addition to detecting design or implementation problems, the Application Profiling tool also includes an Index Consultant that can recommend the creation or removal of indexes. The Index Consultant uses a novel technique to provide useful recommendations without requiring excessive resources, whereby the query optimizer is able to generate specifications for indexes it would like to have [6]. These 'virtual index' specifications can be very general, allowing flexibility in the composition, sequence, and ordering of index columns. The virtual index specification becomes tighter as optimization proceeds, as the optimizer desires more specific orderings of the input [22]. When the Index Consultant is finished, a physical composition and ordering is imposed on the index. This technique allows the optimizer to consider large numbers of potential physical indexes efficiently.

All of the existing parts of the Application Profiling tool require some intervention from the DBA; in some cases, the DBA is only required to approve or disapprove of a recommendation made (for example, when a recommendation is made to create an index). In other cases, the DBA must make manual changes to the database (for instance, moving the transaction log to a separate disk drive). The fulfillment of the promise of self-management would be to have these recommendations implemented automatically.

6. Future work

Virtually every new feature implemented in SQL Anywhere is designed to be adaptive or self-managing. In addition to the 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 graphical administration and modeling tools, and a full-function stored procedure debugger.

As with any software system, SQL Anywhere continues to evolve and improve its capabilities. In the future we will be studying a variety of opportunities to further improve self-managing performance.

Self-management of database statistics poses unique challenges in some user environments. For example, the inevitable delay, no matter how small, between changes to underlying data and the capture and integration of their corresponding distribution statistics can cause problems in application environments where data distributions change drastically at regular intervals. Moreover, the overhead incurred by the server for automatic statistics collection must be carefully managed: the server must be able to intelligently tradeoff their collection with the potential improvement to query workload performance. We are working on

a new statistics collection framework, that includes feedback from the optimizer, in order to proactively determine desired, unnecessary, or inaccurate statistics.

Other items on our research agenda include:

- dynamically changing the server's multiprogramming level in response to database workload and changes to overall system performance;
- better algorithms for apportioning memory across query execution operators within an access plan (cf. references [5, 14]);
- automatic reclustering and/or reorganization of tables and indexes;
- improvements to buffer pool growth and shrinkage strategies to avoid hysteresis and make server performance more predictable;
- better modeling of write performance on removable media, particularly for Windows CE devices; and
- adaptive priortization and scheduling of I/O requests.

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 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.

[3] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. In ACM SIGMOD *International Conference on Management of Data*, pages 353–364, Montréal, Québec, June 1996.

[4] P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation–self-scaling I/O benchmarks, predicted I/O performance. ACM *Transactions on Computer Systems*, 12(4):308–339, Nov. 1994.

[5] B. Dageville and M. Zait. SQL memory management in Oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 962–973, Hong Kong, China, Aug. 2002.

[6] D. J. Farrar and A. Nica. Database system with methodology for automated determination and selection of optimal indexes. US Patent 2005/0203940, Sept. 2005.

[7] 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.

[8] M. Greenwald. Practical algorithms for self-scaling histograms or better than average data collection. *Performance Evaluation*, 20(2):19–40, June 1996.

[9] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad-hoc join costs. *The* VLDB *Journal*, 6(3):241–256, Aug. 1997.

[10] Y. E. Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 19–30, Berlin, Germany, Sept. 2003.

[11] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 240–251, Los Angeles, California, Aug. 1988.

[12] L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated I/O model. ACM *Transactions on Database Systems*, 14(3):401–424, Sept. 1989.

[13] S. Manegold, J. K. Obermaier, and F. Waas. Load balanced query evaluation in shared-everything environments. In *European Conference on Parallel Processing*, pages 1117–1124, 1997.

[14] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *Proceedings, Seventh International Conference on Information and Knowledge Management* (CIKM), pages 116–123, Bethesda, Maryland, Nov. 1998.

[15] A. Nica. System and methodology for cost-based subquery optimization using a left-deep tree join enumeration algorithm. US Patent 2004/0220923, Nov. 2004.

[16] A. Nica. System and methodology for generating bushy trees using a left-deep tree join enumeration algorithm. US Patent 2004/0006561, Jan. 2004.

[17] D. Shasha and P. Bonnet. *Database Tuning*. Morgan-Kaufmann, San Francisco, California, 2003.

[18] A. J. Smith. Sequentiality and prefetching in database systems. ACM *Transactions on Database Systems*, 3(3):223–247, Sept. 1978.

[19] A. Swami and K. B. Schiefer. Estimating page fetches for index scans with finite LRU buffers. *The* VLDB *Journal*, 4(4):675–701, Oct. 1995.

[20] T.-F. Tsuei, A. N. Packer, and K.-T. Ko. Database buffer size investigation for OLTP workloads. In ACM SIGMOD *International Conference on Management of Data*, pages 112–122, Tucson, Arizona, May 1997.

[21] M. Young-Lai. Database system with methodology for distributing query optimization effort over large search spaces. US Patent 6,807,546, Oct. 2004.

[22] M. Young-Lai and A. Nica. Database system with methodology for generalized order optimization. US Patent 2006/0136368, June 2006.

[23] 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, Austrailia, Aug. 1990.

[24] 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.