Memory-Adaptive External Sorting

HweeHaw Pang
Michael J. Carey
Miron Livny

Technical Report #1154

June 1993

Memory-Adaptive External Sorting

HweeHwa Pang
Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin - Madison

Abstract

In real-time and goal-oriented database systems, the amount of memory assigned to queries that sort or join large relations may fluctuate due to contention from other higher-priority transactions. This study focuses on techniques that enable external sorts both to reduce their buffer usage when they lose memory, and to effectively utilize any additional buffers that are given to them. We also show how these techniques can be extended to work with sort-merge joins. A series of experiments confirms that our proposed techniques are useful for sorting and joining large relations in the face of memory fluctuations.

This work was partially supported by a scholarship from the Institute of Systems Science, National University of Singapore, and by an IBM Research Initiation Grant.

An abridged version of this paper appears in the proceedings of the 19th International Conference on Very Large Data Bases, August 1993.

1. Introduction

Database management systems (DBMS) are faced with increasingly demanding performance objectives. These objectives include time constraints, as in real-time database systems [SIGM88, RTS92], and administratively-defined performance goals, as in goal-oriented database systems [Ferg93, Brow93]. Traditional DBMS scheduling policies are no longer adequate to meet such objectives; a DBMS has to prioritize transactions that are competing for system resources according to the system-wide objectives and the resource requirements of the transactions. A consequence of priority scheduling is that transactions may be forced to release some or all of the resources that they hold. Moreover, executing transactions may also be given additional resources as they become available. Active transactions may therefore experience changes in resource availability during their lifetimes, depending on the priority of competing transactions.

A common practice in existing database systems is to allocate a fixed amount of memory to each query (or subquery) throughout its lifetime. Unfortunately, this practice does not work well with prioritized transaction scheduling because certain queries, particularly those that join or sort large relations, can hold on to a large number of buffers for extended periods of time. If these queries are permitted to hold on to their buffers until they complete, other higher-priority transactions may not be able to execute due to a shortage of memory. This seriously reduces the effectiveness of priority scheduling. Moreover, this practice does not allow a query to take advantage of excess memory that may become available. There is therefore a need for large queries to be adaptive when memory availability varies. In a recent paper [Pang93], we presented and evaluated techniques that allow hash joins to adapt to changes in their allocated memory. This study focuses on the same problem for large external sorts, i.e. sorts that involve relations that cannot fit entirely in the available memory, and for sort-merge joins.

Sorting is frequently used in database systems to produce ordered query results. It is also the basis of sort-merge join [Blas77], a join algorithm employed by many existing DBMSs, and it is used in some systems for processing group-by queries. An external sort consists of two steps: the first step fetches portions of the relation into memory to be sorted and written out as sorted runs, while the second step (which may involve several sub-steps) merges these runs into the sorted result. For a large relation, both the sort step and the merge step can potentially utilize many memory pages. Moreover, sorting a large relation may take a long period of time. Consequently, during the lifetime of a large external sort, the DBMS may wish to appropriate some of the sort's memory to satisfy the memory requirements of higher-priority transactions that arrive; buffers that are taken away may subsequently be returned after those

transactions leave the system. Given the prospect of continually having memory taken away and given back during its lifetime, it is desirable for an external sort to have the ability to continue its execution after it loses some of its buffers (and hence be *partially preemptable*). An external sort should preferably also have the capability to subsequently adapt its buffer usage to take advantage of any extra memory that may become available. To simplify our discussion, we shall henceforth refer to these changes in memory allocation as memory fluctuations.

One way to deal with memory fluctuations would be for the DBMS to employ virtual memory techniques to page the buffers of an affected external sort into and out of a smaller region of allocated memory, without having to inform the sort operator. If the DBMS detects that this is causing too many page faults, it could suspend the sort altogether. An advantage of this approach is that it shields the external sort algorithm from the complexity involved in adapting to memory fluctuations. However, there may be severe performance drawbacks associated with this approach. On one hand, suspending external sorts that are affected by memory fluctuations reduces the number of active transactions, which may lead to under-utilization of system resources. Paging the buffers of an external sort, on the other hand, is likely to result in thrashing when the difference in the amount of available memory and the number of buffers used is significant.

In this study, we investigate a different approach, namely, to directly involve the affected external sorts in adapting to memory fluctuations. We propose and study the performance of a memory-change adaptation strategy called *dynamic splitting*. Dynamic splitting adjusts the buffer usage of external sorts to reduce the performance penalty that results from memory shortages and to take advantage of excess memory. In addition, we study how dynamic splitting works with several different in-memory sorting strategies and merging strategies that external sorts can employ. We also show how our techniques can be extended to handle sort-merge joins. To understand the performance trade-offs of different strategies, and to identify those strategies that adapt well to changes in system buffer usage, we have constructed a detailed simulation model of a database system. This model enables us to study the behavior of the different strategies over a wide range of system resource configurations.

The remainder of this paper is organized as follows. Section 2 reviews the standard external sort algorithm, together with some implementation techniques that are commonly employed by external sorts. The issue of adapting external sorts to memory fluctuations is addressed in Section 3, which also introduces the dynamic splitting strategy. A detailed simulator of a database system, intended for studying the performance of the various strategies, is described in Section 4. Section 5 presents the results of a series of simulation experiments showing that, over a wide range of

system conditions, dynamic splitting offers an effective solution to the problem of memory fluctuations. Then, in section 6, we discuss how the same mechanisms that make external sorts memory-adaptive can be extended to sort-merge joins. Finally, our conclusions are presented in Section 7.

2. Standard External Sort Algorithms

An external sort involves two distinct phases. The first phase is a *split* phase, which employs an in-memory sorting method to divide the source relation into a number of sorted runs. The second phase, the *merge* phase, consists of one or more merge steps, each of which combines a number of runs into a single sorted run. The merge phase terminates when only one run remains. Within this framework, the choice of the in-memory sorting method for the split phase is independent of the choice of the merging strategy. This section reviews the common sorting methods and merging strategies that are found in the literature.

2.1. In-Memory Sorting Methods

Quicksort and *replacement selection* are two in-memory sorting methods that are commonly used in external sorts. An external sort that employs Quicksort first fills the available memory with as many pages of the source relation as will fit at a time, sorts the tuples in the memory-resident pages, and then writes the result out as a sorted run. This process is repeated until the entire source relation has been scanned. With Quicksort, the length of the runs produced is the size of the memory that is allocated for the split phase.

The second sorting method, replacement selection, works as follows: Pages of the source relation are fetched, and the tuples in these pages are copied into an ordered heap data structure. As more pages are fetched, the heap gradually grows in size until it occupies all of the available memory. At this point, a page of tuples is repeatedly removed from the heap and written to the current run so as to make space for the next incoming page of tuples. The tuples that are removed are those that have the smallest key values (assuming the source relation is to be sorted in ascending order) in the heap, subject to the condition that these tuples must have greater key values than the latest tuple written out in the current run. When none of the tuples in the heap satisfy this condition, the current run ends and a new run is started. On the average, the length of the runs produced by replacement selection is twice the memory allocated for the split phase [Knut73], i.e. twice as long as the runs generated with Quicksort. Hence, replacement selection creates only half as many runs as Quicksort. This could significantly shorten the merge phase that follows. A nice discussion of the details involved in implementing replacement selection can be found in [Salz90].

Although using replacement selection instead of Quicksort can shorten the merge phase, replacement selection is not always the preferred choice because it can also lead to a longer split phase [Grae90, DeWi91]. With Quicksort, there is a cycle of reading several pages from the source relation, sorting them, and then writing them to disk. In contrast, replacement selection alternates between reading a page from the source relation and writing a page to the current run. When the source relation and the run reside on the same disk, this results in many more disk seeks than in the case of Quicksort [Grae90]. In order to reduce disk seeks in replacement selection, a third possible in-memory sorting method is to use replacement selection, but to do block writes, i.e. to write several pages (say N) out to the run each time, instead of only one page at a time as in the original replacement selection procedure. A large N will result in fewer disk seeks, but at the same time it will reduce the average length of the runs. In the extreme case where N is equal to the amount of available memory, this replacement selection variant will fill all of the available buffers with relation pages, then write the sorted pages out together. In this case, the benefit of replacement selection is lost, since the length of the runs becomes the number of available buffers. Thus, the value of N has to reflect a compromise between reducing disk head movements and increasing the average length of the sorted runs. Whether the original replacement selection, Quicksort, or replacement selection with block writes is preferable depends not only on the hardware characteristics of the system, but also on memory allocation and the size of the relation to be sorted.

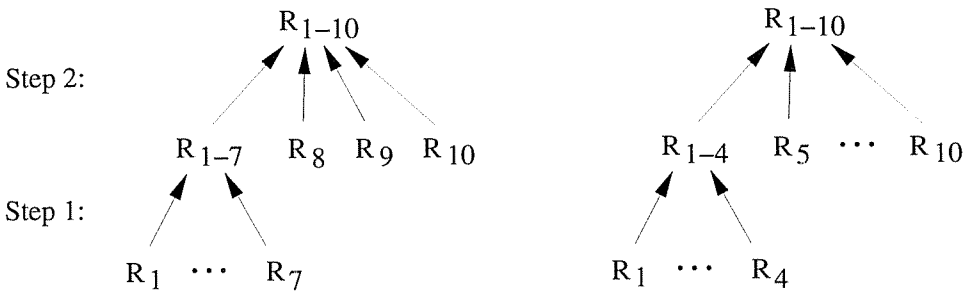
2.2. Merging Strategies

The split phase generates a set of n runs which have to be combined into a single sorted run in the merge phase. The merge phase consists of one or more steps; a merge step takes as input a number of sorted runs and combines them into a longer sorted run. Each input run of a merge step requires an input buffer, and an output buffer is needed for the output run. If at least $n + 1$ pages of memory are available for the merge phase, a single step suffices to combine all of the n runs.

When the source relation is large relative to the available memory, the database system may not be able to allocate enough buffers to a sort operator for it to merge all of its runs in a single step. In this case, preliminary merge steps are required to reduce the number of runs before the final merge can be carried out. Every preliminary merge step incurs extra I/O operations to fetch its input runs from disk and to write out its output run, and there is also extra CPU cost associated with each preliminary step. For this reason, it is desirable for every preliminary step to combine as many runs as the available memory allows, so that there will be as few merge steps as possible. A simple strategy, then, is for each step to merge m - 1 runs, where m is the number of available buffers. Figure 1(a) illustrates this

strategy for the case where n = 10 and m = 8. The 10 runs are denoted by R_1, .., R_{10}, and R_{1-10} denotes the run that results from merging R_1 to R_{10}. In this case, the 10 runs are merged in two steps. The first step merges all the tuples in R_1 to R_7 into R_{1-7}. Step two, which merges R_{1-7}, R_8, R_9 and R_{10} into the final result, begins only after the first step is completed. An alternate strategy is to merge just enough runs in the first step so that each of the subsequent steps merges m - 1 runs. Figure 1(b) illustrates the second strategy. The first merging strategy is called "naive" merging, and the second strategy is called "optimized" merging [Grae90]. From Figures 1(a) and 1(b), it should be apparent that "naive" merging is more expensive than "optimized" merging, as the final step has to process all of the tuples in the relation in both strategies. The preliminary steps incurs extra cost, and should therefore merge as few runs as possible (without increasing the number of merge steps) to keep the extra cost down. By merging more runs, "naive" merging increases the cost of the preliminary steps unnecessarily. Thus, the general rule is to adopt "optimized" merging [Grae93].

