Dynamic Memory Adjustment for External Mergesort

Weiye Zhang
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
w2zhang@bluebox.uwaterloo.ca

Per-Åke Larson
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399, USA
palarson@microsoft.com

Abstract

Sorting is a memory intensive operation whose performance is greatly affected by the amount of memory available as work space. When the input size is unknown or available memory space varies, static memory allocation either wastes memory space or fails to make full use of memory to speed up sorting. This paper presents a method for run-time adjustment of in-memory work space for external mergesort and a policy for allocating memory among concurrent, competing sorts. Experimental results confirm that the new method enables sorts to adapt their memory usage gracefully to the actual input size and available memory space. When multiple sorts compete for memory resources, we found that sort throughput and response time are improved significantly by our policy for memory allocation combined with limiting the number of sorts processed concurrently.

1 Introduction

Sorts and joins are memory intensive operations whose performance is greatly affected by the amount of main memory work space available. Increasing the work space reduces the amount of intermediate data transferred between main memory and disk. More importantly, if enough memory is available, a sort or join can be done entirely in memory.

Many systems rely on static allocation, that is, work space is allocated when an operation starts and remains unchanged until it finishes. This approach is likely to result in some operations wasting memory while others are starved for memory. There are two reasons why this may happen. First, the input size may be unknown or poorly estimated. Second, in a multiuser environment the workload varies, resulting in varying demands on the available memory. Overall performance can be improved by algorithms that enable operations to adjust their memory usage at run time in response to the actual size of their inputs and fluctuations in total memory demand.

Sorting is a frequent operation in database systems. It is used not only to produce sorted output, but also in many sort-based algorithms, such as grouping with aggregation, duplicate removal, sort-merge join and set operations [Gra93]. Sorting can also improve the efficiency of algorithms like nested-loop joins and row retrieval via an index. This paper concentrates on dynamic memory adjustment for sorting but the same approach can be applied to other memory intensive operations.

Pang, Carey and Livny [PCL93a] first studied dynamic memory adjustment for sorting and proposed memory adjustment strategies for external mergesort. For the run formation phase, they considered quicksort and replacement selection. When using quicksort, adjustments can only be done when a run has been finished and output. When using replacement selection, memory adjustments can be done by expanding or shrinking the selection heap. For the merge phase, they studied memory adjustment policies that change merge patterns between merge passes. Their work concentrated on sorts with multiple merge passes and did not consider the effects of several sorts running concurrently.

We believe that the greatest performance improve-

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

ment can be achieved by completing as many sorts as possible in main memory. We propose adjustment mechanisms and a policy that attempt to achieve this. Our policy dynamically adjusts work space size not only in response to the actual input size but also in response to changes in available memory and competing demands from other sort jobs running concurrently.

The rest of this paper is organized as follows. Section 2 describes our memory-adaptive external mergesort. Experimental results are presented in Section 3, comparing the performance of our memory-adaptive sort and a sort with static memory allocation. Section 4 summarizes our findings and offers some conclusions.

2 Dynamic Memory Adjustment for Mergesort

We first outline a memory adaptive version of (external) mergesort, followed by a more detailed discussion of when and how memory usage can be adjusted in the various sort phases. We then discuss adjustment policies and describe the policy we adopted.

2.1 A Memory Adaptive Mergesort

External mergesort consists of two phases: a run formation phase and a merge phase. The standard algorithms for run formation are quicksort and replacement selection [Knu73]. Processor speeds continue to increase faster than memory speeds causing an algorithm's cache behavior to become increasingly important. Following [NBC+94] we therefore opted for a two-phase algorithm for run formation which first sorts data within buffers, followed by an in-memory merge. When a sort cannot be completed entirely in memory, the in-memory merge produces runs and external merging is required. Note that runs may be of variable length because work space size may change between runs.

There are many valid merge patterns; the only requirement is that each merge step must reduce the number of runs so that we eventually end up with a single, completely sorted run. So given S initial runs, possibly of variable length, and a maximum merge fan-in of K, which merge pattern results in the minimum data transmission? Under the assumption that K remains fixed, this problem has a surprisingly simple solution (see [Knu73], pp 365-366): first add enough dummy runs of length zero to make the number of runs minus one divisible by $K-1$ and then repeatedly merge together the K shortest remaining runs until only one run remains.

Unfortunately, we cannot apply this solution directly because we cannot guarantee that K remains fixed throughout the merge phase; each merge step may have a different fan-in. However, we retain part of the solution: once the fan-in for a merge step has been determined (depending on available memory) we always merge the smallest remaining runs.

