Managing Query Compilation Memory Consumption to Improve DBMS Throughput

Boris Baryshnikov, Cipri Clinciu, Conor Cunningham, Leo Giakoumakis, Slava Oks, Stefano Stefani

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
{borisb,ciprianc,conorc,leogia,slavao,stefanis}@microsoft.com

Abstract

While there are known performance trade-offs between database page buffer pool and query execution memory allocation policies, little has been written on the impact of query compilation memory use on overall throughput of the database management system (DBMS). We present a new aspect of the query optimization problem and discuss a solution implemented in Microsoft SQL Server 2005. The solution provides stable throughput for a range of workloads even when memory requests outstrip the ability of the hardware to service those requests.

1. Introduction

Memory Management is a critical component of DBMS performance. In modern systems, memory trade-offs are far more complex than the classic problems of database page buffer management or reserving memory for hashes and sorts during query execution [5]. Current systems use more ad-hoc queries that make query compilation memory more important in memory reservation policies. Furthermore, these ad-hoc deployments make it harder to provision hardware, and thus they are more often run at or beyond the capabilities of the underlying hardware. This requires intelligent trade-offs among other memory consumers in a DBMS, as every byte consumed in query compilation, query plan caches, or other components effectively reduces the available memory for caching data pages or executing queries. As DBMS become more complex, it becomes harder to reason about the impact of memory use on overall throughput effectively.

3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007, Asilomar, California, USA.

In our research we identified compile-intensive ad-hoc workloads that consume enough memory in query compilation to disturb the traditional memory consumption trade-offs between a database page buffer pool and query execution. For these scenarios, overall system throughput was significantly reduced due to memory thrashing among components. Oftentimes, the queries in question required so much memory to perform compilation that other components were effectively starved and no other work could proceed. Even if the system has enough memory to service multiple simultaneous query compilations, allowing all of them to occur at the same time might not maximize throughput. Excessive concurrent compilation memory usage steals a significant number of pages from the database page buffer pool and causes increased physical I/O, reduces memory for efficient query execution, and causes excessive eviction of compiled plans from the plan cache (forcing additional compilation CPU load in the future). The interplay of these memory consumers is complex and has not been adequately studied.

In this paper we present a solution to the above problem by providing a robust query compilation planning mechanism that handles diverse classes of workloads, prevents memory starvation due to many concurrent query compilations, and dynamically adjusts the load to make intelligent memory trade-offs for multiple DBMS subcomponents that improve overall system reliability and throughput. This solution has been implemented and tested against Microsoft SQL Server 2005 and is part of the final product.

2. Memory Influence on Performance

While a number of papers have addressed trade-offs between buffer and query execution memory ([2], [5]), no work has covered the impact of memory consumption from query *compilation* on system throughput. In this paper, we demonstrate that query compilation can impact system performance through its memory consumption for

large workloads and that steps can be taken to mitigate the negative impacts of memory starvation on system throughput. As many DBMS installations run on dedicated hardware, the rest of this paper will assume, for simplicity, that almost all physical memory is available to the DBMS and that it is the only significant memory-consuming process being run on the hardware.

2.1 DBMS Subcomponent Memory Use

Multiple subcomponents in a DBMS consume memory as part of normal operations. In addition to the obvious consumers of memory such as buffer pools and query execution memory, modern DBMS have multiple kinds of caches and very complex queries that are difficult to optimize. As the number of memory-consuming subcomponents within a DBMS has increased over time, the policies used to manage their interactions can be difficult to tune. For example, if a new query attempts to allocate memory to execute, it may need to steal memory from another subcomponent. Such subcomponents might be the buffer pool, the compiled plan cache, or other caches (such as over metadata or cached security tokens). Since a server DBMS is often under memory pressure, these implementation policy choices can significantly impact overall system performance.

Within a DBMS, it is possible for a subcomponent to respond to memory pressure by releasing unneeded or "less valuable" memory. Caches can often be freed and repopulated later, and buffer pool memory works similarly. However, other subcomponents may or may not be architected to quickly respond to memory pressure. When such approaches are insufficient to meet memory needs, the server reaches a "memory deadlock", and some user operation is terminated to reduce memory pressure and allow others to proceed. Each DBMS subcomponent uses memory differently, and this can impact the heuristics and policies required to maximize overall performance. For example, a database page buffer pool caches disk pages for more efficient access to data. The buffer pool would prefer to consume all available free memory that is not in use elsewhere to cache pages, as this has the potential to save future disk I/O costs. Query execution uses memory to speed sorts, spools, and hash operations. This component can often reserve memory at the start of a query execution for its operations, spilling to disk if its reservation is too small. Modern DBMS also can have a number of caches for compiled plans, and the size of each of these caches must be valued against the use of that memory for caching data pages or improving query execution performance. Often, the local code for memory allocation in each subcomponent is not written to be aware that memory will likely come from another subcomponent or to recognize the value of that memory use compared to the current memory request. Poor consideration of the value of the memory utilized in each subcomponent can cause a heavily-loaded system to become unbalanced and perform sub-optimally.

