Memory Allocation
in
Adaptive External Sorting

Kevin L. Beck
B.S., Montana State University, 1985

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

January 1994

The thesis "Memory Allocation in Adaptive External Sorting" by Kevin L. Beck has been examined and approved by the following Examination Committee:

Calton Pu
Associate Professor
Thesis Research Adviser

David Maier
Professor

Leonard Shapiro
Professor
Portland State University

Acknowledgements

Many thanks to Len Shapiro. I owe him a great deal for his help and guidance on this project.

This work made extensive use of the DeNet simulation tool. Thanks to Miron Livny for permission to use this fine tool. Thanks also to Sequent Inc. for the use of their equipment and facilities. Many people have contributed ideas and helpful criticism during this project: Gary Graunke has been invaluable for deep background on how operating systems *really* work. Aki Fleshler and Sam DeFazio have both helped keep the project on track and relevant to real world problems in modern DBMS design. David Helman and Shensong Ni were enthusiastic participants in many long discussions shaping the ideas presented here. Goetz Graefe has contributed greatly to my understanding of DBMS implementation techniques.

Last but not least, I owe a great deal to my wife Lynda for her support and extraordinary patience during the long hours of this project.

Contents

List of Tables

List of Figures

Abstract

Memory Allocation
in
Adaptive External Sorting

Kevin L. Beck, M.S.
Oregon Graduate Institute of Science & Technology, 1994

Supervising Professor: Calton Pu

An adaptive external sorting algorithm is proposed where individual sorts vary the amount of memory they use during their lifetimes based on both the availability of memory and the needs of the sort. Algorithms for allocating memory among multiple concurrent instances of this algorithm are also investigated.

Most previous work in this area has investigated how to partition memory among queries that receive a static memory allocation when they begin. Other work has investigated adaptive algorithms, but has not considered strategies for partitioning memory among multiple concurrent instances of queries based on these algorithms. The proposed strategy combines these two approaches using a central agent, called the "memory broker", to dynamically partition memory among concurrent sorts that use an adaptive algorithm. The adaptive algorithm used in the sorts takes advantage of fluctuating memory allocations by intelligently adjusting the size of the sort's own working set to reduce page faulting. Simulation results are presented to show that this approach results in faster response time than traditional methods of memory allocation.

Chapter 1

Introduction

In a multi-user database, a crucial factor in performance is how memory is partitioned among competing queries. Although commercial database systems are beginning to incorporate more sophisticated algorithms for memory allocation, many current systems still use simple hueristics to allocate memory to memory-intensive operators such as sort and join. For example, there may be a tuning parameter that simply sets a maximum for the size to which an operator's workspace may grow [Ger93]. Another common solution is to allocate as much memory to an operator as it can use and depend on the operating system's virtual memory mechanism to manage it. This solution is simple to implement and may be appropriate for small data sets, but Verkamo has shown that it results in poor performance for large sorts [Ver86].

In this thesis we propose a combination of "adaptive algorithms" that are capable of modifying their behavior to compensate for varying amounts of memory available to them during their lifetime, and a "central agent" to partition memory among concurrent queries. We show that this combination of techniques gives substantial performance improvements over traditional methods of allocating memory. While there has been some recent work on adaptive algorithms and on central agents to partition memory among queries, we believe that our work is the first to show the performance implications of combining these two techniques.

In the area of memory allocation, one of the earliest investigations was by Chou and DeWitt [CD85], who proposed allocating different amounts of memory to different types of queries. More recent work on memory allocation has proposed strategies based on

marginal gains, where concurrent queries are given more or less memory based on their expected improvement in performance [NFS91, YC93]. While these ideas are indeed a step forward, they assume that queries will use a set amount of memory for their duration. This assumption can result in situations where more memory becomes available later, but ends up being wasted because all active queries have already received their allocations.

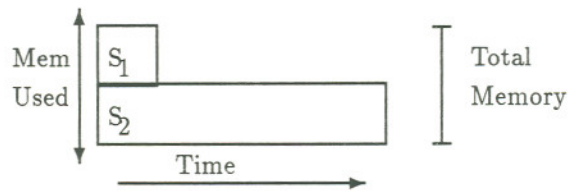
In the area of adaptive algorithms, Pang, Carey and Livny propose a family of algorithms for adaptive sorting and hashing [PCL93a, PCL93b], but do not investigate memory allocation strategies. Zeller and Gray implemented an adaptive hash join [ZG90], but relied on the operating system for memory allocation. Our work is the first of which we are aware that investigates the use of a central agent to allocate memory among competing instances of adaptive algorithms.

We call this central agent the "memory broker" to emphasize that its role is to moderate the exchange of memory among competing queries. Since sorting is such an important component of query processing, we concentrate our efforts on an adaptive external sort. We call our proposed algorithm BAS (Brokered Adaptive Sorting).

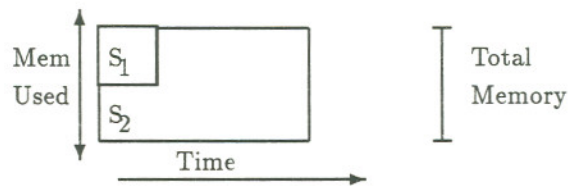
Our goal in this investigation is to increase the performance of the whole DBMS by reducing the average of the response time of all queries. As a simple example of the increased performance that BAS can provide, consider two sorts running concurrently, one sorting much more data than the other. Using a traditional method of memory allocation, both sorts would be allocated constant amounts of memory for their duration. If the large sort is given all of memory in order to maximize its performance, then the sorts can not be run concurrently. One of the sorts would be delayed until the other had completed. Alternatively, if the larger sort is allocated only a portion of the total memory so that the smaller sort can run concurrently, then memory will be wasted after the smaller sort has completed. Our solution is to have the large sort use all, or nearly all, of memory when it is available, and to temporarily give up memory during the time that the smaller sort is executing.