In summary, our variant of mergesort has three phases: an *in-buffer sort* phase which sorts data within a buffer, an *in-memory merge* phase which produces runs by merging sorted buffers, and an *external merge* phase which merges sorted runs. The algorithm is outlined below. We have also indicated at which points in the algorithm we check and possibly adjust the work space size.

```
/* In-Buffer Sort Phase */
while there is more input & memory space
  read data into a buffer
  sort the buffer
  [check/adjust memory]
endloop
/* In-Memory Merge Phase */
if no more input & this is the first run
  merge buffers to produce output and stop
if no more memory or this is the last run
  merge buffers
  write the sorted data into a tmp table
  if there is more input
    [check/adjust memory]
    go to In-Buffer Sort Phase
/* External Merge Phase */
[check/adjust memory]
while max merge width < number of runs
  merge a number of shortest runs
  [check/adjust memory]
merge runs to produce output
```

2.2 Adjustment Mechanisms

In-buffer sort phase

During this phase, the sort process collects data into buffers and sorts each buffer using some in-memory sort algorithm. When it runs out of free buffers, it tries to allocate more memory. If the system can provide more space, the in-buffer sort phase continues. In this way, the work space increases gradually, one buffer at a time. When the sort reaches the end of input or cannot acquire more buffer space, it proceeds to the in-memory merge phase.

If acute shortage of memory space occurs, a sort in this phase could "roll back" its input and release the last buffers acquired. This is a rather drastic step though so we have not considered it further.

In-memory merge phase

During an in-memory merge, the sorted data is written to a temporary file as a run. As buffers become empty, they can either be released (if the system is short of

memory) or used for loading data for the next run. Whether a buffer is to be released or kept is a policy decision. It is not necessary to increase memory space during this phase.

External merge phase

The exact number of runs and amount of data are known when a sort enters this phase. If the number of runs is small, we attempt to allocate enough memory to complete the sort with a single merge step.

When the number of runs is large (relative to available memory), multiple merge steps may be needed. In this case, memory usage can be changed between merge steps by increasing or decreasing the merge fan-in. Once the fan-in for a step has been determined, the shortest runs are selected for merging.

Memory usage can also be adjusted by changing the size of input buffers. Larger buffers reduce disk overhead (total seek time and latency) because fewer I/O requests are needed to transfer the same amount of data. However, this option is not considered in our implementation; we always use a fixed buffer size. Normally, we use 32Kb buffers because we experimentally found that increasing the buffer size further yields only marginal benefits.

It is possible to reduce memory usage in the middle of a merge step, simply by terminating the input from one or more runs. The part of a run that was not processed can be treated as any other run during the next merge step. This seems like a rather radical option so we have not considered it further.

Wait queues

As part of the memory adjustment mechanism, we use multiple wait queues, each with an associated priority. A sort may enter a wait queue because of lack of memory in the system or to yield to higher priority sorts. When memory becomes available, the sorts in the queue with the highest priority are woken up first. A sort may move from one queue to another during processing. When a sort should wait and on what queue are decided by the memory adjustment policy.

2.3 Adjustment Policy

A memory adjustment policy is a set of rules for deciding when and by how much to increase or decrease memory usage of a sort, when a sort should wait and at what priority, and when waiting sorts should be woken up. The policy is independent from the actual memory adjustment mechanisms. By separating policies and mechanisms, we can easily study the effects of different policies.

A memory adjustment policy needs some system wide state information, including the number of active sorts, the amount of free memory in the system, the stage of each sort, etc. It also relies on a set of predefined parameters such as memory adjustment bounds. The objective is to improve system performance (throughput and response time) while at the same time ensuring fair treatment of competing sorts.

System sort space

In principle, a memory-adaptive sort should adjust its memory usage according to the total available memory space to the system. However, some database systems specify a maximum size for total sort space or use a separate buffer pool for sorts. If so, the total memory for sort jobs is limited. The limit can be a hard limit with a fixed value or a soft limit which changes according to the system workload. In this section and the following one, available memory space refers to the available memory reserved for sort jobs.

In our adaptive sort two configuration parameters determine total sort space and memory allocation: *SysSortSpace* and *MemUnit*. *SysSortSpace* is the limit on total memory space available for sorts. *MemUnit* is the size of one data buffer plus related sort structures. A sort allocates memory one *MemUnit* at a time. The value of *SysSortSpace* is set according to total memory size, while *MemUnit* can be used to tune sort performance.

Sort stages

For the purpose of memory adjustment, we consider a sort to be in one of seven different stages.

Stage 0: The sort is waiting to start. Since a small sort requires little memory and releases the memory quickly, it may be beneficial to give a sort in this stage a small amount of memory and let it start. If it requires more space and the system is short of memory, the sort can be put into a wait queue later on.

Stage 1: The sort is processing the first run during the in-memory sort phase. It is not known yet if the input will fit completely in memory. Giving a sort in this stage additional memory may be very beneficial if it results in the input being sorted completely in memory.

Stage 2: All input data has been loaded into memory and the sort is in the in-memory merge phase, i.e., the sort has enough space for an in-memory sort. A sort in this stage is unable to reduce its memory usage. On the other hand, extra memory will not improve the performance of the sort.