Query compilation (and, more specifically, query optimization) consumes memory differently than other DBMS subcomponents. Many modern optimizers consider a number of functionally equivalent alternatives and choose the final plan based on an estimated cost function. This process uses memory to store the different alternatives for the duration of optimization. Some optimizers contain memory-saving structures (such as the Memo in the Cascades Framework [4] used in Microsoft SQL Server) to detect duplicates and avoid exploring any portion of the query plan space more than once. Even with such structures, the memory consumed during optimization is closely related to the number of considered alternatives. For an arbitrary query, the total memory consumed during optimization may be large and hard to predict. This makes it difficult to understand memory consumption performance trade-offs in relation to other DBMS subcomponents. While work has been done on dynamic query optimization, where the number of considered alternatives (and thus the amount of memory consumed) is related to the estimated cost function for the query, no work, to our knowledge, has been done on the value of consuming more memory optimizing a query in a memory-constrained workload.

2.2 DBMS Design vs. System Throughput

Modern x86-based systems have 32-bit address spaces, and this poses limits on the available size of the memory available to a DBMS when implemented within a single process. The common-case on the Windows 2003 Server operating system limits a process to 2GB of virtual address space (3GB is possible with restrictions). Of this, roughly a quarter is needed for process-management tasks such as thread stacks, mapping DLLs into the process, etc. While techniques ([8]) exist to allow some subcomponents, such as the buffer pool, to remap pages in and out of the address space, this is generally difficult to use for complex structures with pointers. In effect, this places a practical limit for a server process about 1.5GB of memory.

The combination of limited memory/virtual address space and a diverse set of memory consumers poses challenges to the DBMS implementer when trying to guarantee good system throughput[1]. A naïve approach of placing caps on

[1] In DBMS where separate processes are used for each subcomponent, memory trade-off choices between/among components are just being performed by the operating system instead of the DBMS implementer. While we have not performed experiments on such a configuration, we expect that there would be opportunities to improve performance if hints or other techniques were used to influence the operating system scheduling/page allocation decisions.

each memory subcomponent to avoid memory starvation does not always work due to the varied nature of workloads and the inability of the system to plan for work in the future without additional guidance from the user. As an example, if one limits the amount of memory any single query can use to execute to 50% of the total available memory, then no single query can monopolize all system resources for a long period of time. Unfortunately, if the workload only consists of one single, long-running query, the system does not perform as well as it might if that query could consume all memory during its execution.

Even within a subcomponent, memory allocation policies can be difficult to tune to achieve system stability. For example, if many large queries are compiling simultaneously, each compilation can consume a significant fraction of system memory. Two query compilations can deadlock each other if both are waiting for memory consumed by another compilation. Even if the system aborts most of these queries to allow a few to complete, those aborted queries may be resubmitted to the system by client code.

A central controlling mechanism is necessary to ensure that the overall system will perform well and be stable in all situations. Making proper decisions to receive, evaluate, and arbitrate requests for memory amongst multiple consumers can improve system stability and increase throughput, even when the system is running at or beyond the capabilities of the hardware. This approach is described in more detail in the following section.

3. Memory Broker

SQL Server 2005 uses a "Memory Broker" to manage the physical memory allocated to DBMS subcomponents. The broker accounts for the memory allocated by each subcomponent, recognizes trends in allocation patterns, and provides the mechanisms to enforce policies for resolving contention both within and among subcomponents. This subcomponent enables a DBMS to make better global decisions about how to manage memory and ultimately achieve improved throughput. The details of the Memory Broker are beyond the scope of this paper, but we provide an overview of the component to place the rest of our solution in context for the reader.

The Memory Broker monitors the total memory usage of each subcomponent and predicts future memory usage by identifying trends. If the system is not using all available physical memory, no action is taken and the system behaves as if the Memory Broker were not there. If the future memory total is expected to exceed the available physical memory, the broker predicts the actual amount of physical memory that the subcomponent should be able to allocate (accounting for requests from other subcomponents). The broker also sends notifications to each subcomponent with its predicted and target memory numbers and informs that subcomponent whether it can continue to consume memory, whether it can safely allocate at its current rate, or whether it needs to release memory. The computation of target memory amounts and the notification of each subcomponent can happen several times a second. In our implementation, the overhead of this mechanism is extremely small. It is still possible to have out-of-memory errors if many subcomponents attempt to grow simultaneously. The system relies on the ability of various subcomponents to make intelligent decisions about the value of optional memory allocations, free unneeded or low-value memory, and reduce the rate of memory allocations over time.