Another important aspect of the merging strategy concerns the choice of input runs. All of the merge steps, other than the final merge, have a choice of input runs and should thus merge the shortest possible runs. Such a choice minimizes the cost of the preliminary merges in two ways: Firstly, choosing the shortest runs for a given merge step obviously minimizes its cost. Secondly, the output run of an early merge step may be selected as one of the input runs of a subsequent preliminary merge step. By minimizing the size of the input runs of the early merge step, and hence the size of its output run, the cost of the later merge step is also reduced because it needs to merge fewer tuples. For these reasons, all of the algorithms studied in this paper adopt the policy of merging the shortest possible runs at any given step.



(a) "Naive" Merging (b) "Optimized" Merging

Figure 1: Merging Strategies

3. Memory-Adaptive External Sort Algorithms

In the previous section, we assumed that the amount of available memory remains the same throughout the lifetime of an external sort. As discussed in the introduction, however, it is desirable for a sort operator to be able to execute with a varying number of buffers. This section gives a detailed description of a set of alternative memory-adaptive external sort algorithms. Since the in-memory sorting methods for the split phase are independent of the merging strategies for the merge phase, we shall first treat the in-memory sorting methods separately before addressing the merging strategies. Finally, we end the section by introducing some notation that will be used to denote different external sort algorithms throughout the rest of the paper.

3.1. Split-Phase Adaptation

If an external sort is in the split phase when it is asked by the DBMS to release a page, the sort can immediately do so if it has unused buffers, i.e. buffers that are not currently occupied by tuples from the relation. If all of its buffers are in use, however, it will have to clear some or all of the memory-resident tuples by writing them to output runs before it can free any of its buffers. In the case of typical implementations of Quicksort, all of the tuples in memory have to be sorted and written out as a new run before a page can be released[1]. When there are many tuples in memory, this may result in considerable delays. In contrast, with replacement selection, the sort needs only to remove a page of tuples from the heap, write the page out, and then release the empty page to the DBMS. Next, we consider the case where an external sort is given additional buffers in the split phase. With Quicksort, if the external sort is in the process of filling its memory with relation pages, the sort can immediately fill the newly allocated buffers with more relation pages. If the external sort has already started sorting its tuples to create a run, however, the new page will remain unused until the run has been written out and the external sort resumes fetching relation pages. In the case of replacement selection, the new buffer can immediately be used to fill the next incoming page of tuples. Thus, replacement selection is much more responsive than Quicksort in adapting to memory fluctuations.

[1] To implement Quicksort efficiently, sorting is usually not carried out on the actual tuples. Instead, a list of (key, pointer) pairs is created and sorting is done on this list. After the sorting is complete, tuples are retrieved from their respective pages by following the pointers associated with the keys. Thus, in a given step of the split phase, it is not possible to simply release a buffer after the first page of tuples has been written out.

3.2. Merge-Phase Adaptation Strategies

In contrast to the split phase, the merge phase does not adapt to memory fluctuations as easily. One possible solution is to adopt hybrid approaches that allow a sort operator to adapt to memory fluctuations only in the split phase, leaving the DBMS to suspend an affected external sort or page its buffers when it is in the merge phase. Besides the drawbacks of suspension and paging that we discussed in the introduction, these hybrid approaches would also prevent an external sort from taking advantage of extra memory (beyond the initially allocated amount) that may become available while the sort is in the merge phase. In this study, we will therefore explore a third alternative, called *dynamic splitting*, that actively involves an affected sort in adapting to memory fluctuations that occur during the merge phase.

3.2.1. Suspension

The most straightforward approach to deal with memory shortages that occur during the merge phase of an external sort is for the DBMS to suspend the external sort altogether. The buffers of the external sort can be taken away once it has been suspended. The only information that is needed to resume the merging is the position of the next tuple in each input run. Since the sort operator already keeps track of this information for normal merging operations, no special mechanisms are necessary for suspension. Our implementation of suspension fetches all the input buffers together when the external sort resumes. This reduces disk seek costs, as opposed to fetching the buffers on demand.

3.2.2. Paging

Another obvious way to deal with memory fluctuations during the merge phase is to resort to MRU paging whenever the memory available to an external sort is insufficient to hold all the input buffers for its current merge step. Our implementation of paging works as follows: The external sort keeps a copy of the current tuple of each input run in its private work space, where the tuples are merged. After writing out the smallest tuple to the output run, the external sort determines which input run this tuple came from, and then attempts to copy the next tuple from this input run. If the buffer for this input run is no longer in memory, the most recently used buffer is selected for replacement, and a disk read is issued to bring the required buffer back in. As with suspension, paging enables an external sort to relinquish its buffers as and when they are needed for replacement or for release to the DBMS.

3.2.3. Dynamic Splitting

Dynamic splitting is a strategy that is designed to adapt the merge phase of external sorts to varying memory allocation. When a shortage causes the available memory to go below the memory requirement of an executing merge step, this strategy adapts by splitting the merge step into a number of sub-steps that each fits within the remaining memory. When additional buffers are given, the merge steps can be combined into larger steps, i.e. steps that have more input runs, to take advantage of the now-larger memory. The details of the dynamic splitting strategy are presented below.

Suppose that a sort operator is currently executing a merge step, which can be either the final merge of all existing runs or a preliminary merge step. If a memory shortage occurs, causing the available memory to become less than the buffer requirement of the current merge step, the sort operator can immediately stop the current step, split the step into a number of sub-steps, and then start executing the first sub-step. To illustrate this, suppose that the merge phase of an external sort started with 10 runs and 11 buffers, which allowed all runs to be merged at once as in Figure 2(a). While the sort is executing this merge step, the available memory is reduced to 8 buffers. The sort operator responds by splitting the merge into a preliminary step that merges R_1 to R_4 into R_{1-4} (assuming "optimized" merging), and a final step that merges R_{1-4} with R_5 to R_{10} into R_{1-10}. After the split, the sort immediately starts to work on the preliminary step. (Note that some of the tuples from R_1 to R_4 have already been merged into R_{1-10} prior to the split, so only the tuples that still remain in R_1 to R_4 will be merged into R_{1-4} by the preliminary step.) This is illustrated in Figure 2(b), where the preliminary step, the merge step with the solid arrows, is the one that is being executed. The final step, which has dotted arrows, is inactive. Suppose that no further changes in memory allocation take place, and that the external sort completes the preliminary step without interruption. There are now only 7 runs, and the sort is ready to resume the final merge step. At this stage, R_{1-10} contains some of the tuples from R_1 to R_{10} that were merged prior to the split, R_5 to R_{10} each contains some remaining tuples, and the remaining tuples of R_1 to R_4 are now in R_{1-4}. To get the entire sorted result, the sort needs to complete R_{1-10}. This is achieved by merging R_{1-4} with whatever is left in R_5 to R_{10}, *appending* the result to R_{1-10} (Figure 2(c)).

Having discussed how dynamic splitting breaks a merge step into sub-steps in response to a memory reduction, we now present the provision in the dynamic splitting strategy that allows an external sort to combine existing merge steps to take advantage of extra buffers as they become available. We shall introduce this provision by continuing our earlier example. Suppose that, while the sort is executing the preliminary step (the step with the solid arrows) in Figure 2(b), the available memory increases to 11 pages again. Instead of completing this step before performing the final

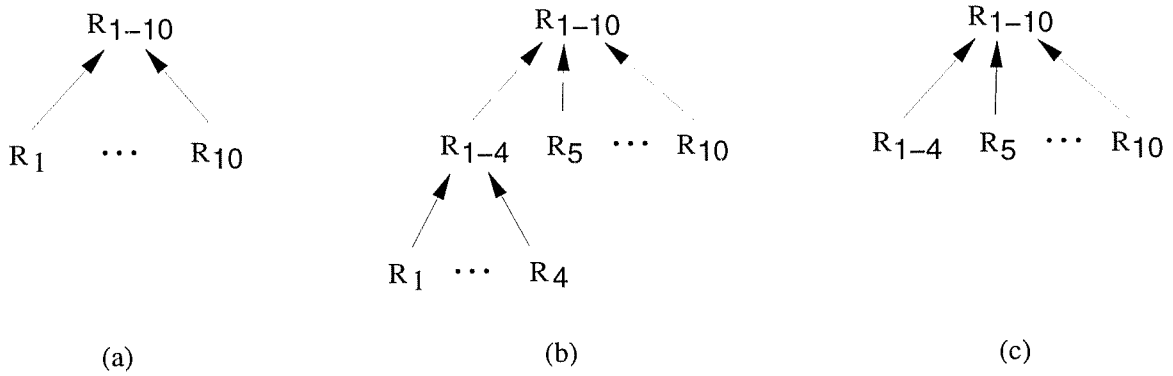


Figure 2: Splitting

merge as discussed previously, the sort operator can switch to the final merge directly. Figure 3 illustrates the process involved. At this stage, R_{1-10} contains some of the tuples from R_1 to R_{10} that were merged prior to the split. To produce the final result, the sort operator needs to append to R_{1-10} the rest of the tuples that were originally left in R_1 to R_{10}. However, since the sort has already been executing the preliminary step, some of the leftover tuples in R_1 to R_4 are now in R_{1-4}. It is therefore necessary for the external sort to first merge R_{1-4} with R_5 to R_{10}, appending the result to R_{1-10}. This is shown in Figure 3(a), where the final step, which has solid arrows, is now active and the preliminary step is inactive. Once R_{1-4} becomes empty, the sort operator can proceed to combine the final step with the preliminary step to produce a new final step that again merges the tuples remaining in R_1 to R_{10}, adding them to R_{1-10} as well (Figure 3(b)).

Although our only example shows a split that breaks a merge step into two sub-steps, the splitting procedure can be recursively applied to break a merge step into more than two sub-steps. For example, the preliminary step in Figure

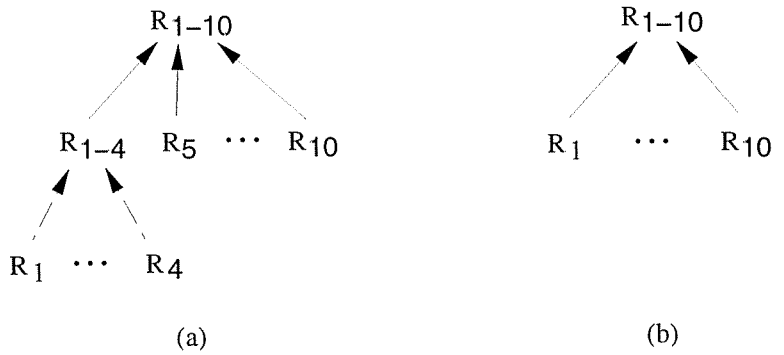


Figure 3: Combining Merge Steps

2(b) can be split again if memory decreases further while the step is being executed. Similarly, it is possible to combine more than two merge steps by applying the combining procedure recursively. To fully exploit the capabilities of dynamic splitting, the merge phase always starts with a step that combines all of the runs produced in the split phase. If the available memory is insufficient to execute this step, it is immediately split into sub-steps that fit in memory. This enables an external sort to take advantage of excess memory that may become available later by combining existing merge steps into steps that merge more runs, helping the sort to recover from a low initial memory allocation if memory happens to be in short supply at the beginning of the merge phase.