Stage 3: The sort is processing the remaining runs during in-memory sort phases. At this stage it is known that an external merge is necessary. Additional

memory may reduce the number of runs, which may reduce the number of external merge steps and speed up merging. If a single merge step is sufficient, the sort time is not very sensitive to memory usage. Therefore, memory space is less critical to a sort in this stage than it is to a sort in stage 1 or stage 5.

Stage 4: The sort is processing the remaining runs during in-memory merge phases. Similar to Stage 3, it is known that external merging is necessary.

Stage 5: The number of runs could not be merged in a single step and the sort is performing intermediate merges during this stage. It checks the available memory before each merge step and adjusts the fan-in accordingly. When there is enough memory to merge all remaining runs in one step, the sort allocates enough space, and goes to the last merge step right away. Since extra memory will help reduce the amount of I/O, additional memory is very important to a sort in this stage.

Stage 6: The sort merges all remaining runs producing the final output. Since the amount of data is known at the start of the merge step, the sort is able to allocate exactly the amount of memory needed. One page less of memory will result in another merge step.

Based on the above analysis, we decided on the following priorities (from highest to lowest): 0, 1, 5, 3, 4. Sorts in stage 2 or stage 6 are not included in the list because they do not change their memory usage.

Memory adjustment bounds

We do not allow a sort to increase or decrease its work space arbitrarily but restrict the size to be within a specified range. The range depends on what stage the sort is in and on the number of active sorts. The main purpose of this restriction is to prevent a sort from monopolizing resources, thereby starving other sorts. The lower bounds prevent sorts from attempting to run with too few resources. Figure 1 illustrates these memory bounds.

- 1stMin: minimum memory for a sort to start. One *MemUnit* is usually enough.

- 1stRunMin: minimum memory for the first run. This bound guarantees that a sort of size less than 1stRunMin will always be sorted in memory.

- 1stMax: maximum memory for the first run. When a sort reaches this point, it gives up its effort to sort the data in memory and falls back on external sorting. A substantial amount of memory is then released to improve the performance of other sorts in the system.

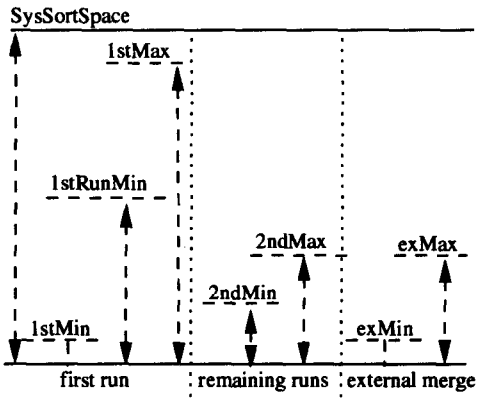

Figure 1: Sort Memory Usage Bounds

- 2ndMin: minimum memory for processing the remaining runs. This should be large enough so that most medium size sorts will require only one merge step.

- 2ndMax: maximum memory for processing the remaining runs. This bound prevents a very large sort from taking too much sort space when there are competing sorts in the system.

- exMin: minimum memory for an external merge. This must be high enough for a fan-in of a least two.

- exMax: maximum memory for an external merge. This prevents a sort consisting of many runs from taking too much sort space for merge buffers. When reaching this limit, a sort converts to using multiple merge steps.

The lower bounds are usually fixed based on system configuration, while the upper bounds depend on total amount of free memory and workload in the system. Table 1 list the values used for our experiments. *evenShareMem* is the total sort space size divided by the number of active sorts in the system. It changes dynamically as the workload changes.

Waiting

When a sort fails to allocate more memory, it can either wait or proceed with its current work space. Proceeding immediately without waiting may cause a small sort to rely on external merging or a sort with relatively few runs to resort to multiple merge steps. On the other hand, waiting increases the sort's response time.

In our system, a sort is allowed to wait only if it has not reach the upper bound on memory for its current stage (1stRunMin, 2ndMax, or exMax). Otherwise, it

Table 1: Default Values for Memory Usage Bounds

memory bound	*default value*
1stMin	$MemUnit$
1stRunMin	$1/8 * SysSortSpace$
1stMax	$freeMem - MemUnit$
2ndMin	$5 * MemUnit$
2ndMax	$evenShareMem$
exMin	$MemUnit$
exMax	$evenShareMem$

will proceed with the memory it has acquired. A sort may wait in one of five situations:

W1: in stage 0 waiting to start;
W2: in stage 1 with 1stMin space;
W3: in stage 1 with more memory;
W4: in stage 3;
W5: before an external merge step.

When memory is released and there are multiple sorts waiting, we must decide which sort to wake up. For reasons explained below we settled on the following priority order for waiting sorts: W1, W3, W5, W4, W2.