The Memory Broker provides an indirect communication channel for one subcomponent to learn about the overall memory pressure on the system. It also helps to mitigate "wild" swings in subcomponent memory allocations and tends to make the overall DBMS behave more reliably by reducing probability of aborting long-running operations such as compiling and/or executing a query.

DBMS subcomponents impose different requirements on a memory subsystem through their usage patterns that can impact how the Memory Broker operates. For example, the database page buffer pool contains data pages that have been loaded from disk. Replacement policies can be used to remove older pages to load currently needed pages, but they can also be used to enable the buffer pool to identify candidates necessary to shrink its size. The value of each page in memory is a function of the simple replacement cost (i.e. a random I/O to re-read the page) plus some value based on the probability that this page will be used again in the near future. Other caches can support shrinking using the same technique.

The memory consumed during query execution is usually predictable since many of the largest allocations can be made using early, high-level decisions at the start of the execution of a query. For example, the size of a hash table can often be predicted based on by-products of the cardinality estimates used in a cost-based optimization process, and this memory can be allocated once for the whole hash table at the start of a query execution. Unlike caches, however, the execution of queries may require that memory be allocated for the duration of the query. Therefore, the subcomponent may be less capable to respond to memory pressure from a Memory Broker at any specific time. However, it can potentially respond to memory pressure based on the shape of the query and the relative position of the operators being executed. A "stop-and-go" operator (where all input rows are consumed before output rows are returned), such as a sort, can be used to identify sections of a query plan that are

not needed after the sort is materialized. Often memory allocations for a query execution are batched to avoid the overhead of many, small allocation and deallocation calls, so there is a trade-off that must be made to find the right balance between returning memory to the system early or efficiently. Optional caches and spools in a query plan is another area where dynamic memory choices could be made based on memory load. These operators can be written to make them use memory dynamically based on what is available to the system at the time that they are run. Other techniques, such as queuing execution requests until memory can be reserved, can also be used to limit the rates at which memory is consumed by this subcomponent. The introduction of memory subcomponent prediction, memory targets, and notifications enable subcomponents to react to memory pressure dynamically and proactively before out-of-memory conditions are required.

Query compilation also uses memory in ways interesting to a Memory Broker component, and this is discussed in detail in the next section.

4. Query Compilation Throttling

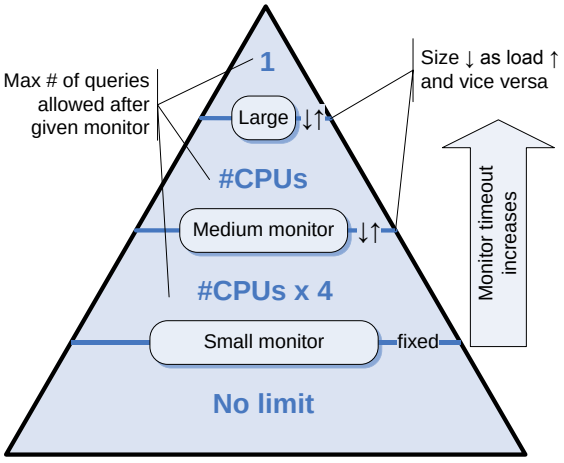
Query compilation consumes memory as a function of both the size of the query tree structure and number of alternatives considered. Beyond dynamic optimization, which has traditionally been based on estimated query runtime and not memory needs, there are no published techniques to avoid memory use during query compilation for standard approaches.

Our analysis of actual compile-intensive workloads showed that high memory consumption is typically caused by several medium/large concurrent ad hoc compilations in a workload instead of one or few very large queries. This makes intuitive sense, as DBMS users probably realize when they are writing a large query and may take steps to isolate it. Many smaller queries pose a more significant challenge, as each query appears reasonable to the author. While it may not be easy (or desirable) to modify the main optimization algorithm to account for memory pressure, it is possible to change the *rate* at which concurrent optimizations proceed to respond to memory pressure. In this section, we describe a query compilation planning mechanism that handles multiple classes of workload goals, dynamically adjusts to system memory pressure, and interacts with the dynamic programming algorithms used in many modern optimizers to make intelligent decisions about memory use during the compilation process. This system improves overall system throughput and reduces resource errors returned to clients when the system is under memory pressure.