There is an important difference between dynamic splitting and the splitting process that was described in Section 2.2, which we will call static splitting to distinguish it from dynamic splitting. When an external sort has more runs to merge than its memory allows, static splitting is used to initiate preliminary merge steps to reduce the number of runs. Once started, a merge step has to execute to completion before another merge step can be executed. In contrast to static splitting, dynamic splitting allows an external sort to switch between merge steps, if it so desires, without having to wait for any step to complete. This ability to switch to a different merge step immediately is essential if an external sort is to effectively adapt its buffer usage to both increases and reductions in its allocated memory during the merge phase.

3.3. Notation for External Sort Algorithms

In this section, we have discussed three in-memory sorting methods and three merge-phase adaptation strategies. These in-memory sorting methods and merge-phase adaptation strategies will be evaluated in the performance study that follows. In addition, we will compare the relative merits of "naive" merging versus "optimized" merging for the following reason: While "optimized" merging always performs at least as well as "naive" merging *for a fixed memory allocation*, it is not obvious that this is still the case if the amount of memory allocated to a sort operator may be reduced while it is executing. In such situations, "naive" merging may turn out to be better because it utilizes all of the currently available buffers right away (while the sort operator still has them). Since the choice of in-memory sorting method, merging strategy and merge-phase adaptation strategy are all independent, there are 18 possible external sort algorithms, each employing a different combination of in-memory sorting method, merging strategy, and merge-phase adaptation strategy. To differentiate between the algorithms, we shall denote each algorithm by a string of the form $X_1X_2X_3$, where X_1 is either *repl*1, *quick*, or *repl*N (replacement selection, Quicksort, or replacement selection with N-page block reads and writes), X_2 is either *naive* or *opt* ("naive" merging or "optimized" merging), and X_3 is either *susp*, *page*, or *split* (suspension, paging, or dynamic splitting). Thus, for example, *quick,opt,susp* denotes external sort with

Quicksort, optimized merging, and suspension. This notation is summarized in Table 1.

4. Database System Simulation Model

To aid in our on-going research on real-time database systems, we have constructed a simulation model of a centralized database system. The portion of our simulation model that is relevant to this study is shown in Figure 4. There are five components: a *Source* that generates transactions one after another, and collects statistics on completed transactions; a *Transaction Manager* that models the execution details of transactions, including external sorts and sort-merge joins; a *Buffer Manager* that implements the buffer management policy; and a *CPU Manager* and a *Disk Manager* that are responsible for managing the system's CPU and disks, respectively. In this section, we describe how the simulation

Parameter	Meaning
In-Memory Sorting Method	
*repl*1	Replacement selection
quick	Quicksort
*repl*N	Replacement selection with N-page reads and writes
Merging Strategy	
naive	"Naive" merging
opt	"Optimized" merging
Merge-Phase Adaptation Strategy	
susp	Suspension
page	Paging
split	Dynamic Splitting

Table 1: Notation for External Sort Strategies

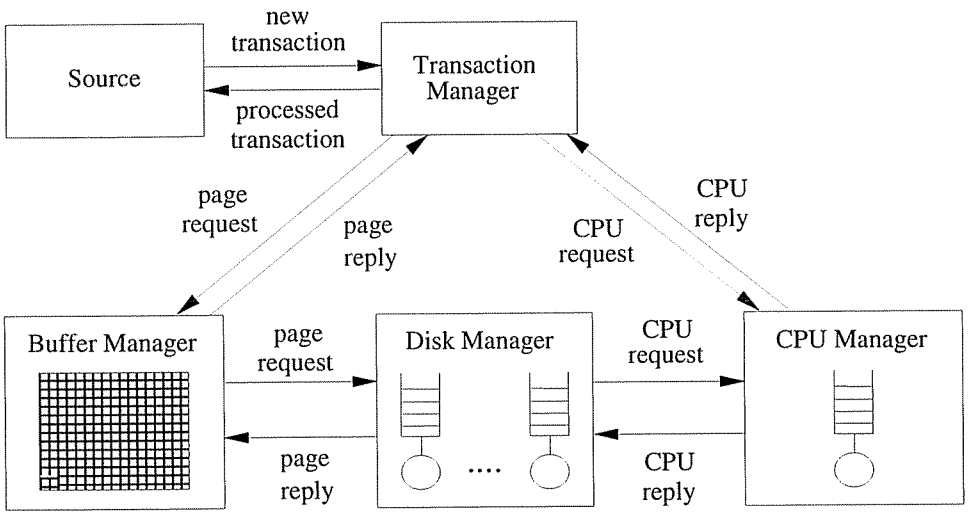


Figure 4: Database System Model

model captures the details of the database, workload, and various physical resources of a database system. The simulator is written in the DeNet simulation language [Livn90].

4.1. Database and Workload Model

Table 2 summarizes the database and workload model parameters that are relevant to this study. Our objective is to simulate a stream of external sorts or sort-merge joins on different source relations. To facilitate this, the database consists of *NumRel* relations. Each relation i ($1 \leq i \leq NumRel$), in turn, has a size of $RelSize_i$ MBytes and occupies contiguous pages on disk. If there are multiple disks, all relations are declustered (horizontally partitioned) [Ries78, Livn87] across all of the disks. To minimize disk head movement, the relations are allotted the middle cylinders of the disks; temporary files occupy either the inner cylinders or the outer cylinders.

In this study, the workload is made up of a series of queries which may be either external sorts or sort-merge joins, depending on the experiment. A new query is submitted to the database system only when the previous query has been completed. Each sort involves a relation **R**, which is uniformly selected from among the relations in the database, while a join involves two different relations **R** and **S**.

To investigate how different memory-adaptive mechanisms react to fluctuations in the amount of available memory, we simulate an environment where queries commonly have to contend for memory with other "transactions" that have small memory requirements and, occasionally, with "transactions" that have large buffer demands. The memory contention experienced by the active queries is modelled here by two other streams of competing memory requests, one small and the other large. The generation of small memory requests follows a Poisson distribution with a mean rate of λ_{small}, and the proportion of the total memory that a small request takes up varies uniformly between 0%

DB Model Parameters	Meaning	Default Value
NumRel	Number of relations in the database	10
$RelSize_i$	Size of relation i	20 MBytes
$TupleSize_i$	Average tuple size of relation i in bytes	256 Bytes
Workload Parameters	**Meaning**	**Default Value**
λ_{small}	Arrival rate of small memory requests	1 request/second
μ_{small}	Duration of small memory requests	0.8 second
MemReqThreshold	Max. % buffer demand of a "small" memory request	20%
λ_{large}	Arrival rate of large memory requests	0.1 request/second
μ_{large}	Duration of large memory requests	5 seconds

Table 2: Database and Workload Model Parameters

and *MemReqThreshold*. Moreover, the duration that a small request remains in the system after receiving its required memory is modelled using an exponential distribution with a mean of μ_{small}. Similarly, large memory requests arrive at a mean rate of λ_{large}. and have a mean duration of μ_{large}. Each large request occupies between 0% and 100% of the total memory.

4.2. Physical Resource Model

The parameters that specify the physical resources of our model, which consist of one CPU, multiple disks and main memory, are listed in Table 3. There is a single queue for the CPU, and a first-come-first-serve (FCFS) scheduling discipline is used. The MIPS rating of the CPU is given by *CPUSpeed*. Table 4 gives the cost of various CPU operations that are involved in the execution of external sorts and sort-merge joins. These CPU costs are based on instruction counts taken from the Gamma database machine [DeWi90].

Turning to the disk model parameters in Table 3, *NumDisks* specifies the number of disks attached to the system. Each disk has its own queue and disk requests are serviced according to the elevator algorithm. The characteristics of the disks are also given in Table 3. Using the parameters in this table, the total time required to complete a disk access is computed as:

Disk Access Time = Seek Time + Rotational Delay + Transfer Time

Parameter	Meaning	Default Value
CPUSpeed	MIPS rating of CPU	20 MIPS
NumDisks	Number of disks	1
SeekFactor	Seek factor of disk	0.000617
RotationTime	Time for one disk rotation	16.7 msec
NumCylinders	Number of cylinders per disk	1500
CylinderSize	Number of pages per cylinder	90 pages
PageSize	Number of bytes per page	8 KBytes
M	Total number of buffer pages	0.3 MBytes
SleepTime	Time between memory write activations	1 second
FlushThreshold	Threshold for memory writes	0 second

Table 3: Physical Resource Model Parameters

Operation	# Instructions	Operation	# Instructions
Initiate a sort	40,000	*Copy a tuple to output buffer*	64
Terminate a sort	10,000	*Compare two keys*	50
Start an I/O operation	1000		

Table 4: Number of CPU Instructions Per Operation

As in [Bitt88], the time required to seek across n tracks is given by:

$$\text{Seek Time (n)} = \text{SeekFactor} \times \sqrt{n}$$

Finally, the system has a total memory size of **M** MBytes. A memory reservation mechanism is provided to allow operators, including sorts and joins, to reserve buffers. Buffers that are reserved are managed by the operators themselves. Page replacement for non-reserved pages is handled as follows: The DBMS first attempts to find the least recently used clean page for replacement, avoiding the dirty pages initially. If there is no clean page, then the least recently used dirty page is selected. Before a dirty page can be replaced, however, its contents need to be written to disk. This lengthens the time that is needed to satisfy the waiting buffer request, and should be avoided if possible. For this reason, an asynchronous memory write process is provided to flush dirty pages to disk periodically [Teng84]. The write process is activated every *SleepTime* seconds. Upon activation, the process flushes all of the dirty pages that are older than *FlushThreshold.* The reason for flushing only the "old" dirty pages is to prevent unnecessary writes of pages that are frequently updated.

5. Experiments and Results

In this section, the database system simulator described in Section 4 is used to evaluate the performance of the alternative memory-adaptive external sort algorithms. We begin with an experiment where the amount of memory that is allocated to each external sort remains unchanged throughout its lifetime. This experiment is intended to give us an initial understanding of the trade-offs between different in-memory sorting methods and merging strategies before we delve into the complexities introduced by memory fluctuations. We then present a baseline model that is used to study the performance impact of memory fluctuations, and further experiments are carried out by varying a few parameters each time. The performance metric of interest here is the average sort response time.

To ensure the statistical validity of our results, we verified that the 90% confidence intervals for response times (computed using the batch means approach [Sarg76]) were sufficiently tight. The size of these confidence intervals was within a few percent of the mean in almost all cases, which is more than sufficient for our purposes. Throughout the paper we discuss only statistically significant performance differences.