In general, sorts with more memory space should have higher priority so that they can finish sooner and release a large amount of memory. However, we assign W1 sorts the highest priority to give very small sorts (requiring less than 1stMin memory) a chance to finish quickly. If a sort requires more memory and there is no free space, it becomes a W2 sort which are assigned a low priority because they hold little memory. Among sorts in stage 1, we make W2 sorts yield to W3 sorts to give them a chance to proceed (and finish) sooner. Sorts in stage 3 are allowed to acquire more memory and become W4 sorts when there is no free space in the system. If the remaining runs can be merged in one step with exMax memory and the sort cannot acquire enough memory to do so, the sort becomes a W5 sort. We give W5 sorts priority over W4 sorts to give them a chance to acquire enough memory to finish quickly and release all memory held.

Fairness

Our memory adjustment policy aims to improve overall system performance, that is, throughput and average response time, but it also takes into account fairness considerations. However, fairness is not achieved by simply assigning the same amount of memory to each sort job. Specifically, the following fairness considerations are reflected in our policy:

- a sort should not allocate more memory than needed. It is unfair for one sort to allocate extra memory it cannot use while others are waiting;
- a sort whose performance is not very sensitive to memory should yield to sorts whose performance is more affected by memory space;
- large sorts should not block small sorts indefinitely, while small sorts should not prevent large sorts from getting a reasonable amount of memory;
- when all other conditions are the same, older sorts should have priority over younger sorts.

3 Experimental Results

3.1 Sort Testbed

To evaluate our ideas for dynamic memory adjustment, we implemented our memory-adaptive sort and a testbed system. The sort testbed emulates (part of) a database environment, as shown in Figure 2. It includes a memory manager, a disk space manager, asynchronous I/O support, a sort job generator, and the sort system. When provided with system configuration parameters and sort test parameters, the testbed generates and executes a sequence of sort jobs and collects performance results.

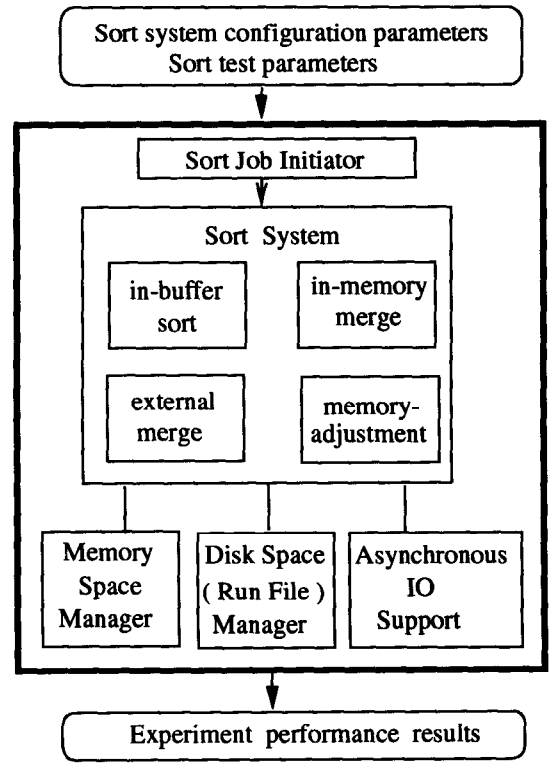

Figure 2: Testbed System

The *memory manager* is similar to the buffer manager in a database system but, in our case, it manages only the system sort space.

The *disk space manager* manages allocation and deallocation of run blocks on disk. It does not rely on the file system for space management – all run files are stored in a raw disk partition.

Asynchronous I/O is implemented by using separate I/O threads. Sort threads and I/O threads communicate through queues. All buffers are in shared memory and raw I/O is used for reading and writing.

The *sort job generator* constructs sort jobs according to the given test parameters and drives the sort system by submitting sort request.

The *sort system* implements our memory-adaptive sort, including in-buffer sort, in-memory merge, external merge, and memory adjustment. The system is multi-threaded with each sort job running as a separate thread.

Input for a sort can either be read from disk or generated on the fly. The sort output is packed into buffers which can then be either written to disk or simply discarded. All experiments reported in this paper were run with input data generated on the fly and discarding output data. This allowed us to drive the sort system at maximal speed. It simulates the case when the sort is an intermediate operator between a (fast) producer and a (fast) consumer operator.

Static sorts are run using exactly the same system, the only difference being that memory adjustment is disabled. In this mode, each sort allocates a fixed amount of memory and releases the whole space when the sort is finished. By using exactly the same sort algorithms, we isolate the effects of dynamic memory adjustment.

3.2 System Configuration

The machine used for the experiments was a Dec Alpha 3000/500S with a clock rate of 150 MHz and a 512 Kb off-chip cache. All runs were stored on a single disk, a Seagate ST-15150W with the following characteristics: average access time (read/write) 8.0/9.0 ms, single track seek (read/write) 0.6/0.9 ms, maximum seek time (read/write) 17/19 ms, average latency 4.17 ms, and transfer rate 47.4 to 71.9 mbits/sec.