4.1 Solution Overview

We propose a query compilation throttling solution that responds to memory pressure by changing the rate at which compilations proceed. If we assume that memory use roughly grows with compilation time, throttling at least some compilations restricts the overall memory usage by the query optimization subcomponent and can improve the system throughput. Blocked compilations wait for resources to become available before continuing. If the compilation of a query remains blocked for an excessively long period of time, its transaction is aborted with a "timeout" error returned to the client. Properly tuned, this approach allows the DBMS implementer to achieve a balance between out-of-memory errors and throttle-induced timeouts for a variety of workloads. Our approach gives preference to compilations that have made the most progress and avoids many cases where a compilation is aborted after completing most, but not all, of the compilation process.

Figure 1 Memory Monitors

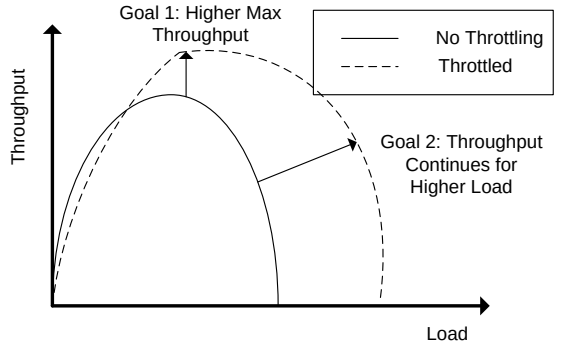


Blocking is implemented through a series of monitors that are acquired during compilation. The blocking is tied to the amount of memory allocated by the task instead of specific points during the query compilation process. As different optimization alternatives and techniques may consume different amounts of memory, this approach handles both differing optimization techniques and remains stable as code changes over multiple software releases. This provides a more robust mechanism to control the impact of compilation on overall system memory load over a wide variety of schema designs and workload categories. These monitors contain progressively higher memory thresholds and progressively lower limits on the number of allowed concurrent compilations as illustrated in Figure 1. The monitors are acquired sequentially by a compilation as memory usage for that task increases and are released in reverse order if memory use decreases during compilation or at the end of the compilation process. If memory is not

available at the time of acquisition, the compilation process is blocked until memory becomes available when other compilations, executions, or memory requests elsewhere in the system are completed. A timeout mechanism is used (with increasing timeouts for later monitors) to return an error to the user if the system is so overloaded that a compilation does not make any progress for a long period of time.

Restraining compilations effectively avoids some cases where many simultaneous compilations consume a disproportionately high fraction of the available physical memory. Since memory is a scarce resource, preserving some fraction of it for use by the database page buffer pool and query execution allows these components to more efficiently perform their functions. Blocking some queries can reduce the need for other subcomponents to return memory from caches if many large, concurrent compilations occur. This can spread memory use over time instead of requiring other subcomponents to release memory. The intended goals of this approach are to improve maximum throughput and to enable that throughput to work for larger client loads on the system, as outlined in Figure 2.

Figure 2 Expected Throttling Results



Our implementation uses three monitors. Experimental analysis showed that dividing query compilations into four memory usage categories balanced the need to handle different classes of workloads and limiting the compilation time overhead of the mechanism. Query compilations that consume less memory than the first monitor threshold proceed unblocked. The first threshold is configured differently for each supported platform architecture to allow a series of small diagnostic queries to proceed without acquisition of the first (smallest) monitor. This enables an administrator to run diagnostic queries even if the system is overloaded with queries consuming every available 'slot' in the memory monitors. The first monitor allows four concurrent compilations per CPU and is used to govern "small" queries. Typically, most OLTP-class queries would fall into this category. The second monitor is required for larger queries, allowing one per CPU. Many TPC-H queries [6], which require the consideration of many alternatives, would be in this category. The final governs the largest queries and allows only one at a time to proceed. This class of query uses a sizable fraction of total available memory on the system. The largest memory-consuming queries are serialized to avoid starvation of other subcomponents and allow the query to complete compilation. This approach allows us to restrict compilation, in most cases, to a reasonable faction of total memory and allow other subcomponents to acquire memory.