5.1. No Memory Fluctuation

As mentioned above, our first experiment is designed to study the trade-offs of different in-memory sorting methods and merging strategies in the context of fixed memory allocation. For this experiment, we let $||R||$ be 20 MBytes, and vary **M**, the total system memory. Every external sort will execute with all of the system memory throughout its lifetime. λ_{small} and λ_{large} are both set to 0 request/second, so that there is no memory fluctuation. The rest of the parameters are assigned their default values in Tables 2 and 3. Finally, for the in-memory sorting method *repl*N, we let N be 6 (meaning that tuples are removed from the heap and written out in blocks of 6 pages). This choice was made because, for our system configuration, N = 6 leads to a considerable reduction in the average per-page disk access time over N = 1, as indicated in Table 5[2], without incurring the penalty of a significant increase in the number of sorted runs that the split phase generates, as will be evident from our experimental results.

Figure 5 presents the response times for the various combinations of in-memory sorting methods and merging strategies. The average number of sorted runs produced by each in-memory sorting method, together with the corresponding average number of merge steps and split-phase duration, are given in Table 6. Since there is no memory fluctuation in this experiment, the merge-phase adaptation strategies do not come into play here. The figure shows that all of the response times drop sharply initially as **M** is increased. As **M** grows beyond 0.6 MBytes, however, all of the curves level off. This behavior can be attributed to the reduction in the number of merge steps that takes place as the average number of generated runs decreases. As is evident from Table 6, the number of required merge steps initially drops drastically. However, once **M** reaches 0.6 MBytes, all three in-memory sorting methods produce fewer runs than the number of available buffers; thus, there can be no further reduction in the number of merge steps (until **M** grows to 20 MBytes, at which point there will be a sudden drop in response time because it will then be possible to sort the entire relation all at once in memory). In this region, increasing **M** leads to fewer sorted runs at the end of the split phase, and hence lower disk seek costs when the runs are merged; this accounts for the slight reductions in response time at the

N	1	2	4	6	8	10	12
Time	62	36	26	23	22	21	21

Table 5: Average Per-Page Disk Access Time (msec)

[2] The average per-page disk access time shown in the table includes the time spent waiting for service, i.e., including waits for completions of previously issued asynchronous disk write requests.

right-hand side of Figure 5.

Comparing the response times of the three in-memory sorting methods, it is clear that *repl* 1 consistently yields the worst performance. This is due to the large number of random I/Os that *repl* 1 produces, as the external sort alternates between reading a relation page and writing a page to the output run. In contrast, Quicksort writes out an entire run each time, thus producing considerably fewer random I/Os. Quicksort therefore has a much shorter split phase than *repl* 1, which more than offsets the longer merge phase that results from the larger number of runs that Quicksort generates. (Similar observations about the relative trade-offs between Quicksort and *repl* 1 were made in [Grae90,

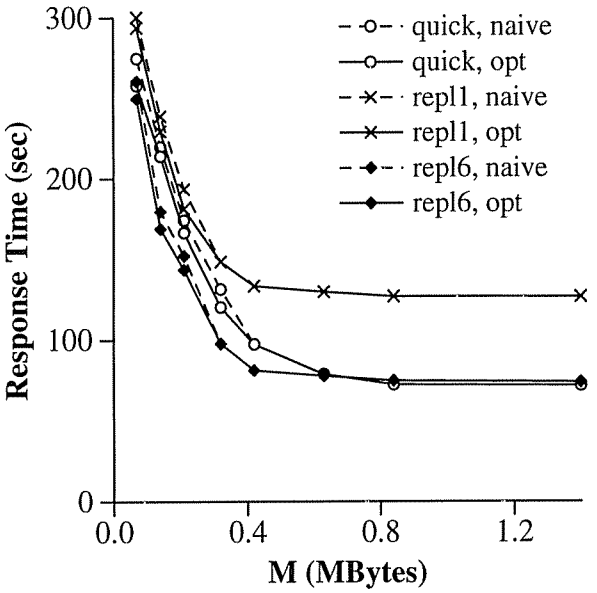


Figure 5: No Memory Fluctuation

M MBytes (pages)	0.07 (9)	0.14 (18)	0.20 (27)	0.31 (41)	0.41 (54)	0.61 (81)	0.82 (108)	1.36 (179)
# of Runs								
quick	280	149	101	65	52	34	25	15
repl 1	141	75	52	33	27	18	13	8
repl 6	202	89	57	35	28	19	14	9
# of Merge Steps								
quick	32.0	9.0	4.0	2.0	1.0	1.0	1.0	1.0
repl 1	15.7	4.2	1.9	1.0	1.0	1.0	1.0	1.0
repl 6	22.4	4.9	2.1	1.0	1.0	1.0	1.0	1.0
Split-Phase Duration (sec)								
quick	34	31	29	29	28	27	27	27
repl 1	89	86	85	84	83	83	82	82
repl 6	34	31	31	31	30	30	30	30

Table 6: Performance Results for No Memory Fluctuation

DeWi91].) By writing multiple pages instead of only a single page each time as in *repl* 1, *repl* 6 is able to significantly reduce the number of disk seeks in replacement selection, bringing the duration of its split phase much closer to that of *quick*. Moreover, the number of runs that *repl* 6 creates is only marginally more than *repl* 1 in almost all cases. Thus, *repl* 6 is clearly superior to *repl* 1 as a replacement selection procedure. Between *quick* and *repl* 6, *repl* 6 is the winner when $\mathbf{M} < 0.6$ MBytes, whereas *quick* is just slightly faster for $\mathbf{M} > 0.6$ MBytes. The trade-off between *quick* and *repl* 6 is again due to the number of runs that the two approaches generate, relative to the amount of allocated memory. Table 6 shows that for $\mathbf{M} < 0.6$ MBytes, *quick* results in more merge steps, and consequently a longer merge phase, than *repl* 6. This is why *repl* 6, which creates significantly fewer runs than *quick*, is superior there. For $\mathbf{M} \geq 0.6$ MBytes, there is enough memory to merge all of the runs produced by *quick* in a single step, so *repl* 6's fewer runs gives it little advantage over *quick*. In this region, the duration of the split phase becomes the dominant factor. Since Quicksort requires fewer CPU instructions than replacement selection, which incurs extra CPU cost in copying tuples between its heap structure and the input/output buffers, *quick* is marginally faster than *repl* 6 in this region.

Next, we turn our attention to the two merging strategies, optimized merging (*opt*) and naive merging (*naive*). Figure 5 shows that *opt* consistently leads to shorter response times than *naive* for $\mathbf{M} < 0.4$ MBytes, whereas the two merging strategies yield identical performance when $\mathbf{M} > 0.4$ MBytes. Recall that *naive* and *opt* differ in the number of runs that they combine in the first preliminary merge step. The output run of the first preliminary merge step may in turn be combined by a subsequent merge step, the output run of which may be the input of yet another merge step, and so on. The decision of *naive* to include more runs in the first preliminary step thus leads to an increase in the cost of each of these affected steps [Grae93]. The more merge steps there are, the larger the number of affected steps becomes, and consequently the higher the penalty of *naive* gets. For small **M** values, the number of sorted runs that the merge phase has to combine is large relative to the available memory, as shown in Table 6. This results in many merge steps, causing the observed differences in response time between *naive* and *opt* in Figure 5. Conversely, when **M** is large, the number of merge steps required is small, and so is the penalty of choosing *naive* over *opt*. As **M** increases, the number of merge steps reduces gradually until, when only a single merge step suffices to combine all of the runs, there is no difference between the two merging strategies.

Having now gained initial intuition regarding the performance characteristics and the relative merits of the in-memory sorting methods and merging strategies for fixed memory allocation, we can now proceed to evaluate their performance in the face of memory fluctuations. We will also explore how they interact with the merge-phase adaptation

strategies described in Section 3.2.

5.2. Baseline Experiment

In our baseline experiment, we simulate a situation where the relation to be sorted is much larger than the available memory. This is done by setting $||R||$ to 20 MBytes and **M** to 0.3 MBytes. Small memory requests arrive at an average rate of λ_{small} = 1 request/second and stay in the system for an average of μ_{small} = 0.8 second. *MemReqThreshold* is set to 20%. Large memory requests arrive at λ_{large} = 0.1 request/second, and each large request lasts an average of μ_{large} = 5 seconds. The parameter settings for this experiment are summarized in Tables 2 and 3.

Figure 6 gives the response time of the various external sort algorithms for this experiment. The figure shows a wide spread of response times, from a high of 320 seconds produced by *quick,opt,susp* down to a low of 141 seconds, using *repl6,opt,split*. This indicates that the choice of external sort algorithm can have a very significant performance impact. We observe that the four shortest response times are all produced by external sorts that employ *split*. Moreover, the five worse performers all employ *susp*. To understand the reason behind these behaviors, we shall analyze the merge-phase adaptation strategies before considering the in-memory sorting methods and the merging strategies further, as the merge-phase adaptation strategies appear to exert the greatest influence on performance.

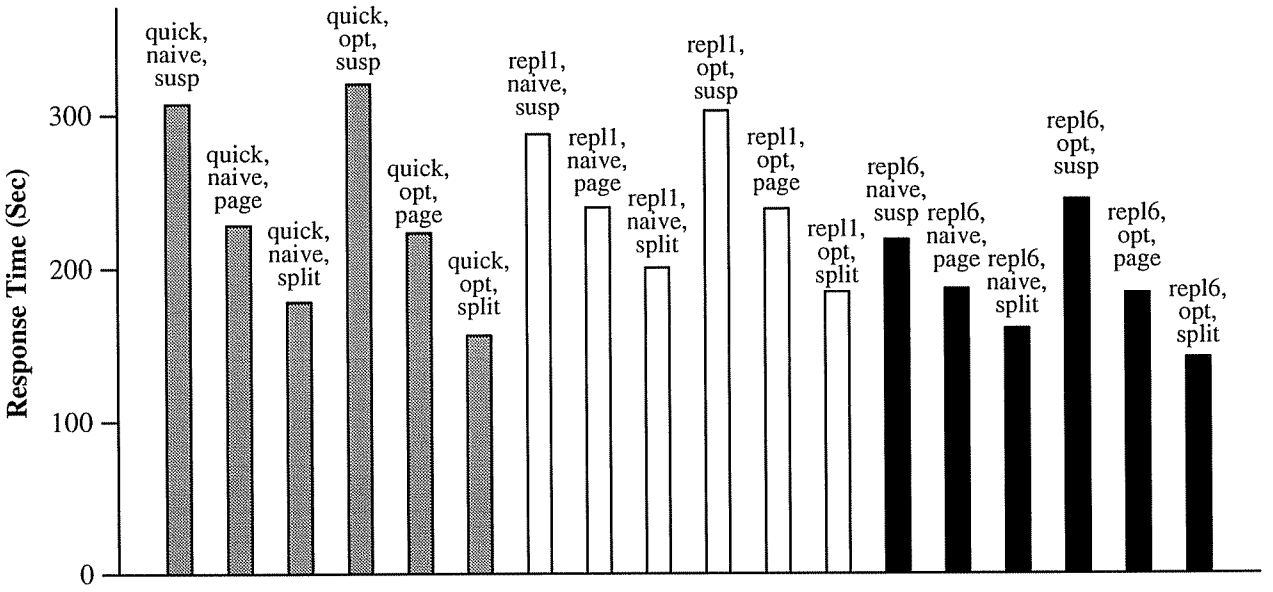


Figure 6: Response Times for Baseline Experiment

The response times given in Figure 6 are also listed in Table 7, which is organized to highlight the performance trade-offs associated with the different merge-phase adaptation strategies. For example, with Quicksort and naive merging, the first row of Table 7 shows that the average response times are, respectively, 307 and 228 seconds when suspension and paging are used, while only 178 seconds are required in the case of dynamic splitting, as indicated by the third column of the same row. All three merge-phase adaptation strategies have minimal delays in responding to memory fluctuations, as they allow an external sort to release the memory occupied by its input buffers immediately upon request, before taking merge-phase adaptation strategy-specific actions to adjust to the memory shortage. In fact, the observed average delays for the merge phase are consistently less than 1 msec; for this reason, we do not show the delays caused by the merge-phase adaptation strategies here. In terms of response times, however, there is a marked difference between the performance of the three merge-phase adaptation strategies. Among the three, suspension (*susp*) has the worst response times because it does not allow an external sort to make any progress when there is a memory shortage. Paging (*page*) and dynamic splitting (*split*), in contrast, both enable an external sort to keep progressing, which is why they are faster than *susp*. When there is a memory shortage, *page* incurs extra I/Os in paging its input buffers. This is a better alternative than *susp*, but the penalty of paging can be high because the number of extra I/Os is proportional to the extent of the memory shortage. In the case of *split*, an external sort deals with memory shortages by initiating a merge step that fits the remaining memory. This reduces the number of input runs for subsequent merge steps, thereby making them less vulnerable to memory fluctuations. Moreover, *split* is able to take advantage of excess buffers when they become available by switching to a merge step that combines more runs. This is why, as expected, *split* is able to produce shorter response times than *page*.