Table 2 lists the configuration parameters and their default values used in the experiments.

System sort space is the total memory space available for sorts. The *one sort space limit* is used by memory-static sort as the default memory size.

Sort buffer size is the size of a data buffer for in-memory sort/merge. The unit of memory adjustment is a data buffer plus the space for additional data structure for sorting. Instead of sorting the records in the data buffer directly, we sort a set of pointers pointing to the records.

The *run block size* is the buffer size for external merge and the I/O transfer unit. To be able to drive disks at full speed, we use two I/O threads and two extra buffers per disk.

Table 2: Sort System Parameters

Parameter	*Default Value*
system sort space	32 Mb
one sort space limit	4 Mb
sort buffer size	64 Kb
run block size	32 Kb
number of disks	1
I/O agents per disk	2
read-ahead buffers	2
maximum concurrency	10

Maximum concurrency limits the number of active sorts. When the number of active sorts reaches this limit, incoming sorts are forced to wait until the number of active sorts drops below the limit.

The data plotted in the graphs in this section represent averages computed from five experiments.

3.3 Single Sort Performance

When there is only one active sort in the system (the single sort case), our adaptive sort is able to employ all memory resources available while a static sort is limited by the single sort space limit. Figure 3 shows the observed elapsed time of a single sort as a function of input size. Figure 4 shows the corresponding throughput measured as the amount of sorted data produced per second. Static sort changes from in-memory sort to external sort at an input size of 3585 Kb, while the adaptive sort changes at an input size of 29 Mb.

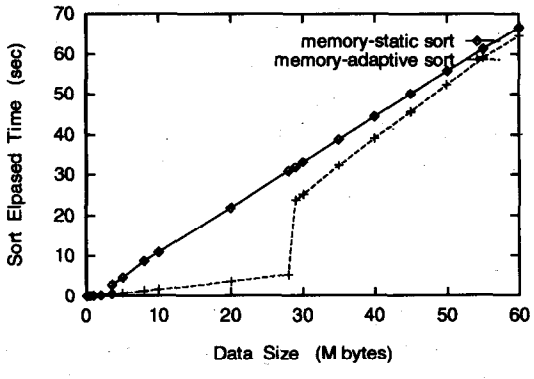

Figure 3: Elapsed Time as a Function of Input Size

For input less than 3585 Kb, both adaptive sort and static sort finish the sort entirely in memory and have the same elapsed time and throughput. For medium size input (3585 Kb - 29 Mb), static sort relies on external merging, while adaptive sort sorts the data completely in memory. The difference in throughput

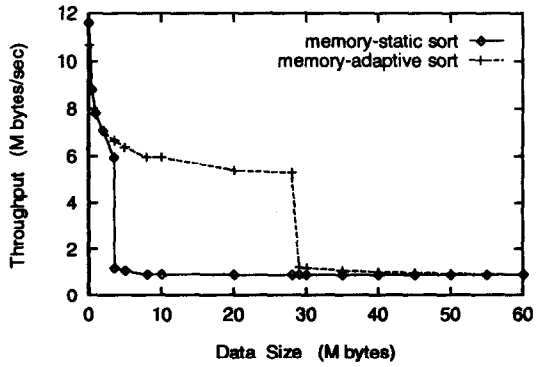

Figure 4: Throughput as a Function of Input Size

is dramatic, dropping from about 6 Mb/s to slightly over 1 Mb/s. One of the main objectives of memory-adaptive sort is to exploit this difference by trying to complete as many sorts as possible in memory.

Although the system sort space size was fixed in these experiment, adaptive sort also utilizes memory efficiently when the system sort space changes dynamically. Static sort allocates the same amount of memory for all sorts. If the system sort space is too small to meet the requirement, the sort has to wait. However, adaptive sort can proceed with a small amount of memory. If the input size happens to be small, the job finishes quickly without waiting for a large chunk of memory it in fact does not need.

In summary, adaptive sort saves memory space on small sorts, drastically reduces the elapsed time of medium size sorts, and performs better than or as well as static sort for large inputs.

3.4 Concurrent Sorts

A database system does not have the luxury of running only one operation at a time. Many operations may be running concurrently, competing for memory and I/O resources. This section reports on experiments investigating the effects of memory adjustment on (sort) system throughput and response time when multiple sorts are running concurrently.

Workload

The workload for each experiment consisted of a sequence of sort jobs of varying size. The input size of a sort job was randomly drawn from a specified sort size distribution. Several different distributions were used (see further below). Input records were 64 bytes long with a randomly generated 10 byte key.

Within each test run, a fixed number of sort jobs were always running concurrently. This *degree of concurrency* is an input parameter for a test run. If the degree of concurrency is n, n sort jobs would be started initially and as soon as one finished another one would be started.