Figure 3 Compilation Throttling Example

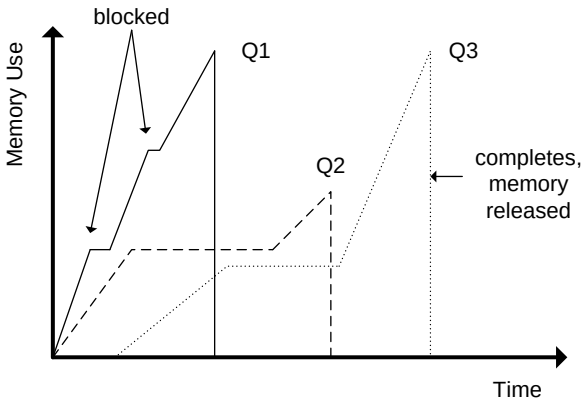


Figure 3 contains a simplified example to describe how query compilation throttling might work in practice. In this example, Q1 and Q2 start compiling at approximately the same time. However, Q1 consumes memory at a faster rate than Q2. This could occur if the query was larger, the tables contained more complex schemas, or perhaps that thread of control received more time to execute. Q1 then blocks at the first monitor, as denoted by the first flat portion of the graph of Q1's memory use. This occurred because other queries (not shown in the example) were consuming enough resources to induce throttling. Once enough memory is available, Q1 continues, blocks again at the next monitor, eventually is allowed to continue, and finally finishes compilation. At the end of compilation, memory used in the process is freed and the query plan is ready for execution. Q2 executes in a similar manner. It waits much longer to acquire the first monitor (meaning that the system is under more memory pressure or that other compilations are concurrently executing and using memory). Q2 finishes and frees memory, but it did not require as much memory as Q1 to compile. In this example, Q3 is actually blocked by Q2 and only proceeds once Q2 is finished and releases its resources. From the perspective of the subcomponent author, the only perceptible difference in this process from a traditional, unblocked system is that the thread sometimes receives less time for its work. The individual thread scheduling choices are made by the system based on global goals that effectively prioritize earlier over later compiles when making scheduling (and thus allocation) decisions.

4.2 Extensions

We have extended this approach with two novel extensions. First, we have made the monitor memory thresholds for the larger gateways dynamic. In our experiments, some customer workloads perform best with different gateway values. In other words, the relative balance of optimal subcomponent memory use is not constant across all workloads. For example, one workload may use a fixed set of queries that rarely need to be compiled, while another workload may only use dynamic, ad-hoc SQL statements. The impact from these workloads on compilation memory would be very different.

Our solution acts upon this realization by leveraging the reported "target" memory consumption level for the query subcomponent. This target is the desired allocation level reported by the broker to each subcomponent, and it is a reflection of the memory usage activity of each subcomponent over time. This allows the query sub-system to throttle compilation memory more aggressively when other subcomponents are heavily using memory, making the system even more responsive to memory pressure. The thresholds are computed attempting to divide the overall query compilation target memory across the categories identified by the monitors. For example, the second monitor threshold is computed as [target] * F / S, where F and S are respectively the fraction of the target allotted to and the current number of small query compilations. In other words, small queries together can consume up the F fraction of the target, after which the top memory consumers are forced to upgrade to the medium category. The values of the F fractions were identified with a long process of tuning and experimentation against several actual workloads. Dynamic adjustment of the monitor thresholds gives additional ability to control memory during query compilation.

Another extension to our solution leverages the notification mechanisms to determine that the system will likely run out of memory before an individual compilation completes. When this happens, we can return the best plan from the set of already explored plans instead of simply returning out-of-memory errors. As not all query plans are efficient, we reserve this approach to very expensive queries that consume a relatively large fraction of total available memory during compilation. While there is no strict guarantee that picking a sub-optimal plan will be better than an out-of-memory error for a large query, in practice we found that this often would improve overall throughput. By restricting this to late in the optimization process, it is more likely that at least one reasonable plan has been found already because most of the search space has been explored. Additionally, modern dynamic optimizers may place more speculative optimization alternatives late in the process, so skipping these may not hurt average query plan quality.

Both techniques allow the system to better handle low-memory conditions.

5. Experimental Results

Standard database benchmarks (TPC-H, TPC-C [6]) contain queries with moderate or small memory requirements to compile. Large decision support systems run queries with much higher complexity and resource requirements. To evaluate our solution, we developed a performance benchmark based on a product sales analysis application created by a SQL Server 2005 customer. For the purposes of this paper, we will refer to that benchmark as the SALES benchmark.

5.1 SALES Benchmark

The SALES application is a Decision Support System (DSS) which uses a large data warehouse to store data from product sales across the world. This application submits almost exclusively ad-hoc queries over significant fractions of the data. Many users can submit queries simultaneously. The customer runs a number of large-CPU systems over read-only copies of their database at or near capacity to handle their user query load due to their unpredictable, ad-hoc workload.