Next, we evaluate the trade-offs among the in-memory sorting methods. To facilitate interpretation of the results, we reorganize Table 7 into Table 8 to highlight the impact of the different in-memory sorting methods. Also included

	susp	**page**	**split**
Response Time (sec)			
quick,naive	307	228	178
quick,opt	320	223	156
repl 1,*naive*	287	239	200
repl 1,*opt*	302	238	184
repl 6,*naive*	218	186	160
repl 6,*opt*	244	183	141

Table 7: Performance of Merge-Phase Adaptation Strategies

	quick	repl1	repl6
# of Runs	154	99	103
Split-Phase Duration (sec)	28	77	33
Split-Phase Delay (sec)			
mean	0.056	0.032	0.020
maximum	0.180	0.149	0.147
Response Time (sec)			
naive,susp	307	287	218
naive,page	228	239	186
naive,split	178	200	160
opt,susp	320	302	244
opt,page	223	238	183
opt,split	156	184	141

Table 8: Performance of In-Memory Sorting Methods

in the table are the average duration of the split phase, the average number of sorted runs produced in this phase, and the average split-phase delay (the time that each in-memory sorting method takes to respond to memory shortages). In the table, all the algorithms that employ the same in-memory sorting method have the same average number of runs, split-phase duration and split-phase delay, as the merging strategies and merge-phase adaptation strategies concern only the merge phase and not the split phase. Due to the much longer split-phase durations that result from excessive disk seeks, as seen in Section 5.1, replacement selection (*repl*1) is almost always slower than Quicksort (*quick*) and replacement selection with block writes (*repl*6). The only exceptions occur when *quick* is used in conjunction with *susp*, which produces the worst response times. The reason is because *quick* generates many more sorted runs than *repl* 1, making the external sorts much more vulnerable to memory shortages. When used with *susp*, the much slower merge phase thus overwhelms any savings that *quick* derives from a shorter split phase. The results also clearly indicate that, as for fixed memory allocations, *repl* 6 outperforms both *repl* 1 and *quick*: *repl* 6 is faster than *repl* 1 due to *repl* 6's much shorter split-phase duration, while *repl* 6 outperforms *quick* because *quick* generates more runs and hence results in more merge steps in the merge phase. Moreover, among the three in-memory sorting methods, *quick* is the least responsive in reacting to memory fluctuations. As discussed in Section 3.1, Quicksort results in considerable split-phase delays because it has to sort all of the memory-resident tuples and then write them out before it can release its buffers. In contrast, the two replacement selection procedures lead to short split-phase delays since they need only to write out just enough pages of tuples from the heap to satisfy a waiting memory request. *repl* 1 is slower than *repl* 6 in reacting to memory shortages because *repl* 1 writes out only a single page of tuples each time, keeping the remaining memory pages filled with tuples from the source relation. As a result, every memory request, regardless of its size, encounters a delay while waiting for the external sort to free its memory. In comparison, *repl* 6 flushes a block of 6

pages to disk each time. After a flush, it takes a while before the 6 free memory pages can be filled with source relation tuples again, at which point another flush is carried out. Hence *repl* 6 leaves a few free buffers around most of the time. These existing free buffers help to reduce the number of pages that have to be written out in order to satisfy a memory request, thereby shortening the delay it experiences. In cases where the memory requests are small, the free buffers alone usually suffice to meet the requirement of the requests, so they need not be delayed at all. This explains the difference in average split-phase delays in Table 8 due to the choice of *repl* 1 versus *repl* 6.

We now examine the two alternative merging strategies. Table 9 focuses on the relative merits of naive merging (*naive*) versus optimized merging (*opt*). The table shows that *opt* is better than *naive* when used in conjunction with paging or dynamic splitting, while the reverse is true when the merge-phase adaptation strategy is suspension. Recall that *naive* combines more runs in the first merge step, leaving fewer runs to the final merge step. This makes the external sort more vulnerable to memory shortages in the first step than in the final step. In contrast, *opt* attempts to minimize cost by merging as few runs in the first step as possible without increasing the number of merge steps. The result is that the external sort is less vulnerable to memory shortages in the first step, but becomes more vulnerable in the final step due to the larger number of runs that are left until the final step. Since the final step (which has to process all of the tuples in the relation) typically lasts longer than the first step, the net effect is that *opt* makes an external sort more vulnerable to memory shortages than *naive*. Thus, whether *opt* is better than *naive* depends on how much time *opt* saves by merging fewer runs in the first step, as compared to the penalty caused by exposing the external sort to memory shortages for a longer period of time. With suspension, an external sort does not make any progress at all when there is a memory shortage, so the penalty of *opt* outweighs its advantage; this explains why *opt* performs badly with *susp*. In contrast to suspension, paging and dynamic splitting enable an external sort to keep progressing during periods of memory shortages. Thus, the penalty of *opt* is not as high, leading *opt* to be beneficial with both paging and dynamic splitting.

To summarize the results of this experiment, we can reach the following conclusions about cases where the relation to be sorted is significantly larger than the available memory. First, dynamic splitting is superior to paging, while suspension results in very large response times and should be avoided. Second, among the three in-memory sorting methods, *repl* 6 combines *repl* 1's advantages (producing long sorted runs and short split-phase delays in responding to memory shortages), together with the short-split-phase-duration characteristic of *quick*, making *repl* 6 the in-memory sorting method of choice here. Finally, provided paging or dynamic splitting is used, *opt* is beneficial and preferable to

	naive	opt
quick,susp	307	320
quick,page	228	223
quick,split	178	156
repl 1,*susp*	287	302
repl 1,*page*	239	238
repl 1,*split*	200	184
repl 6,*susp*	218	244
repl 6,*page*	186	183
repl 6,*split*	160	141

Table 9: Response Time (seconds) for Merging Strategies

naive. Overall, *repl* 6,*opt*,*split* appears to be the most promising algorithm, followed by *repl* 6,*naive*,*split* and *quick*,*opt*,*split*.

5.3. M to ||*R*|| Ratio

In the next experiment, we study the sensitivity of the external sort algorithms to different ratios of memory size to relation size. This is achieved by varying **M**, the total number of buffers, while keeping the other parameters constant at their settings of Tables 2 and 3. In particular, the memory fluctuation rates are the same as in the baseline experiment, and ||*R*|| remains at 20 MBytes so that an increase in **M** results in an increase in the memory to relation size ratio. For this experiment, the in-memory sorting methods examined will be limited to Quicksort (*quick*) and replacement selection with block writes (*repl* 6); *repl* 1 will not be considered further because it produces only slightly fewer runs than *repl* 6 while incurring the penalty of a much longer split phase. We will also exclude suspension, since it renders an external sort inactive when memory shortages occur, and is therefore not as effective as paging or dynamic splitting as the baseline experiment showed.

We first examine the performance of the two remaining merge-phase adaptation strategies, dynamic splitting (*split*) and paging (*page*). The merge-phase delays produced by both of the merge-phase adaptation strategies are less than 1 msec for the entire range of **M** values that we examined, hence we do not show the merge-phase delays here. Figure 7 plots the response times for the algorithms that employ replacement selection with block writes (*repl* 6) as a function of **M**. The algorithms that use Quicksort follow the same trends as those in Figure 7 and are not shown here. Note that, with the workload parameter settings for this experiment, the range of the average amount of memory available for external sorts here is the same as the range of memory sizes used in Section 5.1. Figure 7 shows that *split* consistently performs at least as well as *page*: for **M** = 0.1 MBytes, *split* is about 30% faster than *page*, but the difference between their response times narrows considerably when **M** increases; for **M** > 0.6 MBytes, the difference is

insignificant. The reason for this trend is that an increase in **M** leads to an increase in the average length of the sorted runs produced in the split phase, producing a corresponding decrease in the number of runs that have to be merged. This makes the external sorts less vulnerable to memory shortages during the merge phase, so there are fewer occasions when paging or dynamic splitting are required. In contrast, a small **M** will increase an external sort's reliance on its merge-phase adaptation strategy, which is why the performance differences between *split* and *page* are more pronounced for smaller **M** values.