An easy way to visualize the effects of memory allocation on the execution time of a query using an adaptive algorithm is to draw a space-time graph that shows each query

as a block that is as long as the query took to execute, and as tall as the memory in use at a given time. In Figure 1.1(a), we illustrate a small sort (labeled S_1) running concurrently with a large sort (labeled S_2) using static memory allocations.



(a) Sort Using Static Memory Allocation



(b) Sort Using Adaptive Memory Allocation

Figure 1.1: Space Time Diagrams Comparing Performance of Non-Adaptive and Adaptive Algorithms

Our strategy allows a sort to dynamically adjust the amount of memory it uses during its lifetime. In Figure 1.1(b), we show how the larger sort could use the memory that becomes available after the smaller sort finishes. This strategy allows the large sort to complete sooner, without affecting the small sort. In this example, the improvement in average response time for the two sorts is 29%. (See Appendix A for a detailed derivation of this example). Our simulations have shown that for more realistic workloads, an overall improvement of 10% to 20% is more typical (see Chapter 5).

This simple example illustrates how our method can reduce the average response time of all queries in the workload. The same ideas also work well for workloads involving more than two sorts. In addition, these ideas can be extended naturally to other memory intensive database algorithms including the various join algorithms. We discuss how an

adaptive hash join might be made to work with our memory broker in Section 1.3.

The remainder of this introduction is structured as follows: Section 1.1 contains a brief overview of the algorithm used for the external sort in BAS. Section 1.2 presents overviews of the algorithms used in the memory broker portion of the two variants of BAS.

1.1 Overview of Algorithm for Adaptive External Sort

The adaptive sort algorithm used by BAS is a modification of a classic external sort algorithm presented by Knuth [Knu73]. This classic algorithm begins by repeatedly reading a memory full of data from the input, sorting it, and writing it to disk as a temporary file called a *run*. Following the nomenclature used by Knuth we refer to these initial runs as "level-0" runs. After all of the data has been partitioned into runs in this manner, the sort reads data from some subset of the runs, merging the data in them into progressively larger runs. (The number of runs in this subset is generally referred to as the *fan in*.) Eventually, this merging process reduces the number of runs to a small enough number that all runs can be merged into the final output. Each round of merging a subset of runs is called a *merge phase*.

Our adaptive sort differs from the classic one in that it can use different amounts of memory at different points in its lifetime. The way it changes its memory allocation is to "check in" with the memory broker at points when it could change its memory allocation at low cost cost. These points are just before starting to create a new run and just before beginning a merge phase. Thus some runs may be larger by virtue of having been created when the sort was using more memory. Likewise, some merge phases merge more runs than others.

Conceptually, checking in consists of sending a message to the memory broker and waiting for a reply. In an actual implementation, checking in could be accomplished in any number of ways, e.g. through the use of signals and sockets. In Section 3.5 we present a more detailed discussion of possible implementations of checking in. In any event, the

amount of information exchanged at a checkpoint is quite small and algorithms do not check in very often.

Our algorithm contains two refinements of the main ideas presented above. The first refinement has to do with planning ahead during run creation so that the last few blocks of the input data can be held in memory rather than written to temporary files. The second also involves planning ahead, but during the merge phases to avoid unnecessary passes through the data. These refinements are presented in Sections 3.3.3.1 and 3.3.3.2 of the detailed description of the sort algorithm of BAS in Chapter 3.

1.2 Overview of Algorithms for Memory Broker

Recall that our goal is to lower the average response time of all queries in the DBMS. The way that BAS helps achieve this goal is to allow queries to adapt the amount of memory they use in response to changes in the system workload. The intent of this adaptation is to allow each query to use as much memory as possible at any given moment while ensuring that the total amount in use by all queries never exceeds the total amount available in the system.

To keep track of how much memory is in use and which query should receive memory as it becomes available, we propose a central agent that we call the "memory broker". There are any number of possible strategies that the memory broker might use in allocating memory to queries using adaptive algorithms. We simulated and compared two such strategies, plus a third strategy to compare against (the control in our experiments):

1. Equal Allocation. This algorithm is a simple memory allocation strategy that takes advantage of our adaptive external sort algorithm. Using this strategy, the memory broker tries to give each query an equal share of the available memory.

2. Marginal Gains. This is a more sophisticated algorithm that takes into account the needs of each query when allocating memory. Queries receiving memory allocations from a memory broker using marginal gains submit "bids" for how much each

additional block of memory would be worth to them. These bids are based on the expected IO savings of an additional block of memory.

3. Static allocation. In addition to our two algorithms that take advantage of our adaptive sort algorithm, we implemented a third memory allocation algorithm that is intended to be a baseline to measure how much the performance of a system might be improved by incorporating BAS. We call this algorithm static, since the amount of memory which it allocates to a sort is fixed for the life of the sort.

All variants of the memory broker keep track of how much memory is currently available in the system. At each checkpoint, a query sends a message to the memory broker requesting permission to use more memory or announcing that it has finished using memory. Depending on the needs of other queries in the system, the memory broker may allocate more, less or the same amount of memory in response to requests for more memory. Queries block, waiting acknowledgment of their requests. This gives the memory broker the option of holding a request until the memory necessary to satisfy it becomes available.

The equal allocation and marginal gains variants of BAS both include a tuning parameter, but one of the strengths of BAS is that this parameter turned out to be largely unnecessary. In Section 5.3 we show that the performance of BAS is equally good for a wide range of the possible values of this parameter.

1.2.1 Equal Allocation

In the equal allocation variant of the memory broker for BAS, we model a memory broker that attempts to allocate an equal amount of memory to every query in the system. The basic strategy is to give each query a maximum of $1/n^{th}$ of the total memory every time it checks in, where n is the number of queries currently active in the system. In Section 3.4.1 we present a more detailed description of this algorithm.

This simple algorithm proved to perform surprisingly well.

1.2.2 Marginal Gains

In our implementation of marginal gains, the memory broker maintains a "reserve" of memory to sell to queries that need it more than other active queries. There is a tunable parameter that determines the maximum size to which this reserve may grow. Of course, the reserve will grow to all of memory if there are no queries running.

