Self-Tuning Technology in Microsoft SQL Server

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

1 Introduction

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

2 Physical Database Design Tool

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

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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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

2.1 Functionality of the Index Tuning Wizard

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

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

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

2.2 Architectural Overview

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

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

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

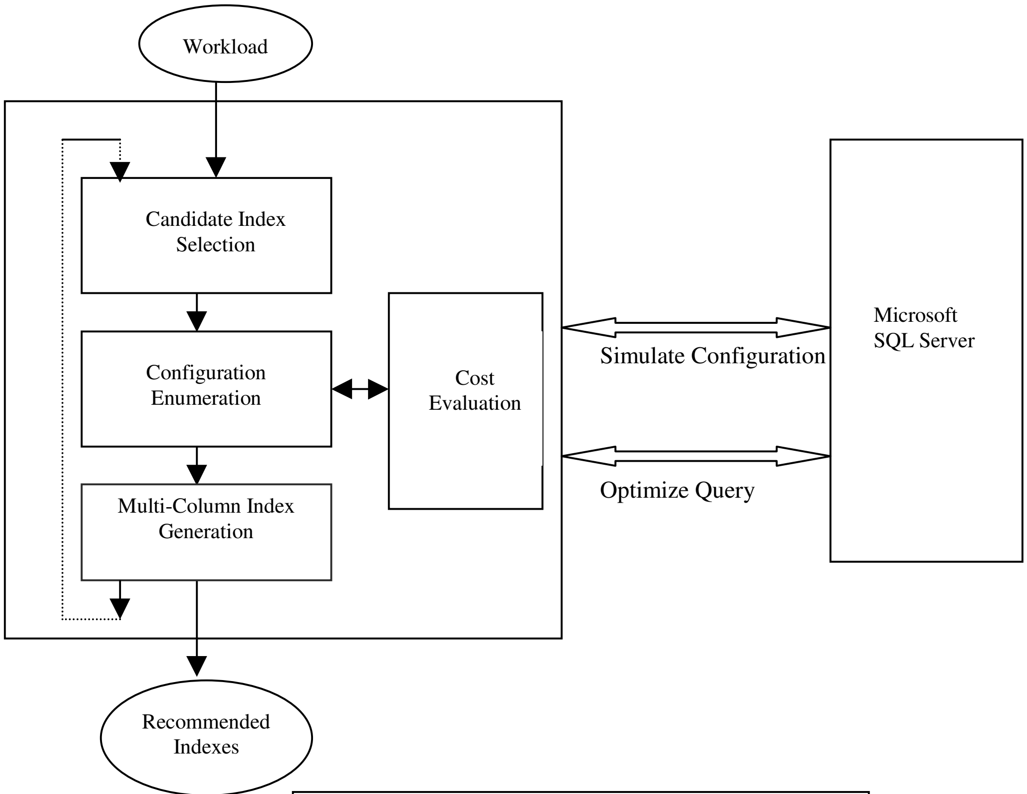

Figure 1. Architecture of Index Tuning Wizard

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

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

3 Relational Engine and Query Processor

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

3.1 Memory Management

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

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

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

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

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

3.2 Distribution statistics

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

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

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

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

4 Storage Engine

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

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

4.1 Dynamic Resource Allocation

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

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

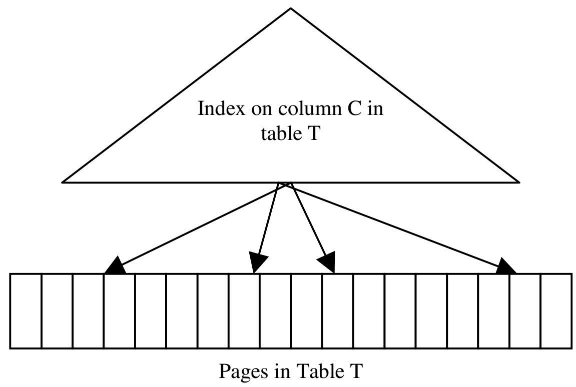

Figure 2. Base Table Access through an Index

4.2 Resource Optimization Strategies

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

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

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

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

4.3 Feedback-based Techniques

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

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

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

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

References

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

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

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

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

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

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

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

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