We now turn our attention to the in-memory sorting methods, Quicksort (*quick*) and replacement selection with block writes (*repl* 6). The response time of the algorithms based on dynamic splitting, the most promising merge-phase adaptation strategy, are shown in Figure 8. The results indicate that *repl* 6 is about 5% faster than *quick* when **M** = 0.1 MBytes. As **M** increases, the response times of the two in-memory sorting methods converge gradually; beyond **M** = 0.9 MBytes, *repl* 6 and *quick* have about the same response times. This trend was also observed in the first experiment where external sorts executed with fixed memory allocations throughout their lifetimes. Compared to Figure 5 for the first experiment, however, the response time difference between *quick* and *repl* 6 at the left-side side of Figure 8 is noticeably smaller. The reason is because, by sorting and writing out the entire contents of its memory in response to a memory shortage, *quick* frees up all of its buffers so that additional memory requests that arrive while the current run is being generated can be satisfied without requiring further actions on the part of the external sort. *repl* 6, in contrast,

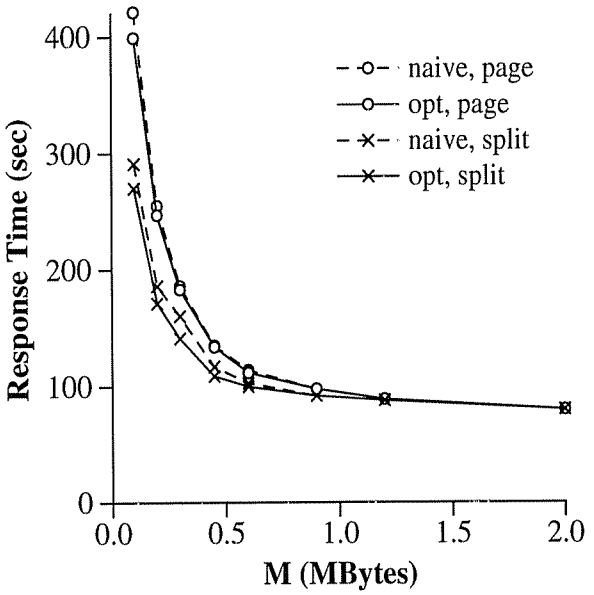


Figure 7: *repl* 6 (**M** to ||*R*|| Ratio)

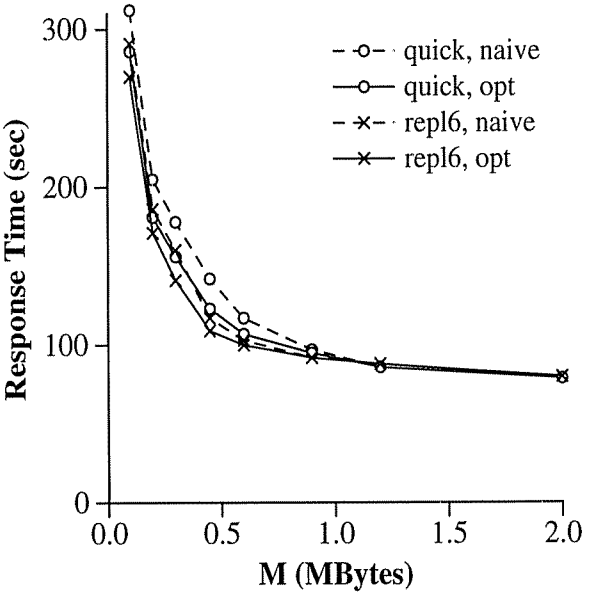


Figure 8: *split* (**M** to ||*R*|| Ratio)

frees up just enough memory to meet the demands of a waiting memory request. When the next memory request arrives, *repl* 6 is forced to write out another block of buffers. Consequently, *repl* 6 experiences more interference from competing memory requests than *quick*. This explains *quick*'s performance gains on *repl* 6 for **M** < 0.9 MBytes where external sorts are sensitive to memory fluctuations, though *repl* 6 still yields faster response times than *quick* here. Besides its generally shorter response times, another factor that favors *repl* 6 over *quick* is *repl* 6's responsiveness to memory fluctuations. Figure 9 gives the mean and maximum split-phase delays for the two in-memory sorting methods. The figure shows that the split-phase delays caused by both sorting methods grow as **M** increases. For *repl* 6, the split-phase delay is proportional to **M** simply because, as **M** increases, so does the size of the memory fluctuations (due to the way our workload is defined). This leads to an increase in the average number of pages that *repl* 6 has to write out to satisfy each memory shortage. In the case of *quick*, this growth is due to the increased size of the sorted runs, which take longer to sort and to write out. Figure 9 also shows that, besides consistently being slower in responding to memory fluctuations, the split-phase delays produced by *quick* also grow at a much faster rate than that of *repl* 6. In particular, at **M** = 2 MBytes, the mean delay of *quick* reaches almost 0.4 second, which is 4 times as long as that of *repl* 6. Considering both the response times and the split-phase delays produced by the two in-memory sorting methods, *repl* 6 appears to be superior to *quick* overall.

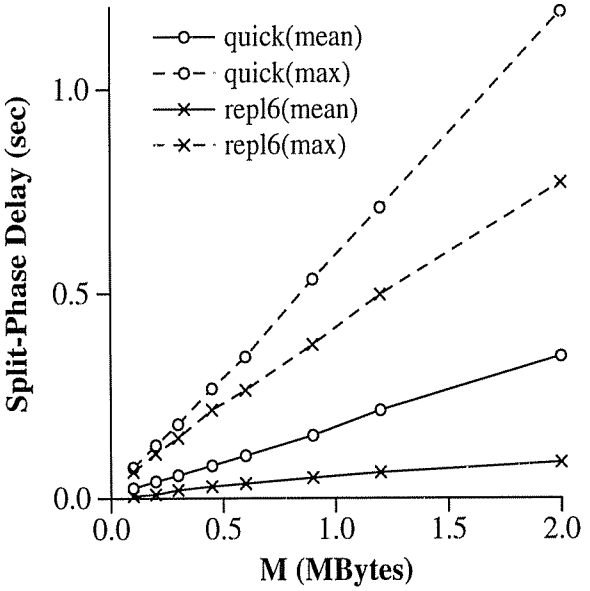


Figure 9: Delay of In-Memory Sorting Methods

Finally, we evaluate the two alternative merging strategies. Figures 7 and 8 (shown previously) show the response times for both algorithms that employ naive merging (*naive*) and algorithms that employ optimized merging (*opt*). While these figures cover only a subset of the entire space of eight alternative algorithms, the remaining algorithms give similar results and are not shown. Like the difference between *split* and *page*, there is a significant difference between *naive* and *opt* for small **M** values. When **M** = 0.1 MBytes, *naive* results in a slightly over 5% increase in response time compared to *opt*. The difference between the two merging strategies diminishes steadily as **M** increases until, at **M** = 0.9 MBytes, both strategies yield identical performance. Again, this behavior is similar to what we observed in the static memory allocation case, so we shall not elaborate further on the cause.

The results of this experiment support our baseline experiment's conclusion that, overall, dynamic splitting yields better performance than paging. Moreover, *repl* 6 is superior to *quick*, both in terms of response time and responsiveness to memory fluctuations. Finally, among the two merging strategies, optimized merging is the preferred choice.

5.4. Magnitude of Memory Fluctuations

Our next experiment is designed to explore the sensitivity of the memory-adaptive mechanisms to different memory fluctuation magnitudes. Instead of an environment where most of the contenders for system memory are small memory requests, as in previous experiments, here we examine a situation where most of the memory requests are large. To achieve this, we interchange the arrival rate and duration of the small and the large requests, so that now λ_{small} = 0.1 request/second, μ_{small} = 5 seconds, λ_{large} = 1 request/second, and μ_{large} is 0.8 second. All the other parameters are set as in the previous experiment.

Figure 10 highlights the performance difference between dynamic splitting (*split*) and paging (*page*). Compared to the performance results obtained for the previous experiment (shown in Figure 7), we note that here both *split* and *page* produce longer response times. Moreover, the difference in response time between *split* and *page* is greater here. These changes are due to the increased frequency of large memory requests, which reduces the number of buffers that are available to the external sorts. This leads to an increase in the number of merge steps in the merge phase, and lengthens the response time of *split*. For example, for **M** = 0.1 MBytes, *split*'s response time is now 345 seconds, whereas it was only 280 seconds previously. In the case of *page*, there are additional factors that adversely affect the performance of the external sorts: When the actual number of buffers that an external sort has is smaller than the buffer requirement of an executing merge step, the penalty in extra I/Os that paging incurs is proportional to the extent of the

memory discrepancy. In this experiment, where memory availability fluctuates more widely, the penalty of paging is magnified by the larger memory discrepancies. Moreover, an external sort based on paging is unable to utilize memory that is in excess of its initial memory allocation. This handicap causes paging to suffer from memory fluctuations; moreover, the larger the memory fluctuations, the greater an impact this handicap exerts on sort performance. Together, these two factors slow down the performance of *page* over and above the performance penalty already imposed by the larger number of merge steps. In particular, they account for the 120-second hike in *page*'s response time, from an average of 410 seconds in Figure 7 to an average of 530 seconds here, compared to the smaller 65-second increase in the case of *split*.

The performance results for the two in-memory sorting methods, Quicksort (*quick*) and replacement selection with block writes (*repl* 6), are shown in Figure 11. From the figure, it is apparent that the increase in the magnitude of memory fluctuations narrows the performance difference between *quick* and *repl* 6 as compared to the previous experiment (Figure 8). The reason is because here frequent large memory shortages force *repl* 6 to write out many memory-resident tuples each time. This hampers *repl* 6's ability to keep a large selection of tuples in memory and to write out only those tuples that have small key values, leading to shorter output runs. As a result, the number of runs that *repl* 6 produces becomes much closer to that of *quick*.

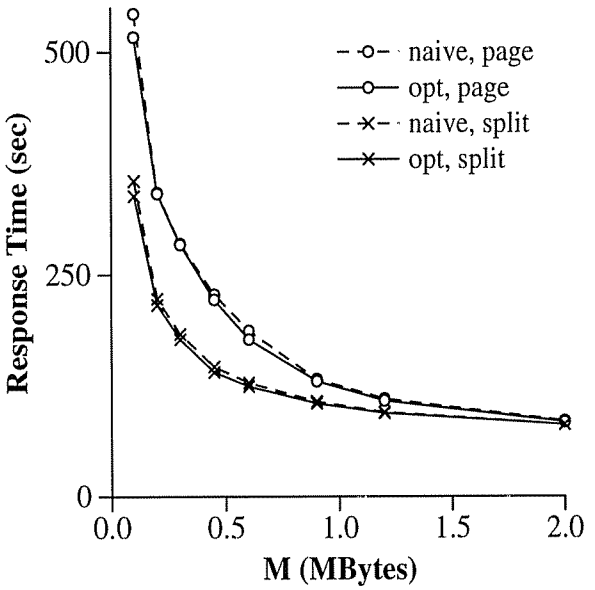


Figure 10: *repl* 6 (Memory Fluctuation Magnitude)

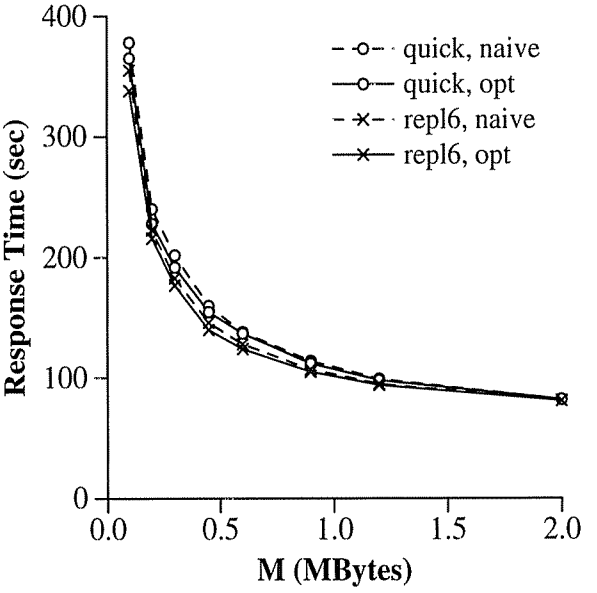


Figure 11: *split* (Memory Fluctuation Magnitude)

Finally, we examine how the change in memory fluctuation magnitude impacts the merging strategies. The response times of some algorithms that employ naive merging (*naive*) and others that are based on optimized merging (*opt*) are given in both Figures 10 and 11. In this experiment, where external sorts frequently experience large fluctuations in their allocated memory, the number of runs that an external sort selects for the first preliminary merge step during a split, whether according to *naive* or based on *opt*, often turns out to be sub-optimal because of significant changes that occur in the external sort's memory allocation during the preliminary merge steps. Thus, *opt* is not that much better than *naive* here, in contrast to the previous experiment where memory allocation was less volatile.