To gather the information that it needs to decide when to buy and when to sell, the memory broker solicits a "bid" from each query that checks in. This bid is the amount by which an additional block of memory would reduce the total number of IO operations necessary to complete the query. (See Appendix B for the formula used to calculate bids and its derivation.) The memory broker keeps track of the average bid of all active queries. If the query currently checked in is bidding below the current average bid value, and the memory broker's reserve is not yet at its maximum size, then the memory broker will "buy" memory from the query to give to other queries that are bidding higher. Conversely, if the query is bidding higher than the current average, then the memory broker will sell it memory until either it exhausts its reserve or the query's bid drops below the average.

Surprisingly, marginal gains actually performs worse than equal allocation for most of the workloads that we tested. The reasons for this behavior are discussed in detail in Section 3.4.2.2.

1.2.3 Static Allocation

The static allocation strategy causes the memory broker to always allocate the same amount of memory to every query. This has the effect of making our adaptive sort algorithm act like an ordinary, non-adaptive external sort. Using this algorithm, the memory broker gives each query a maximum of $P_{st} \times M$ blocks of memory, where M is the total number of blocks of physical memory, and P_{st} is a tunable parameter $0 < P_{st} \leq 1$. If a new query is submitted to the system and all memory is already in use, the new query is queued until memory becomes available for it.

Static is intended to be similar to how many commercial databases have traditionally allocated memory to operators. Although commercial database systems are beginning to use more sophisticated methods, it is still common for them to allocate memory to an operator based on a constant or tuning parameter [Ger93].

1.3 How Our Ideas Could Apply to Joins

Although we chose to concentrate on sorts, we believe that our ideas could be extended to joins as well. In this section we present our thoughts on how our ideas could be extended to hash joins, but leave detailed investigation of adaptive joins for future research.

As mentioned above, queries "check in" with the memory broker at times when it is convenient for them to change memory allocation. For sorts, these times are:

1. Before creating a new run.
2. Before beginning a new merge phase.
3. When the sort is done (to hand memory used back to the system).

Our strategy for making hash joins adaptive would be to have them over-partition their input, similar to GRACE by Nakayama, Kitsuregawa and Takagi [NKT88]. Some hueristic could be used to estimate how much memory a join would eventually be allowed to use. Based on this estimate, some number of partitions in excess of the amount that would be correct for a static hash join would be created initially. Twice as many partitions as the static version might be a reasonable choice. The join would then proceed much as a traditional hash join, but trying to hold as many partitions as possible in memory instead of immediately spooling them to disk. If the join eventually received more memory than originally expected, it could hold more partitions. Conversely, if it got less memory than expected, more partitions would end up on disk. If memory were squeezed very severely, it might be necessary to start combining partitions.

Zeller and Gray have implemented an adaptive hash join [ZG90]. Our proposal differs from theirs in that our memory broker would explicitly track the total amount

of memory in use by all queries currently running in the system. In contrast, their implementation had no central agent corresponding to our memory broker. Instead, each join monitored the page fault rate and reduced its memory consumption if this rate rose above a threshold. We believe that our approach would prevent page faulting.

1.4 Thesis Overview

The remainder of this thesis is organized as follows: Chapter 2 summarizes related work by other authors. Each of their ideas is compared and contrasted against BAS. Chapter 3 presents in detail the main algorithms used in BAS, along with the goals and assumptions on which these algorithms are based. In Chapter 4 we explain the simulator used to investigate the behavior of BAS. This chapter also contains a detailed description of the workloads used in our investigations. Chapter 5 contains the results of our investigation. Results are analyzed as they are presented. Where possible, we compare our results to those of other investigators. In Chapter 6 we present our conclusions and summary.

Chapter 2

Related Work

In this chapter we present brief summaries of related work by other authors. After each summary we compare and contrast our work with theirs.

To provide a framework for comparing our work to that of other authors, we categorize ideas using two properties. First, whether an idea makes use of adaptive algorithms. Second, whether it uses a central broker. In Table 2.1 we show where we believe the ideas discussed in this chapter fall.

	Not Adaptive	*Adaptive*
No Central Broker	Global LRU DBMIN [CD85]	Adaptive Hash Join [ZG90] Partially Preemptible Hash Join [PCL93b] Memory-Adaptive External Sort [PCL93a]
Central Broker	MG-x-y [NFS91] ROC [YC93]	BAS

Table 2.1: Classification of Related Work

2.1 DBMIN - Memory Allocation

One of the earliest investigations of memory management that directly addressed issues of partitioning memory among concurrent queries was done by Chou and DeWitt

[CD85]. They presented an algorithm, called DBMIN, that showed the advantage of giving each query a separate share of the buffer space, as opposed to depending entirely on a global pool of buffer pages. DBMIN was largely concerned with replacement strategies for different types of queries to use within their private buffer pools. With respect to the allocation of memory to queries, DBMIN in most cases gave each query a minimum amount of buffer space and left the rest in a global pool. If there was insufficient memory available to allocate the recommended minimum, DBMIN would queue the query until more memory became available. Later work by Ng, Faloutsos, and Sellis [NFS91] investigated the application of marginal gains to decide when it would be advantageous to start queries with less than their recommended minimum allocation of memory.

DBMIN differs from BAS in at least two important ways:

- DBMIN did not use adaptive algorithms. Queries were given a share of the buffer space that was calculated based on the situation at the moment that they were admitted to the system. DBMIN did include a "global buffer pool". This global pool contained all the buffers which were not currently assigned to a particular query. Under some circumstances queries could end up using buffers from the global pool in addition to ones that they owned, but there was no provision to assign these buffers intelligently. In contrast, the sorting algorithm in BAS changes its behavior when its memory allocation changes.

- Queries in DBMIN had no knowledge of how many buffer pages were allocated to them. In BAS, queries not only know exactly how much memory they have, but continuously adjust their execution strategy to make the best use of what they have.