To get some basis for deciding on a distribution of sort sizes, we analyzed the sorts generated when running the TPC-D benchmark queries [Raa95]. More specifically, we analyzed the execution plans used by DB2/6000 version 2 for each of the 17 queries on a TPC-D database with 26 indexes. We found a total of 55 sorts with the size distribution shown in Table 3.

Table 3: TPC-D Sort Sizes, scale factor 1.0

Input size range	Average size	No of sorts	Frequency
0 – 100K	17K	15	27%
100K – 1M	380K	19	35%
1M – 4M	2M	11	20%
4M – 10M	7M	4	7%
10M – 30M	16M	6	11%

Our analysis revealed that small sorts occurred frequently while large sorts were relatively rare. Small sorts were often used in nested loop joins to sort row identifiers before accessing the inner table. Many of the TPC-D queries also require a sort of the final result, which usually is small. Large sorts were typically caused by sort-merge joins or group-by.

The number and size distribution of sorts depend on the database system and the execution plans generated so no general conclusions can be drawn from this analysis. Nevertheless, it provides some data where there was none before.

Table 4 shows the four sort job sets used for the experiments in this section. D1 is from execution plans of a set of queries on a small database in our system. D3 is based on the result of our analysis of the queries in the TPC-D benchmark. D2 is a case between D1 and D3 while D4 contains larger sorts than D1-D3.

Unrestricted concurrency

When the number of concurrent sorts increases, each sort gets less memory and there is more competition for I/O bandwidth. More sorts will require external merging which reduces throughput measured in bytes of sorted data produced per second. The question is how rapidly performance deteriorates.

Figure 5 to Figure 8 show the sorted data throughput as a function of the number of sorts running concurrently.

All sorts in D1 are small enough to always be sorted in memory, even with 12 sorts running concurrently. In this case the system is completely CPU bound. Figure 5 shows that the two sort methods achieve about

Table 4: Sort Job Characteristics

Sort Data Set D1: 100 sorts					
Sort Size	50K	600K	1M	2.5M	
Frequency	62%	27%	7%	4%	
Sort Data Set D2: 100 sorts					
Sort Size	60K	1M	3M	5M	10M
Frequency	10%	20%	60%	5%	5%
Sort Data Set D3: 100 sorts					
Sort Size	17K	380K	2M	7M	16M
Frequency	27%	35%	20%	7%	11%
Sort Data Set D4: 100 sorts					
Sort Size	60K	3M	5M	50M	100M
Frequency	10%	55%	30%	3%	2%

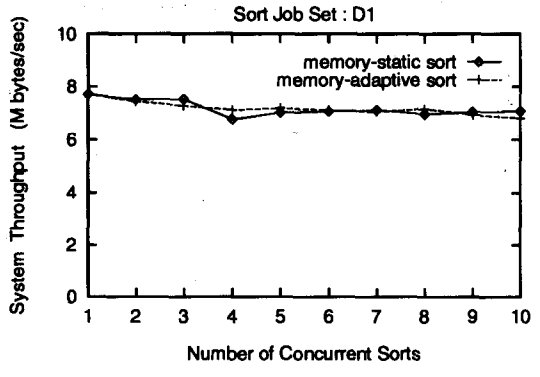

Figure 5: Throughput of D1

the same throughput, which confirms that the overhead of dynamic memory adjustment is minimal. As the number of concurrent sorts increases, throughput decreases only slightly. This is a result of more frequent thread switching which (probably) also results in poorer cache performance.

For the other three workloads, memory-adaptive sort has significantly higher throughput when the number of concurrent sorts is low (see Figures 6 to 8). In the best case, the throughput is up to 6 times higher. The difference decreases as the number of concurrent sorts increases because of the increased competition for memory and I/O bandwidth. This shows that memory-adaptive sort works in the sense that, when possible, it exploits available memory to speed up sort jobs and gracefully degrades when the competition for memory space increases.

Only workload D4, see Figure 8, shows increased throughput as the number of concurrent sorts increases (up to 4). The few large sorts in this workload are completely I/O bound, leaving free CPU cycles that will only be used (by small sorts) when there are enough sorts active at the same time.