In summary, this experiment reveals that large fluctuations in memory availability accentuate the importance of the merge-phase adaptation strategy, while diminishing the differences between the alternative in-memory sorting methods and merging strategies. Again, the results reinforce our previous conclusions about the usefulness of dynamic splitting in dealing with memory fluctuations.

5.5. Rate of Memory Fluctuations

Our last experiment for external sorts is designed to investigate how different memory fluctuation rates might affect the relative performance of the merge-phase adaptation strategies and the in-memory sorting methods. We vary the fluctuation rates by first lowering them to λ_{small} = 0.2 request/second and λ_{large} = 0.02 request/second. To ensure that this does not change the average available memory from that in the baseline experiment, the duration of the memory requests are prolonged by the same factor, i.e. μ_{small} = 4 seconds and μ_{large} = 25 seconds. Next, we raise the rate of fluctuation by a factor of 25, setting λ_{small} = 5 requests/second, μ_{small} = 0.16 second, λ_{large} = 0.5 request/second, and μ_{large} = 1 second. The performance results are presented in Figures 12 and 13.

Figures 12 and 13 show the performance results of four of the external sort algorithms for both the fast and slow memory fluctuations (labeled *fast* and *slow*, respectively, in the figures). In the figures, the solid lines show the response times of the algorithms, while the dotted lines give the split-phase durations. The solid curves in these two figures show that, while the relative performance of the algorithms remains the same as in our previous experiments, the change in memory fluctuation rate does have an impact on the response time of the algorithms for small **M** values. The figures also indicate that when **M** is large, increasing fluctuation rate has little impact on the response time because external sorts are not sensitive to memory fluctuations in this region, as discussed in previous experiments. As **M** is decreased, external sorts become vulnerable to memory fluctuations, and switching the fluctuation rate parameters from

their slow settings to their fast settings increases the response times of all four external sort algorithms shown here. For paging, the reason is that, when memory allocation increases after a shortage, paging requires some time before the pages that have been swapped out can be brought back in to fill the newly allocated memory. During this time, the effective number of buffers used is less than the allocated memory. Therefore, when the memory fluctuation rate increases, the effective memory utilization goes down and this leads to longer response times. In the case of dynamic splitting, the external sorts react to changes in memory allocation by switching merge steps. Each switch incurs some overhead in bringing the input and output buffers of the new step into memory, so dynamic splitting is also adversely affected by increased memory fluctuations. After **M** = 0.3 MBytes, however, further reduction in **M** narrows the gap between the response times for the slow and the fast fluctuation settings. This phenomenon is due to the fact that, as **M** decreases, so does the magnitude of the memory fluctuations, and hence the performance penalty imposed by these fluctuations. This is why external sorts suffer less from the more frequent fluctuations when the buffer size is very small than when **M** is slightly larger. In contrast to the merge-phase adaptation strategies, the in-memory sorting methods are insensitive to changes in the fluctuation rate, as indicated by the dotted lines in Figures 12 and 13, since the average available memory remains the same despite changes in the fluctuation rate.

To summarize, the results of this experiment lead us to conclude that, over a wide range of memory fluctuation rates, the algorithm *repl*6,*opt*,*split* delivers the best overall performance among those that we considered. Dynamic

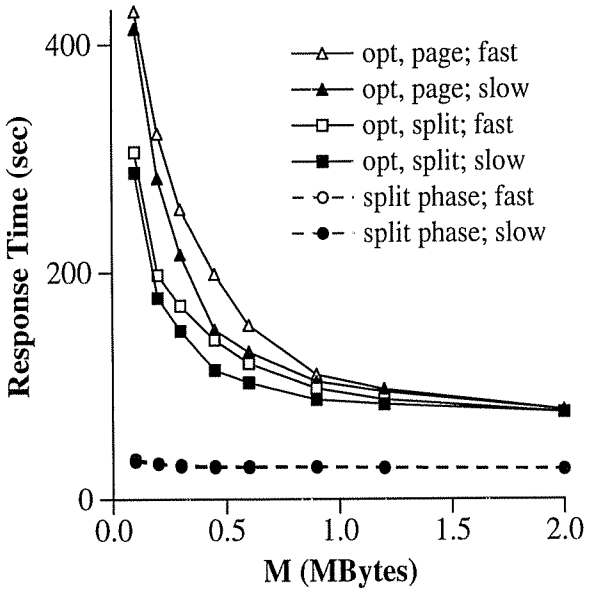


Figure 12: *quick* (Memory Fluctuation Rate)

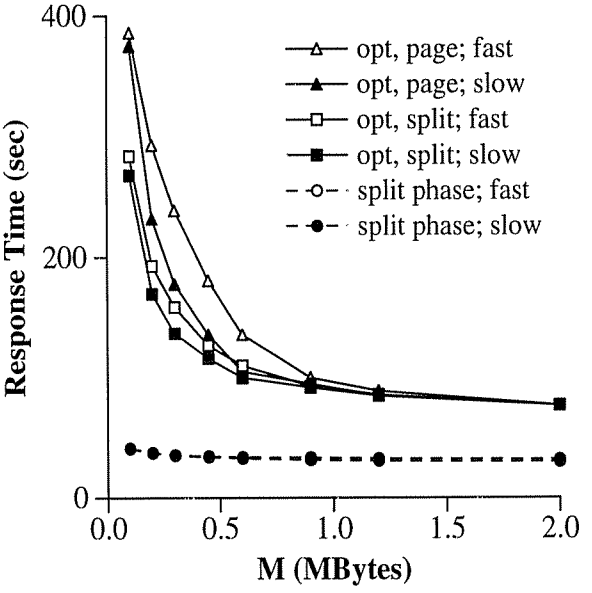


Figure 13: *repl* 6 (Memory Fluctuation Rate)

splitting therefore appears to be a promising merge-phase adaptation strategy in practice.

5.6. Other Issues

The experiments in this section showed that a disadvantage of Quicksort is its relatively long delay in releasing memory when required to do so. We have also studied a second variation of the Quicksort algorithm that attempts to overcome this limitation. Recall that Quicksort builds a list of (key, pointer) pairs, sorts this list, and then retrieves tuples from the memory-resident relation pages via the pointers. The second Quicksort variation that we studied allows an external sort to release buffers that are occupied by relation pages upon request, provided that none of the tuples in those buffers have been written out to the current output run[3]. When the external sort attempts to retrieve a tuple from a page and finds that it is no longer in memory, the external sort simply passes over this tuple, leaving it for the subsequent run. The pages that are released will then be refetched when the external sort starts reading in relation pages for the next sorted run. Preliminary experiments with this second Quicksort variation showed that it leads to only a slightly shorter delay in releasing memory compared to the Quicksort algorithm that is described in Section 3.1 and used in the bulk of this paper. The reason for the small difference is that the time needed to sort the (key, pointer) list is relatively short compared to the time required to write out the sorted run. Moreover, once the external sort begins to write out the sorted run, retrieving the sorted tuples will cause the external sort to "dirty" its memory pages randomly (assuming that the key values in the source relation are randomly distributed), quickly making all its pages unavailable for release until all of the tuples in those pages have been written out. Also, this slight reduction in delay is achieved at the expense of a much longer sort response time because, when memory is released, relation pages in the released memory will have to be refetched later. This results in increased I/Os and prolongs the split phase. Finally, releasing memory pages before their contents can be written out also shortens the sorted runs, leading to an increase in the number of runs that have to be merged, hence lengthening the merge phase. We therefore did not present performance results for this second Quicksort variation in this study.

[3] It is possible for an external sort to release its buffers even after some of the tuples in those buffers have been written out, as long as the sort operator keeps track of which particular tuples have been written and which have not, so that tuples that have already been written out are not processed again when the released pages are refetched later for processing. This complicates the codes for the external sort algorithm, and increases run-time overheads. We therefore did not pursue this option.

6. Sort-Merge Joins

Sort-merge join is a join algorithm employed by many existing database systems. Although recent work has shown hash join to often be superior to sort-merge join in performing ad-hoc join operations [Brat84, DeWi84, Shap86], sort-merge join is still useful under certain conditions, e.g. when significant data skew is present, or when the results need to be presented in sorted order [Grae93]. Hence sort-merge join is likely to continue to be offered as one of the alternative join algorithms in future DBMSs. In this section, we address the issue of extending the techniques that we have explored earlier to handle sort-merge joins, thus making them memory-adaptive.

6.1. Memory-Adaptive Sort-Merge Joins

Like the external sort algorithm, a sort-merge join consists of a split phase and a merge phase. The split phase divides the two source relations into two separate sets of sorted runs. This is exactly as in the case of external sorts, except that now there is an additional relation to split. The in-memory sorting methods that we have examined, namely replacement selection, Quicksort and replacement selection with block writes, can thus be used here without any changes. In the merge phase, runs from both relations are merged concurrently, and sorted tuples from the two relations are joined directly as they are merged. In the event that the total number of runs from the two relations exceeds the available memory, the final merge step, i.e. the step that combines all the runs from both relations and produces the join results, has to be split. The preliminary step that is created as a result of this split will work on one of the relations, merging some of its existing runs into a longer sorted run. Since there are two relations, the preliminary step has a choice of which relation to merge. To minimize the cost of the preliminary step, the chosen relation is the one that will lead to a smaller total input size for the merge step. For example, if the preliminary step has to merge 15 runs, the number of pages in the smallest 15 runs from each individual relation is summed, and the relation with the smaller sum is selected. Any of the three merge-phase adaptation strategies, i.e. suspension, paging and dynamic splitting, can be used to adapt the sort-merge join to memory fluctuations during the merge phase. However, the naive and optimized merging strategies have to be modified slightly in order to comply with the requirement that each preliminary step merges only runs from the same relation.

During a split, the desired number of runs to be merged in the preliminary step is determined by either the naive or the optimized merging strategy. In some cases, one or both of the relations may not have that many runs. To illustrate this point, consider a situation where a sort-merge join has 11 buffers, and the two relations are split into 5 runs

and 14 runs, respectively. Both naive merging and optimized merging will attempt to merge 10 runs in the preliminary step so that the remaining runs can be merged all at the same time. Unfortunately, the first relation has only 5 runs, so it cannot be chosen for the preliminary step. In such cases, we modify the naive and optimized merging strategies to select the relation that has more runs for the preliminary step, in order not to introduce more steps to the merge phase.

6.2. Experiment and Results

Since the same basic mechanisms work for both external sorts and sort-merge joins, we expect the relative performance trade-offs between the different in-memory sorting methods, merging strategies and merge-phase adaptation strategies to be the same in both cases. To confirm this, we present one set of experimental results. In this experiment, each sort-merge join involves two relations, **R** and **S**, of sizes $||R||$ and $||S||$, respectively. We let $||R||$ be 2 MBytes and $||S||$ be 20 MBytes to simulate a primary key-foreign key join. Moreover, **M**, the total number of buffers, is varied while the other parameters are kept constant at their settings of Tables 2 and 3.