2.2 MG-x-y - Marginal Gains

Ng, Faloutsos and Sellis proposed a family of algorithms they called MG-x-y [NFS91]. These were direct extensions of DBMIN by Chou and DeWitt [CD85]. MG-x-y allocated

memory to concurrent queries based on marginal gains. It also allowed queries to proceed with sub-optimal allocations of memory, overcoming a deficiency in DBMIN.

Our work differs from MG-x-y by allowing a query to adapt by using different amounts of memory during its lifetime based on how much is available. Because MG-x-y allocates memory to jobs once, when they start, it runs the risk of giving away all of memory to a few jobs because it cannot know that more are about to be submitted. At the same time, MG-x-y must be careful not to allocate memory too cautiously, else jobs end up starved for memory. To deal with this problem, the authors introduced two parameters for tuning their algorithm (hence the "x" and "y" in "MG-x-y") and investigated methods of setting these parameters optimally. BAS, on the other hand, is self-tuning for the most part, as we show in Section 5.3.

In a later paper, the authors added "predictive load control" to their MG-x-y algorithm so that it would not need to be parameterized [FNS91]. Predictive load control uses the recent history of the workload to predict the effect of different distributions of buffers among queries waiting for admission to the system. While this eliminates the need to set parameters for MG-x-y, it assumes a workload such that there are normally jobs queued up waiting to enter the system. It is not clear that MG-x-y with predictive load control would do much for a system that was not saturated. Predictive load control also depends on a predictable workload. In contrast, BAS can provide a considerable benefit in lightly loaded systems and in systems with varying workloads, as we show in Chapter 5.

2.3 Return on Consumption

Yu and Cornell use a combination of simulation and analytical techniques to investigate optimal partitionings of buffer space among concurrent queries [YC93]. They argue persuasively that the space-time product of memory consumption should be used in calculating the marginal utility of memory allocations. They apply their ideas to three methods of joins: hash, sort-merge and nested loops.

While the goals of Yu and Cornell's work are similar to those of BAS, their work differs substantially from ours in that they do not consider adaptive algorithms.

2.4 Adaptive Hash Join

Zeller and Gray have implemented an adaptive hash join algorithm as a prototype in Tandem's NonStop SQL [ZG90]. Their algorithm uses an operating system service to monitor the page fault rate of the entire system. When the rate of page faults exceeds some threshold, their hash join algorithm adjusts to use less memory. Conversely, when page faults fall below a threshold, the algorithm adjusts to take advantage of more memory.

While Zeller and Gray showed this to be an improvement over the join algorithm previously implemented in Tandem's NonStop SQL, we see their reliance on monitoring the page fault rate of the entire system as a serious weakness. In their implementation, the adaptive mechanism can only be triggered when a join uses more memory than is physically available. It has been shown that database algorithms perform significantly better when they manage their own memory, rather than relying on the virtual memory manager of the operating system [Ver86, Sto81]. In contrast, since BAS knows exactly how much memory is in use by all queries in the system, it can arrange memory usage such that the sum total of memory used by all queries never exceeds the physical memory available. As a result, BAS will never cause page faults during the execution of a query.

To illustrate this difference between BAS and adaptive hash join, imagine that the system is currently running a single query that is large enough to use most of the physical memory on the system. When another large query enters the system, BAS would initially give the new query only as much memory as was currently unused. Soon, the first query would check in, and the memory broker could reduce its memory allocation. The memory freed by this new allocation could then be allocated to the second query. At no time would the queries use more memory than was physically available. In contrast, adaptive hash join would deal with such a situation by intentionally giving the second query more

memory than is actually available, depending on the first query to notice that it is now causing page faults and adjust accordingly. Page faulting by queries in a DBMS is an expensive way to get attention.

2.5 Partially Preemptible Hash Joins

Pang, Carey and Livny investigate a family of hash join algorithms designed to adapt to fluctuations in available memory [PCL93b]. Their work uses the simulation tool DeNet to study a model in which a single hash join is subjected to random fluctuations in the amount of memory it is allowed to use. Our work investigates issues of memory allocation in the case of multiple adaptive queries running concurrently.

2.6 Memory-Adaptive External Sorting

Pang, Carey and Livny have investigated a family of external sorting algorithms designed to adapt to fluctuations in available memory [PCL93a]. This paper appears to be a companion paper to [PCL93b]. It also uses the simulation tool DeNet to study a similar model of a single external sort adapting to random fluctuations in available memory. Our work investigates methods of allocating memory to multiple concurrent adaptive external sorts with the goal of minimizing the average response time.

Chapter 3

Algorithms Used

In this chapter we give the details of the key algorithms used in this thesis. In Section 3.1 we list our assumptions. Section 3.2 discusses the goals these algorithms are intended to achieve and the measure used to quantify their behavior. Section 3.3.1 gives a little background on how existing external sorts work, then 3.3.2 and 3.3.3 show what modifications we made to make them adaptive. In Section 3.4 we discuss two alternative algorithms for the memory broker. In Section 3.5 we discuss briefly why we chose to design BAS using check points rather than interrupts. The results of these two different algorithms will be compared in Chapter 5.

There are two principal entities investigated in this work: our version of an adaptive sorting algorithm and the memory broker. We refer to the combination of these two algorithms as BAS (Brokered Adaptive Sort). Since we investigated two possible algorithms for the memory broker portion of BAS, we will sometimes distinguish between the variants as "the equal allocation variant of BAS" and "the marginal gains variant of BAS". Both variants make use of the same sorting algorithm.

3.1 Assumptions

We make the following assumptions and simplifications:

- We investigate only the behavior of sorts. While our ideas are applicable to joins, we leave that topic for future research.

- We model sorts whose input starts and ends on magnetic disk. That is, none of them read their input from a pipe or another process nor do any of them write their output directly to another process.

- We assume that sorts know exactly how much data they are sorting. This allows them to calculate the potential value of a proposed memory allocation. In contrast, it would be difficult to apply the more sophisticated of our two memory broker algorithms (marginal gains) to a sort that was required to be able to handle an unknown amount of data emerging from a pipe.