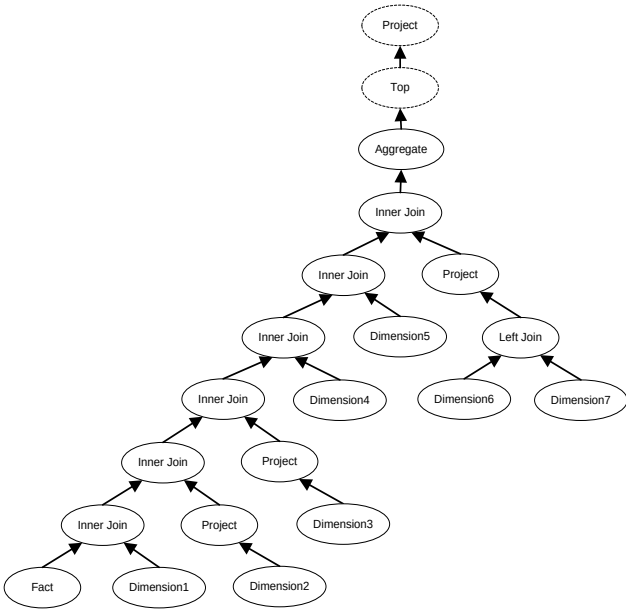
The SALES benchmark uses a somewhat typical data warehouse schema, meaning that it has a large fact table and a number of smaller dimension tables. The largest fact table from the database contains over 400 million rows. An "average" query in this benchmark contains between 15 and 20 joins and computes aggregate(s) on the join results. As a comparison, TPC-H queries contain between 0 and 8 joins with similar numbers of indexes per table. The data mart in our experiments contains a snapshot of the data from the customer's application and is 524 GB in size.

We executed this benchmark against SQL Server 2005. It features dynamic optimization, meaning that the time spent optimizing a query is a function of the estimated cost of the query. Therefore, more expensive queries receive more optimization time. In our experiments, the queries in the SALES benchmark use one to two orders of magnitude more memory than TPC-H queries of similar scale.

Our benchmark models the basic functionality of the application and contains 10 complex queries that are representative of the workload. To simulate the large number of unique query compilations, our load generator modifies each base query before it is submitted to the database server to make it appear unique [7] and to defeat plan-caching features in the DBMS.

This is an example of a typical query tree in the SALES Benchmark:

Figure 4 Typical SALES Query Tree



In the benchmark, we define a limit for the response time of each query based on the original customer requirements. Benchmark runs which violate these response time limits are considered invalid.

5.2 Execution Environment/Results

We execute the SALES benchmark using a custom load generator which simulates a number of concurrent database users who submit queries to the database server. For these experiments, we use a server with 8 Intel Xeon (32-bit) 700 MHz x86-based processors and 4GB of main memory. The server is using 8 SCSI-II 72GB disks configured in as a single RAID-0 drive array on a 2-channel, 160 MB/channel Ultra3/Ultra2 SCSI controller. The software on the machine is Microsoft Windows 2003 Enterprise Edition SP 1 and Microsoft SQL Server 2005. This system is a typical medium server installation and should reasonably reflect the hardware on which scaling problems are currently being seen in DBMS installations today.

Queries in this benchmark generally compile for 10-90 seconds and execute for 30 seconds to 10 minutes. In each subcomponent, these queries consume nontrivial amounts of memory to compile and execute. They also access large fractions of the database and thus put pressure on the database page buffer pool. Therefore, these subcomponents are actively competing for memory during the execution of this benchmark. Failed queries are retried up to 15 times by the client application to mimic the customer behavior. The probability that a query will be aborted due to memory shortages is high, and the cost of each failure is also high (as the work will be retried). This places a high value on biasing resource use towards those operations likely to succeed on the first attempt.

Our experiments measure the throughput and reliability of the DBMS while running both *at and beyond* the capabilities of the hardware. "Throughput" in this context means the number of queries successfully completed per unit of time. Through experimentation, we determined that this benchmark produces maximum throughput with 30 clients on this hardware configuration. Throughput is reduced with fewer users. Increasing the number of users beyond 30 saturates the server and causes some operations to fail due to resource limitations. To measure the effect of running the system under memory pressure, we performed experiments using 30, 35, and 40 clients.

The benchmark imposes extreme loads on the server, and it takes some time for the various structures in each subcomponent to warm up and become stable enough to measure results. The results presented in this section do not include this warm-up period and the data starts at an intermediate time index. There is some fluctuation in the numbers reported because of the different sizes of the queries being executed and the non-deterministic interplay of a number of different clients attempting to proceed at once in a complex system. Experiments were run multiple times, and the results were repeatable for all types of runs presented.

5.2.1 Throughput Results

Figure 5 presents throughput results for the query workload for 30 clients. For each graph, the darker line with diamond points represents the results when throttling was enabled. The lighter line with square points represents the non-throttled data. The points represent the number of successful query completions since the last point in time.

Throttling improves overall throughput by approximately 35% for the 30 client case, allowing a sustained completion of 30-40 queries per time slice in the benchmark. Un-throttled compilations in this benchmark will consume most available memory on the machine and starve query execution memory and the buffer pool. Throttling also helps the 35 and 40 client cases. As visible in Figure 6 and Figure 7, the throughput is lower in each of these cases when compared to the 30 client case. However, throttling still improves throughput for a given number of clients for each of these client loads.