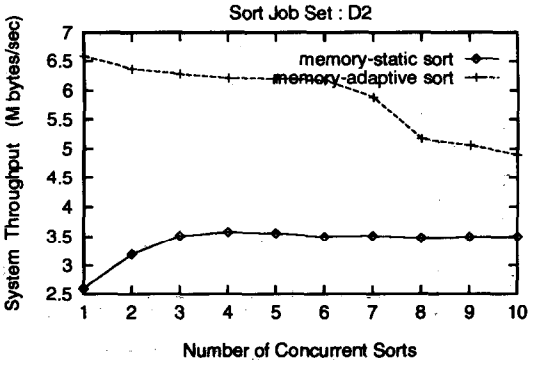

Figure 6: Throughput of D2

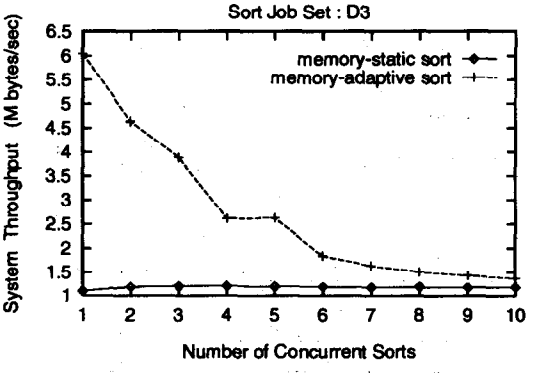

Figure 7: Throughput of D3

An important objective of memory-adaptive sort is to reduce the number of external sorts. Table 5 shows that, when memory space is available, all but the largest sorts are completed entirely in memory. When many sorts run concurrently, less memory is available for each sort so fewer sorts can be completed in memory. This effect accounts for most of the decrease in throughput.

Limiting concurrency

A database system has no control over the load but it can decide how to make use of its resources to improve throughput and/or response time. As we saw in the previous section, running too many sorts concurrently reduces throughput significantly. But the system does not have to start executing a sort immediately if the resources are already strained; it can make the sort wait until enough resources have been freed up. So the question is: How many sorts should the system run concurrently? The experiments described in this section attempt to provide some insight into this issue.

In these experiments we had 10 clients repeatedly submitting sort jobs. As soon as a client's previous job finished, it submitted another sort job. In other words, there were always 10 outstanding sort jobs, some being

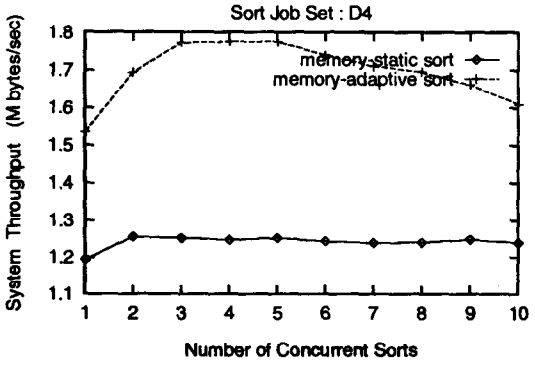

Figure 8: Throughput of D4

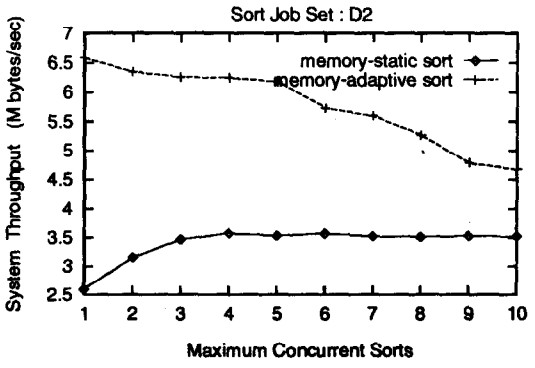

Figure 9: Throughput of D2

Table 5: Number of External Sorts (out of 100 Sorts)

Concurrent	D2		D3		D4	
sorts	ma	st	ma	st	ma	st
1	0	10	0	18	5	35
2	0	10	3	18	5	35
3	0	10	5	18	5	35
4	0	10	7	18	5	35
5	0	10	7	18	5	35
6	0	10	11	18	6	35
7	1	10	12	18	8	35
8	4	10	13	18	8	35
9	4	10	14	18	10	35
10	5	10	15	18	12	35

(ma: adaptive sort; st: static sort)

processed and some waiting to start. We then varied the number of sorts being processed concurrently and measured throughput and response time. Response time is the average time from when a client submitted a request until the last record in the output arrived.

Figures 9 to 12 show the throughput and average response time for D2 and D4 as the limit on concurrent sorts varies. (Limiting the number on concurrent sorts has no effect on D1 because the sorts are so small. The results for D2 and D3 are very similar because all sorts in these job sets are less then 32 Mb and, hence, can be sorted entirely in memory if run in isolation.) In all cases, except for D1, memory-adaptive sort achieves both better throughput and response time than static sort.

The graphs are best read from right to left. We first consider data set D2, see Figures 9 and 10. As the number of sorts being processed concurrently is decreased, both throughput and average response time improve for memory-adaptive sorts as more and more of the sorts are done in memory. The reverse is true for static sort.