- We model a DBMS running on a single CPU system with multiple disks. We assume that there are sufficient disks that some reasonable number of sorts can run concurrently without resulting in significant disk contention. During our experiments, the maximum number of concurrent sorts was arbitrarily set to four.

- We assume that the database system as a whole has a fixed amount of physical memory assigned to it permanently and furthermore that it is aware of this amount.

- We assume a buffer manager for the DBMS where queries use pointers to records maintained in a global buffer. In this scheme, operators "pin" and "unpin" buffers as they use them. This model is inspired by Volcano [Gra90a]. The important point is that queries need to have some control over how much memory is available to them.

- We ignore issues of data sharing. That is, our simulation does not model multiple queries reading from the same base input.

- In our formulas, we ignore certain small uses of memory. In particular, we ignore any memory needed for pointer arrays in sorting and the single output buffer needed for assembling blocks of sorted or merged data to be written back to disk.

3.2 Goals

The goal of BAS is to improve the performance of the DBMS. Traditionally, there are two main measures of the performance of a DBMS: throughput and response time. We have chosen to use average response time as our measure of the performance of the system.

Our primary reason for choosing response time as our measure of system performance is that we feel it better reflects the concerns of current users of interactive database systems. Throughput is of more concern in older style batch-oriented systems. Another reason for choosing response time over throughput is that it is easier to measure directly.

We define response time as the time from when a job is submitted until when it completes. This time includes time that the job spends waiting in a queue to be admitted to the system. Some sources define response time as the time from when a job is submitted until the user first sees output. While this definition is useful in other contexts, it raises issues such as whether improvements in response time are legitimate, or simply reflect jobs using more resources early, and then deferring work once they have reached the point of presenting output. Our definition gives a more honest measure of how quickly work is being accomplished in the system.

By using a straight average of the response time of all jobs in the workload as our measure, we will naturally give more weight to improvements in the response times of larger jobs. This bias is desirable, since jobs that are small enough to finish quickly regardless of the memory allocation scheme used are of less concern anyway. As an example, consider a workload that consists of two jobs, one of which takes about 10 seconds and another which takes about 10 minutes. Our measure would give preference to a system that completed the two jobs in 15 seconds and 9 minutes, respectively, over one that completed them in 5 seconds and 11 minutes. Thus, the two minute improvement in response time for the large job outweighs the 10 second penalty for the small job because it is a larger quantity, even though the difference in response time for the smaller job is proportionally smaller.

3.3 Algorithm for External Sort

We begin our presentation of the external sort algorithm used in BAS by reviewing how external sorts are normally done. Then we present the differences between ordinary external sorts and the adaptive external sort algorithm we use in BAS.

3.3.1 Overview of Classic External Sort Algorithm

Sorts are normally categorized as internal or external depending on whether all of the data to be sorted can fit in memory (internal) or not (external). There are many algorithms for external sorting [Knu73]. We have picked what we believe to be a common, robust, and efficient version of external sorting as the basis for BAS.

The algorithm that we have chosen to for our external sort is composed of two stages: run creation and merging. These stages are illustrated schematically in Figure 3.1. During run creation, as much data as will fit is read into memory. An array of pointers is then created where each pointer points to one record in the input data. The pointers are then sorted using an efficient internal sort algorithm — presumably quicksort. The use of pointers accomplishes two purposes. First, it is much faster to exchange pointers in memory than to copy entire records. Second, it makes it straightforward to sort data containing variable-sized records.

After the data has been sorted, it is written in sorted order to a temporary file on disk called a run. Since consecutive records are not contiguous in memory, they must be copied to an output buffer to assemble blocks of contiguous sorted data.

This process of run creation is repeated until all of the data resides in sorted runs on disk. For sorts below a certain size, there is a small refinement possible, wherein a portion of the data is kept in memory to reduce the amount of IO. This refinement is explained in detail in Section 3.3.3.1 below.

Following the nomenclature of Knuth, we refer to these initial runs as "level-0" runs. After all of the input has been sorted into level-0 runs, the algorithm begins its first merge phase. If there is sufficient memory for all of the level-0 runs to be merged into

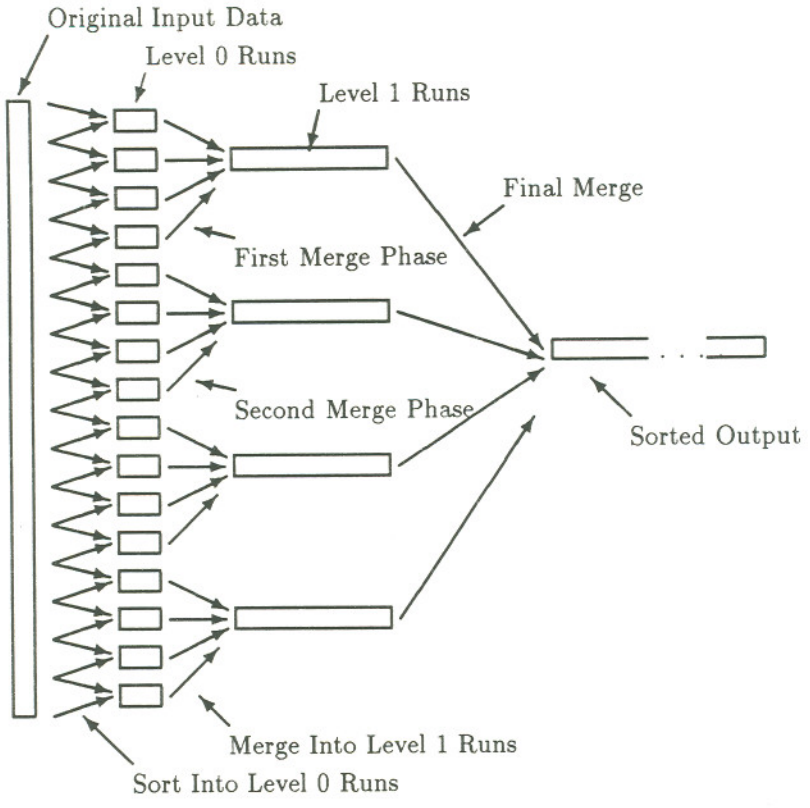


Figure 3.1: Schematic Diagram of Classic External Sorting

the final output, then there will be only one merge phase. Otherwise, the sort merges as many of the existing sorted runs as possible into a larger sorted run. This process is repeated until the number of runs is reduced to a small enough number that they can all be merged into the final output. We call each round of merging smaller runs into a larger one a "merge phase".