Figure 5 Throughput - 30 clients

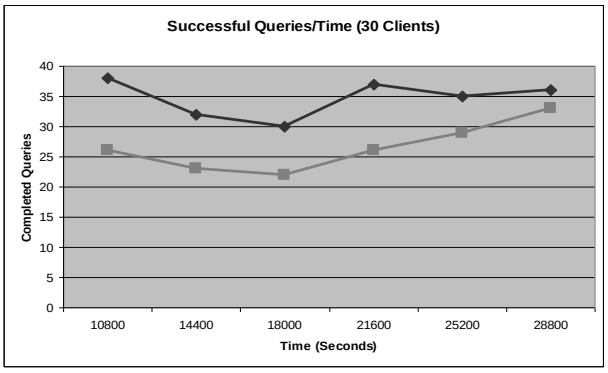


Figure 6 Throughput - 35 clients

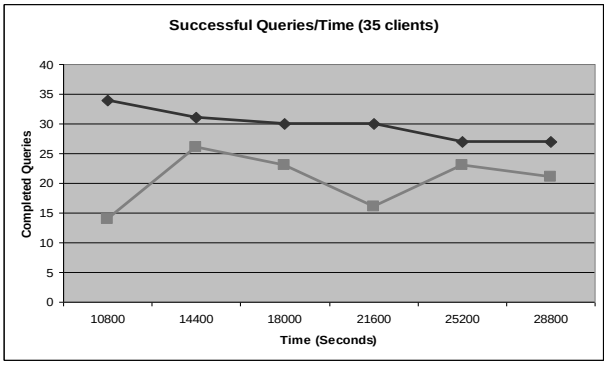
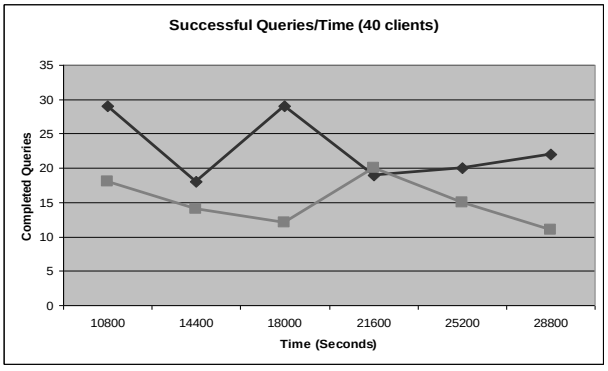


Figure 7 Throughput - 40 clients



Since the data volumes are very large in this benchmark, almost every complex execution operation is performed via hashing. Therefore, each query execution is bound by the maximum size of these hash tables and the CPU work required to build and probe these structures.

5.2.2 Reliability Results

We also measured the percentage of successful query attempts in the system. This is a metric of the probability of any given query compilation/execution attempt will succeed. Retries are counted as separate submissions in this approach. As was the case in the previous section, the darker diamond line represents the results when throttling is enabled, while the lighter line with square points represents the results when throttling is not enabled. The graphs shown in Figures 8-10 represent the 30, 35, and 40 client runs seen in the previous section.