D4 contains a few large sorts that cannot be completed in memory. In the time it takes to complete a 100 Mb sort, about 24 (6x4) sorts of size 25 Mb can be completed (assuming they can be done in memory). So in this case, processing only one sort at a time is clearly not a good idea. This effect is also visible in the graphs. Figure 11 shows that throughput initially increases as the limit on concurrent sorts decreases but then starts dropping (because CPU and memory resources are not fully utilized). Response time, see Figure 12, increases steadily as fewer sorts are processed concurrently.

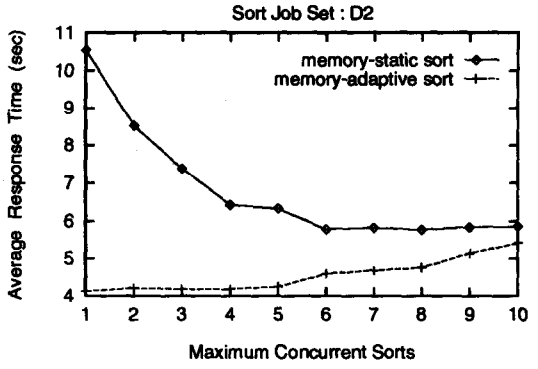

Figure 10: Average Response Time of D2

These experiments reinforce what we found in the previous section: completing as many sorts as possible in memory is crucial to overall system performance. But we also found that it is important to fully utilize available resources (memory, CPU, I/O).

4 Summary

This paper introduced a method and policy for dynamically adjusting the memory usage of external merge-sort at run time. We experimentally showed that this enables sorts to adapt their memory usage gracefully to the actual input size and fluctuations in available memory space. This was found to improve sort throughput significantly compared with static memory

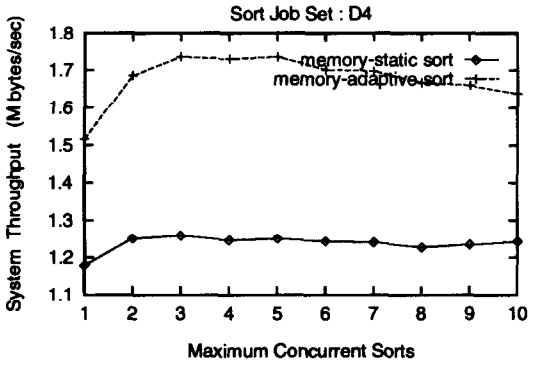

Figure 11: Throughput of D4

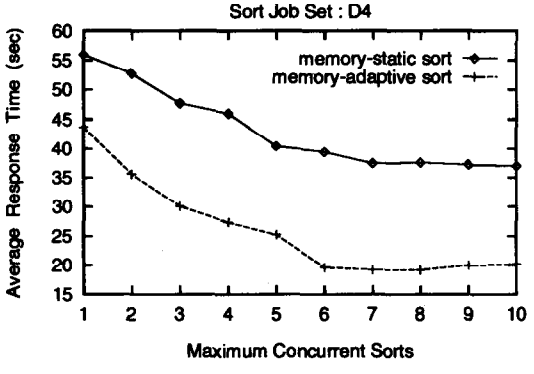

Figure 12: Average Response Time of D4

allocation.

To improve sort throughput, it is crucial to complete as many sorts as possible in memory. Allowing many sorts to run concurrently reduces the memory available to each sort, thereby reducing the fraction of in-memory sorts. In our experiments we found that limiting the number of sorts running concurrently improved both throughput and response time. This conclusion does not hold in general though; it depends on the sort size distribution.

We proposed a policy for balancing memory allocation among sorts running concurrently and competing for memory space. The policy worked fine in the sense that system throughput and response time were much improved over static allocation.

This paper applied dynamic memory adjustment to sorting. The same techniques can be applied to other memory intensive operations, join being the obvious candidate. Sort-merge join uses little memory for the actual join (except when there are many rows with the same value for the join columns). More memory is required for sorting the two input tables and the performance of sort-merge join depends largely on sort performance.

Dynamic memory adjustment is more important to hash join algorithms. Memory adjustment for hash joins has been studied by [ZG90], [PCL93b], and [DG94]. However, their work focused on how a single join can use extra space or release part of its space to affect I/O transfer unit size. They did not take into account the memory requirements in different stages of a join and did not consider balancing memory requirements among concurrent joins (and sorts).

References

[DG94] Diane L. Davison and Goetz Graefe. Memory-contention responsive hash joins. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 379–390, Sept. 1994.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun. 1993.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, J. Gray, and Dave Lomet. Alphasort: A risc machine sort. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 233–242, 1994.

[PCL93a] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 618–629, Aug. 1993.

[PCL93b] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptible hash joins. In *Proc. of ACM SIGMOD Conf.*, pages 59–69, May 1993.

[Raa95] F. Raab. *TPC Benchmark(tm) D (Decision Support), Working Draft 9.1.* Transaction Processing Performance Council, San Jose CA, 95112-6311, USA, February 1995.

[ZG90] Hansjorg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 186–197, Aug 1990.