In this thesis, we are primarily concerned with sorts where the data is too large to merge in a single pass. The number of runs that *can* be merged at a time is referred to as the "fan in" of the merge. The fan in depends partly on the amount of memory available, but mostly on the characteristics of the IO subsystem. IO to disks is typically more

efficient for larger block sizes [Gra90b]. Hence the best performance is not necessarily obtained by maximizing the fan in. Often, better performance is obtained by using larger blocks for more efficient IO, even at the expense of a smaller fan in and hence extra passes through the data. Since finding the ideal block size is not part of this investigation, we measure memory in terms of the number of blocks, rather than kilobytes or megabytes.

3.3.2 Overview of External Sort Algorithm in BAS

The external sort algorithm for BAS differs from the classic external sort algorithm in that it can use different amounts of memory at different points in its lifetime, in order to adjust to changes in the amount of memory available. Our goal is to use as much memory as is available in order to reduce the number of IO operations necessary to complete the sort, without using so much that paging occurs. The result, as far as the sort algorithm is concerned, is that some runs are larger than others and some merge phases involve more runs than others. See Figure 3.2 for a schematic of what might result.

The way sorts adjust their memory allocation is to "check in" with the memory broker at points when they could change their memory allocation without too much cost. These points are just before starting to create a new run (except for the final run) and just before beginning a merge phase (except for the final merge phase). These restrictions on not changing memory just before the last run or merge are discussed in Sections 3.3.3.1 and 3.3.3.2.

Conceptually, checking in consists of sending a message to the memory broker and waiting for a reply. We discuss possible implementations of this process in Section 3.5. At a minimum, the message sent to the memory broker by the sort needs to contain the following information:

1. The amount of memory currently allocated to the sort.

2. The maximum amount of memory that the sort could use.

Depending on the sophistication of the memory broker, it may need additional information on which to base its memory allocation:

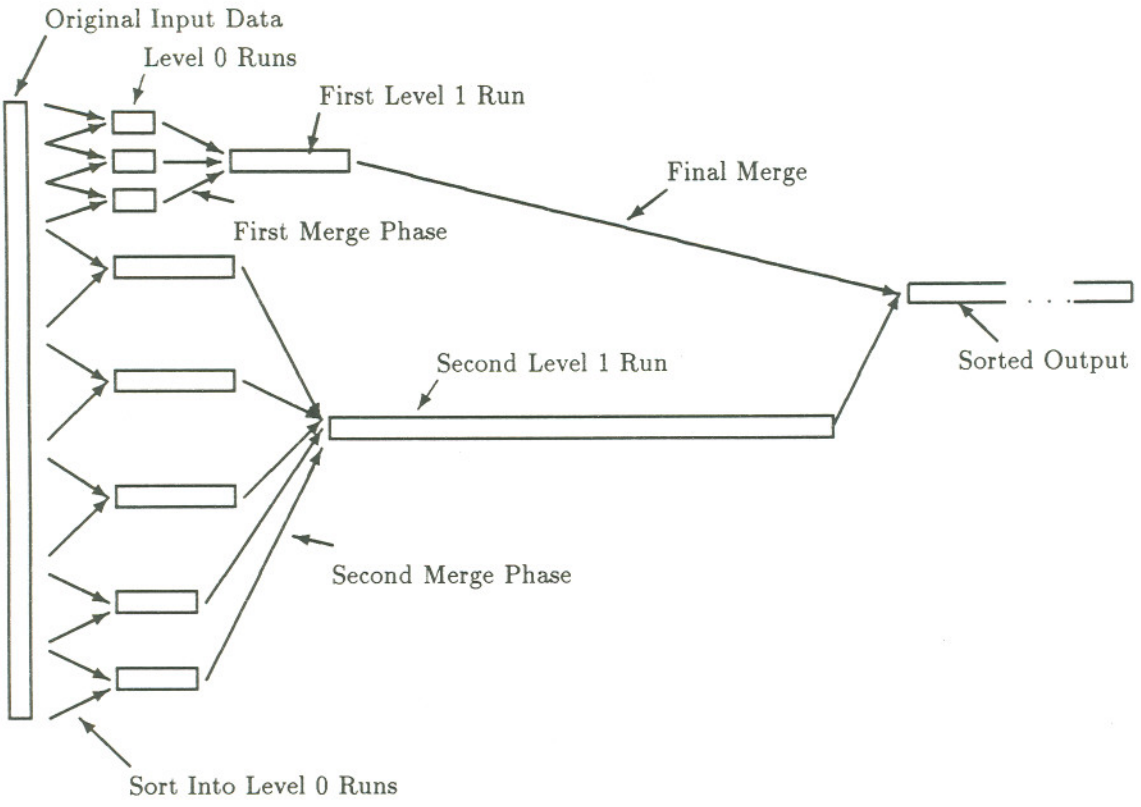


Figure 3.2: Schematic Diagram of Adaptive External Sorting

3. Amount of data remaining to be sorted.

4. Number of runs on disk.

Because we assume that sorts manage their own buffer space, the memory broker need do nothing more than inform the sort how much memory it is allowed to use after each checkpoint. There is no need for the memory broker to actually allocate any memory in the sense of manipulating page tables or managing blocks of free space.

An interesting consequence of the fact that checkpoints only occur at points when the sort is about to create a run or begin a merge phase, is that the more memory a sort

is allocated the longer it is until it checks in again. The reason that this happens is that the size of the run created (or data merged during a merge phase) is proportion to the size of the memory used. Since more data takes more time to sort or merge, it is longer until the sort is finished with that step and ready to check in again.