Figure 8 Reliability - 30 clients

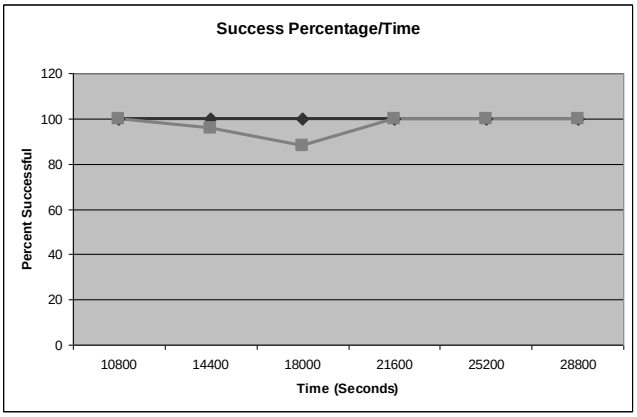


Figure 9 Reliability - 35 clients

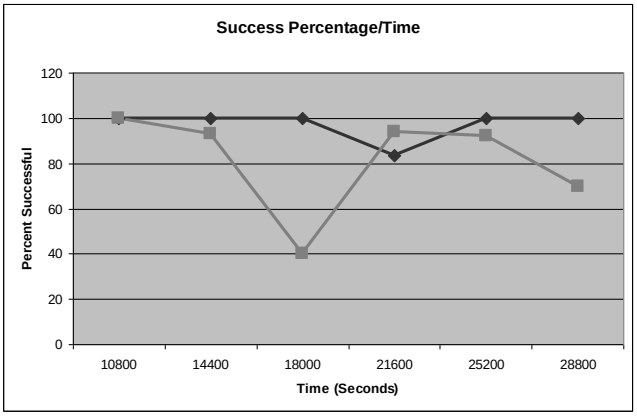
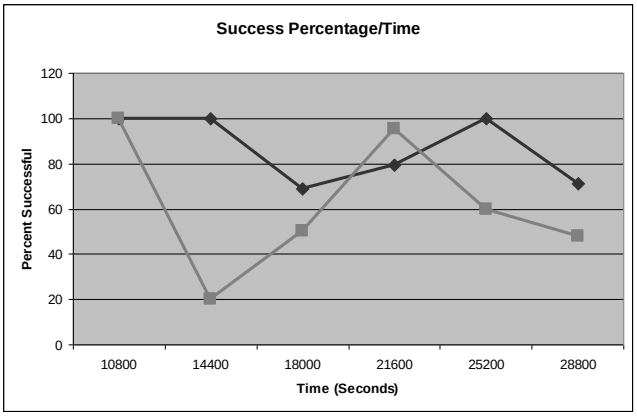


Figure 10 Reliability - 40 clients



The 30 client run demonstrates that both versions complete almost all operations without error once the system has reached a steady state. The non-throttled version has an occasional but infrequent error, and this shows that 30 clients represents the limit for the non-throttled approach on this hardware configuration. Interestingly, the throughput numbers are still higher in the throttled code. This leads us to believe that the errors

are not the significant reason for the difference in performance, at least in the 30 client case. We conclude that this is likely related to resource allocations such as memory.

As additional clients are added to the system, the stress on the system is increased and the probability of any individual query failing also increases (either in compilation or execution). We can see this in the results shown in Figure 9 and Figure 10. The percentage of failures starts to increase as the system aborts operations to free memory. The system also starts to behave less reliably once we increase the load substantially. Figure 10 is representative of the issue – over time, the system may abort no queries or most queries as it tries to service the requests. However, throttling does improve the odds that a query is completed successfully, even under more extreme memory loads. We conclude that this approach helps in achieving the goals outlined in Figure 2 by improving throughput for regular operations and allowing the system to maintain some of these gains over more extreme loads for this class of application.

6. Related Work

Much work has been done on database page buffer pool allocation/replacement strategies and execution/buffer pool trade-offs, however neither of these works specifically address compilation memory or memory from other DBMS caches. [2] and [5] are representative of the field. [5] discusses the trade-offs associated with query execution memory and buffer pool memory. [2] covers different execution classes for different kinds of queries and fairness across queries. [3] discusses the integration of cache state into query optimization. [1] covers the concept of cross-query plan fragment sharing.

7. Conclusions

We introduce a new form of memory/performance trade-off related to many concurrent query compilations and determine that using excessive amounts of memory in a DBMS subcomponent can impact overall system performance. By making incremental memory allocations more "expensive", we can introduce a notion of cost for each DBMS subcomponent that enables more intelligent heuristics and trade-offs to improve overall system performance. Our approach utilizes a series of monitors that restrict the future memory allocations of query compilations, effectively slowing their progress.

In our experiments, we demonstrate that throttling query compilations can improve overall system throughput by restricting compilation memory use to a smaller fraction of overall memory, even in ad-hoc workloads. This improves overall throughput and increases service reliability, even under loads beyond the capability of the hardware. In our experiments, we were able to improve system throughput by 35%.

8. References

[1] Mehta, M., Soloviev, V., and DeWitt, D. Batch Scheduling in Parallel Database Systems. *Proceedings of the Ninth International Conference on Data Engineering (ICDE)* 1993. 400-410.

[2] Mehta, M. and DeWitt, D. Dynamic Memory Allocation for Multiple Query Workloads. *Proceedings of the Nineteenth International Conference on Very Large Data Bases (VLDB)* 1993.

[3] Cornell, D. and Yu, P. Integration of Buffer Management and Query Optimization In Relational Database Environment. *Proceedings of the Fifteenth International on Very Large Data Bases (VLDB) 1989* . 247-256.

[4] Graefe, G. The Cascades Framework for Query Optimization. Data Engineering Bulletin 18 (3) 1995. 19-29.

[5] Brown, K., Carey, M., and Livney, M. Managing Memory to Meet Multiclass Workload Response Time Goals. *Proceedings of the 19th Conference on Very Large Data Bases (VLDB) 1993.*

[6] Transaction Processing Performance Council. http://www.tpc.org

[7] Gray, J. (Ed.). The Benchmark Handbook for Database and Transaction Processing Systems, 1991. Morgan Kaufmann Publishers San Mateo, CA, USA. Chapter 11, p12-15

[8] Address Windowing Extensions and Microsoft Windows 2000 Datacenter Server. March 30, 1999. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/awewindata.asp