The join response times for this experiment are plotted in Figures 14 and 15. As expected, the performance trends for each algorithm, as well as the relative trade-offs between the different algorithms, are virtually identical to what we saw for external sorts in Figures 7 and 8: Dynamic splitting is clearly the merge-phase adaptation strategy of choice, while replacement selection with block writes is the winner among the in-memory sorting methods. Moreover,

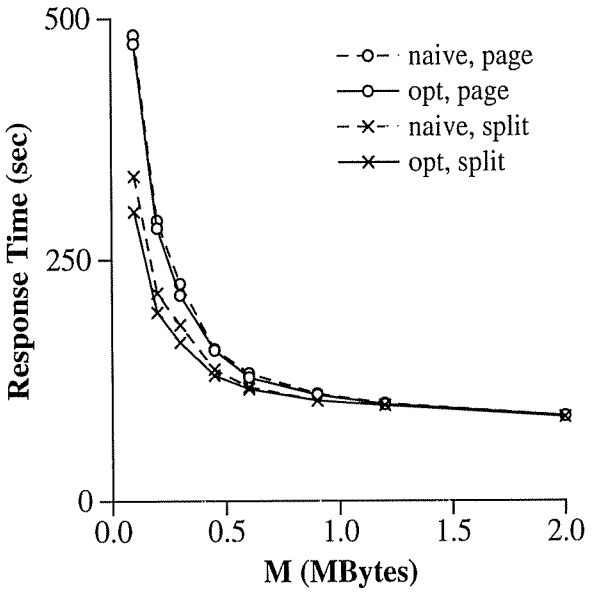


Figure 14: *repl* 6 (Sort-Merge Joins)

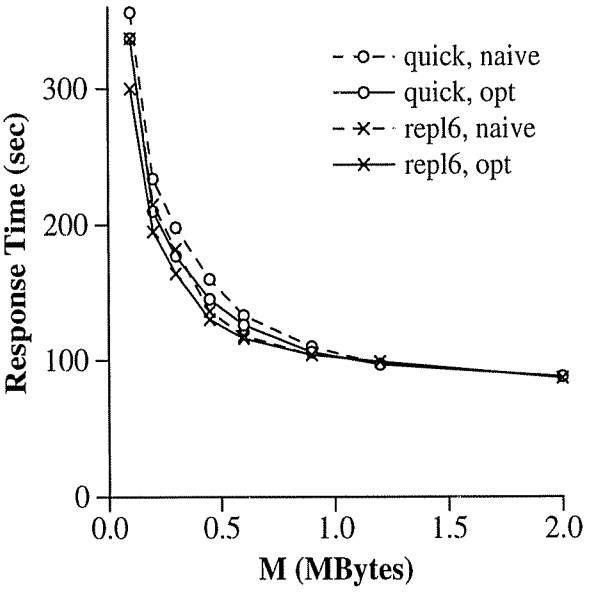


Figure 15: *split* (Sort-Merge Joins)

optimized merging outperforms naive merging.

In summary, the results of this experiment confirm that the merge-phase adaptation strategies, in-memory sorting methods and merging strategies that we have explored in the context of external sorts are equally applicable to sort-merge joins. Therefore the combination of *repl*6,*opt*,*split* provides an effective means to do sort-merge joins in the face of fluctuations in memory availability.

7. Conclusion

In this paper, we have addressed issues related to query execution in situations where the amount of memory available to a query may be reduced or increased during its lifetime. These situations will arise in real-time or goal-oriented database systems, where memory may be appropriated from a query to meet the buffer requests of higher-priority transactions, and where additional memory may be made available when other queries complete and free their buffers. In particular, we considered the specific problem of external sorts, which require large numbers of buffers to execute efficiently and are thus especially susceptible to fluctuations in memory availability. Simple approaches that react to a reduction in an external sort's allocated memory by suspending the sort altogether, or by paging the buffers of the sort into and out of the remaining memory, may lead to under-utilization of system resources or thrashing. Furthermore, these approaches do not allow external sorts to make use of extra memory (beyond their initial memory allocation) that may become available during their lifetime. There is therefore a need for more sophisticated approaches that enable external sorts to adapt to memory fluctuations.

An external sort consists of two phases: the split phase fetches portions of the relation into memory, where they are sorted and then written out as sorted runs, and the merge phase combines the resulting runs into the sorted result. The merge phase consists of one or more merge steps, each of which combines a number of runs into a single, longer run. We studied Quicksort and replacement selection, two common in-memory sorting methods that are used for the split phase. In addition, we also studied a variation of replacement selection that uses block writes to reduce disk seeks. All three in-memory sorting methods allow external sorts to respond to memory shortages by writing sorted tuples out to reduce their buffer usage; when memory increases, the newly allocated memory is used to fill more relation pages. In contrast to the in-memory sorting methods, the merge phase is not as easily adapted to memory fluctuations. We therefore examined hybrid approaches that allow external sorts to adapt to memory fluctuations only in the split phase, letting the DBMS suspend the external sorts or page their buffers if memory shortages occur while they are in the

merge phase. In addition, we proposed a merge-phase adaptation strategy, called *dynamic splitting*, that enables external sorts to better respond to memory shortages and to exploit excess memory in the merge phase by involving the sorts in adapting to memory fluctuations. This strategy splits an executing merge step into sub-steps that fit within the remaining memory when a shortage occurs, and it combines existing merge steps into larger steps (i.e. steps that merge more runs at once) to take advantage of excess buffers when they become available.

To understand how effective the different in-memory sorting methods and merge-phase adaptation strategies are in dealing with memory fluctuations, we constructed a detailed simulation model. A series of experiments revealed that, when the available memory is small relative to the relation to be sorted, the merge-phase adaptation strategy is the dominant performance factor. Among the merge-phase adaptation strategies, dynamic splitting outperforms paging; the smaller the size of memory is relative to the relation, the more significant the performance difference between the two strategies becomes. The third merge-phase adaptation strategy, suspension, consistently yields unsatisfactory response times. Thus dynamic splitting appears to be an attractive strategy for sorting large relations. Our results also showed that replacement selection with block writes is the preferred in-memory sorting method. Besides consistently producing response times that are at least as fast as Quicksort, replacement selection with block writes also makes external sorts more responsive, compared to Quicksort, in releasing memory when required to do so. Overall, our results indicate that the combination of dynamic splitting and replacement selection with block writes enables external sorts to deal effectively with memory fluctuations.

Like external sorts, sort-merge joins are vulnerable to memory fluctuations due to their large memory requirements. The sort-merge join algorithm is also made up of a split phase and a merge phase. The split phase divides each of the two operand relations into sets of sorted runs. In the merge phase, runs from both relations are combined concurrently, and the sorted tuples from the two relations are joined directly. If there are too many runs to be merged at the same time, preliminary steps are created to merged some of the existing runs from one (or both) relation(s) into longer sorted runs. The same techniques that we examined in the context of external sorts can be applied to sort-merge joins in order to make them memory-adaptive. Moreover, the same relative performance trade-offs apply to both external sorts and sort-merge joins. Therefore, the combination of dynamic splitting and replacement selection with block writes is also an effective means to adapt sort-merge joins to fluctuations in memory availability.

For future work, we intend to explore alternative strategies for adapting the merge phase of an external sort. One such strategy is to dynamically adjust the buffer size (i.e., the I/O block size) according to memory availability; a

combination of buffer size adjustment and dynamic splitting would likely yield an even more effective solution than dynamic splitting alone, particularly for large memory sizes. Another important avenue for future work is the development of strategies for appropriately utilizing adaptive sort and join methods in the context of adaptive query plans for complex (e.g., many-way join) queries.

References

[Abbo88] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1988.

[Bitt88] D. Bitton, J. Gray, "Disk Shadowing", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1988.

[Blas77] M.W. Blasgen, K.P. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal, Vol. 16,4*, 1977.

[Brat84] K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proc. of the 10th Int. Conf. on Very Large Data Bases*, August 1984.

[Brow93] K.P. Brown, M.J. Carey, M. Livny, "Managing Memory to Meet Multiclass Workload Response Time Goals", *Proc. of the 19th Int. Conf. on Very Large Data Bases*, August 1993.

[DeWi84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M. Stonebraker, D. Wood, "Implementation Techniques for Main Memory Database Systems", *Proc. of the ACM 1984 SIGMOD Conf.*, June 1984.

[DeWi90] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I Hsiao, R. Rasmussen, "The Gamma Database Machine Project", *IEEE Trans. on Knowledge and Data Engineering, Vol. 2,1*, March 1990.

[DeWi91] D.J. DeWitt, J.F. Naughton, D.A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", *Proc. of the Int. Conf. on Parallel and Distributed Information Systems*, December 1991.

[Ferg93] D. Ferguson, C. Nikolaou, L. Georgiadis, "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems", *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems*, January 1989.

[Grae90] G. Graefe, "Parallel External Sorting in Volcano", *Technical Report CU-CS-459-90*, University of Colorado, Boulder, March 1990.

[Grae93] G. Graefe, A. Linville, L.D. Shapiro, "Sort versus Hash Revisited", *IEEE Transactions on Knowledge and Data Engineering*, to appear, 1993.

[Hari90] J. Haritsa, M. Carey, M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of the 1990 ACM PODS Symposium*, April 1990.

[Huan89] J. Huang, J.A. Stankovic, D. Towsley, K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing", *Proc. of the 1989 IEEE 10th Real-Time Systems Symposium (RTSS)*.

[Kim91] W. Kim, J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling", *Proc. of the 1991 IEEE 12th Real-Time Systems Symposium (RTSS)*.

[Knut73] D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

[Kort90] H.F. Korth, N. Soparkar, A. Silberschatz, "Triggered Real-Time Databases with Consistency Constraints", *Proc. of the 16th Int. Conf. on Very Large Data Bases*, August 1990.

[Livn87] M. Livny, S. Khoshafian, H. Boral, "Multi-Disk Management Algorithms", *Proc. of the ACM 1987 SIGMETRICS Conf.*, May 1987.

[Livn90] M. Livny, "DeNet User's Guide, Version 1.5", *Computer Sciences Department, University of Wisconsin, Madison*, 1990.

[Pang93] H. Pang, M.J. Carey, M. Livny, "Partially Preemptible Hash Joins", *Proc. of the ACM 1993 SIGMOD Conf.*, May 1993.

[Ries78] D. Ries, R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems", *UCB/ERL Technical Report M78/22*, UC Berkeley, May 1978.

[RTS92] *Real-Time Systems, 4(3)*, Special Issue on Real-Time Databases, September 1992.

[Salz90] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, B. Vaughan, "FastSort: A Distributed Single-Input Single-Output External Sort", *Proc. of the ACM 1990 SIGMOD Conf.*, May 1990.

[Sarg76] R. Sargent, "Statistical Analysis of Simulation Output Data", *Proc. of the 4th Annual Symposium on the Simulation of Computer Systems*, August 1976.

[Shap86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems, Vol. 11,3*, September 1986.

[SIGM88] *SIGMOD Record, Vol. 17,1*, Special Issue on Real-Time Data Base Systems, S. Son, editor, March 1988.

[Teng84] J. Teng, R.A. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance", *IBM Systems Journal, Vol. 23,2*, 1984.

[Zell90] H. Zeller, J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. of the 16th Int. Conf. on Very Large Data Bases*, August 1990.