When we first designed BAS, we were worried that this phenomenon would cause sorts with large allocations to "hog" memory. To avoid this problem, we created tunable parameters for the adaptive algorithms to cap the amount of memory allocated to a single sort (see Sections 3.4.1 and 3.4.2). Fortunately, this turned out not to be a problem, and the tuning parameters are largely unnecessary (see Section 5.3).

3.3.3 Refinements

Our algorithm contains two refinements of the main ideas presented above. The first refinement has to do with planning ahead during run creation so that blocks of data can be held in memory rather than written to temporary files. The second also involves planning ahead, but during the merge phases to avoid unnecessary passes through the data.

3.3.3.1 Refinements During Run Creation

If the sort is given the use of enough memory during its lifetime that it can merge all runs in a single merge phase, then it is possible to arrange the sizes of the last few runs such that the final run generated can be retained in memory. Every block of data retained in memory in this manner saves two IO operations; one to write the block to a run, and one to read it back in during merging.

During the run creation phase of the sort, the sort initially makes each run as large as its memory allocation. However, the memory allocations during the course of run creation are unlikely to be such that the last run exactly fills memory. Instead, the last run created is almost certainly going to be smaller than memory. Our refinement is to make the penultimate run smaller than it would otherwise have been so that the maximum amount of input data is deferred to the final "in memory" run. This strategy

is complicated by the fact that the final run must be somewhat smaller to leave room for input buffers, used to read runs which were written to disk.

The following examples show what could happen if a sort proceeds first without and then with the use of this refinement. The case of sorting without the run-size refinement is presented first. This scenario is illustrated in Figure 3.3.

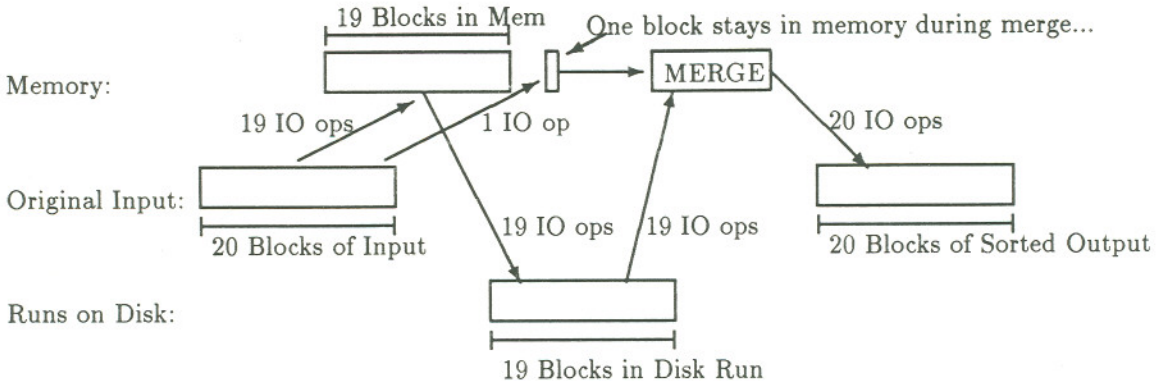


Figure 3.3: Sorting Without Run Size Refinement

1. Sort of 20 blocks of data submitted to system.
2. On first checking in, the memory broker allocates 19 blocks of memory to the sort.
3. The sort naively sorts 19 blocks of data and writes it out to a single disk run.
4. The sort checks in again. Nothing much has changed and the sort could still get 19 blocks of memory. It can't use that much now, so instead it gets 3 blocks: one to read and sort the remaining data, one to read the disk run for merging and one to assemble sorted blocks from the in-memory run.
5. Now the sort reads and sorts the remaining block.
6. Finally, the 19 blocks in the disk run are read and merged with the one block of sorted data held in memory.

The total I/O for this scenario is:

1. 19+19 to read, sort, write first run
2. 1 to read and sort final block
3. 19 to read disk run for merging
4. 20 to write final output

for a total of 78 I/O's. Contrast this result with what happens when the sort plans ahead a little bit:

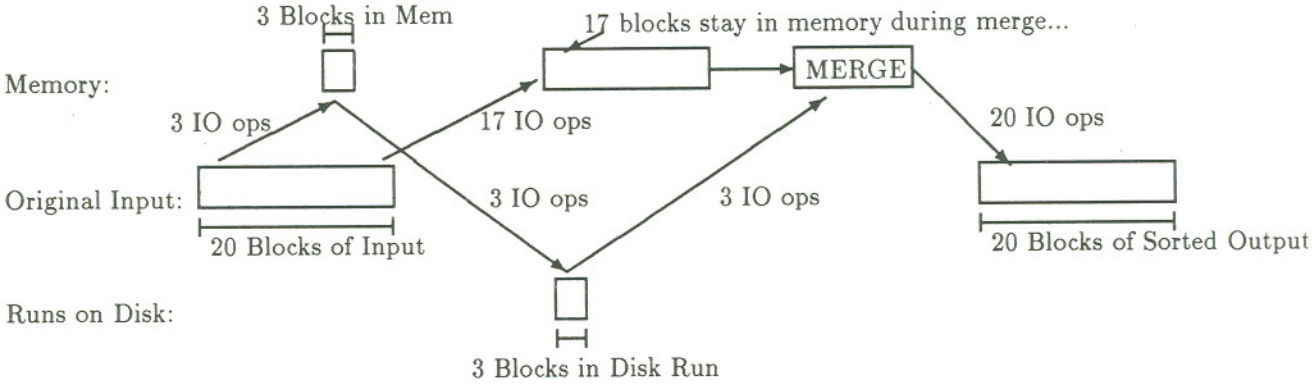


Figure 3.4: Sorting With Run Size Refinement

1. Sort of 20 blocks of data submitted to system.
2. On first checking in, the memory broker allocates 19 blocks of memory to the sort.
3. The sort sorts only 3 blocks of data and writes it out to a single disk run.
4. Now, even though the sort is at a point where it could check in, it doesn't. The reason is that it couldn't use any more memory than it already has, and any less would lead to a performance disaster similar to the one outlined above.

5. Sort reads in remaining 17 blocks of input data and *holds them in memory.*

6. Finally, the 3 blocks of sorted data on disk are read and merged with the 17 blocks of sorted data held in memory.

The total I/O for this is

1. 3+3 to read, sort, write first run

2. 17 to read and sort final (in-memory) run

3. 3 to read disk run for merging

4. 20 to write final output

For a total of 46 I/O's. This strategy works whenever the sort can get a large enough memory allocation while still executing in the run creation phase. "Large enough" means that the remaining data can be sorted in two runs, while still leaving space for the final in-memory run to coexist with the input buffers for the merge.

3.3.3.2 Refinements During Merge Phases

Knuth presents numerous refinements to merging in external sorts [Knu73]. Most of them involve adjusting the number of runs merged in early merge phases so as to reduce the total IO of the sort. None of these can be applied directly by BAS because they all depend in one way or another on planning ahead. Since the amount of memory allocated to the sort may change from one merge phase to the next in BAS, it is not possible to plan ahead in this manner. At the same time, the performance penalty for merging naively can be so great that it is worthwhile to do some optimization. Our compromise is to do this type of planning ahead only when the number of runs has been reduced to a value that the sort could merge in only two phases using the current memory allocation.

This decision to optimize the final two merge phases is a compromise. If BAS were to optimize all of the merge phases, then it would not be possible for it to adapt after it finished run creation. On the other hand, doing no merge optimization at all can greatly

increase the number of IO operations necessary to complete the sort. Optimizing only the last two merge phases gives a significant performance improvement while sacrificing very little of BAS's adaptive nature.

3.4 Algorithms for the Memory Broker

We implemented and compared two adaptive algorithms for the memory broker:

1. Equal Allocation. This algorithm attempts to equalize the memory allocation of all queries currently active in the system. It is intended to be a very simple memory allocation strategy that takes advantage of our adaptive external sort algorithm.

2. Marginal Gains. This algorithm takes into account the needs of each query when allocating memory.

For comparison purposes we also implemented a third, non-adaptive, algorithm that we call static. It is intended to be an approximation of how memory allocation has traditionally been done in many commercial database systems [Ger93]. Static gives each query some fixed amount of memory, regardless of the current workload. Not knowing a priori what fixed amount would be a fair comparison, we made this a tunable parameter P_{st}. In the simulator, P_{st} is the largest fraction of the total memory that will be allocated to a query. (Unless the query is so small that it cannot use that much memory — even under the static scheme we do not give a query more memory than it can use.) If a new query is submitted to the system and all memory is already in use, the new query is queued until memory becomes available for it.

The memory broker keeps track of how much memory is currently available in the system. Queries send messages to the memory broker requesting permission to use more memory or announcing that they have finished using memory. Queries block waiting acknowledgment of their requests. This gives the memory broker the option of holding a request until the memory necessary to satisfy it becomes available.

In our model of the computer system on which our algorithms run, we assume that there is no disk contention as long as the number of concurrent sorts is limited to some

reasonable value. To ensure that the memory broker does not launch more sorts than this limit, we introduced a parameter called "load control". Load control is an integer specifying the maximum number of concurrent jobs allowed. When the number of concurrent jobs reaches the load control limit, further jobs are queued until a running job finishes and exits the system. All variants of the memory broker observe the limit specified by the load control parameter.

Each of the adaptive algorithms for the memory broker is discussed in more detail below.

3.4.1 Equal Allocation

In the equal allocation variant of BAS, we model a memory broker that attempts to allocate equal shares of memory to all active queries. Furthermore, it attempts to keep as much of the memory in use as possible.

To achieve these two goals, each query should receive an allocation of M/n, where M is the total memory in the system and n is the number of queries currently running. However, simply passing out M/n blocks of memory every time a query checks in won't work because the first large query will grab all of memory and hold it, locking out any new queries. Four modifications to this simple strategy are necessary to make it work correctly:

1. The value n must include queries that have been queued for lack of memory (up to a maximum of the load control limit). For example: imagine that there are currently two queries running, each using half of the memory. When a third query enters the system, it will be queued since there is not memory available. Now, the two queries which are running will eventually check in. When they do, they will each get one third of memory, rather than one half as they had before. This new allocation frees up memory that the queued job can use.

2. Queries should be started as soon as there is enough memory available for them to make progress. When there are several queries running and a new one wants to

enter the system, it can take a significant amount of time for the memory broker to free up its full share of memory. Instead, the query should be given as much memory as is available as soon as there is enough to guarantee that it can make progress.

3. There needs to be a limit on how much memory can be allocated to any one query. The reason is that the adaptive sort algorithm used by BAS does not check in after it starts the final merge. If one query were given all of the memory in the system, it could end up preventing new queries from entering the system for a considerable length of time during its final merge.

   To prevent these sorts of problems, the memory broker uses a tunable parameter P_{eq} where $0 < P_{eq} \leq 1$. P_{eq} is the largest fraction of the total memory which a single sort can ever be allocated. For the main experiments presented in this thesis, P_{eq} was set to 0.5. In Section 5.3 we show that BAS performs well over a wide range of values for this parameter.

4. A query should not be allocated more memory than it can use effectively. Since most of the sorts in the workloads investigated in our experiments are much larger than the total memory, this issue rarely comes into play. Regardless, the memory broker does make sure that the memory allocation to a sort does not exceed the maximum that it could use.

The algorithm can be summarized as follows: Each time a query requests memory it is given the smallest of:

- M/n blocks of memory (where n includes queries queued for lack of memory, but does not exceed the limit set by load control).
- All of the remaining free memory in the system.
- $M \times P_{eq}$ (the maximum memory allowed to a single query).
- The maximum memory that the query could use.

Not that for the equal allocation variant of BAS, the operation of the memory broker does not depend on the size of the sort. Hence, our assumption that sorts know how much data they are sorting (see Section 3.1) is irrelevant for this variant.

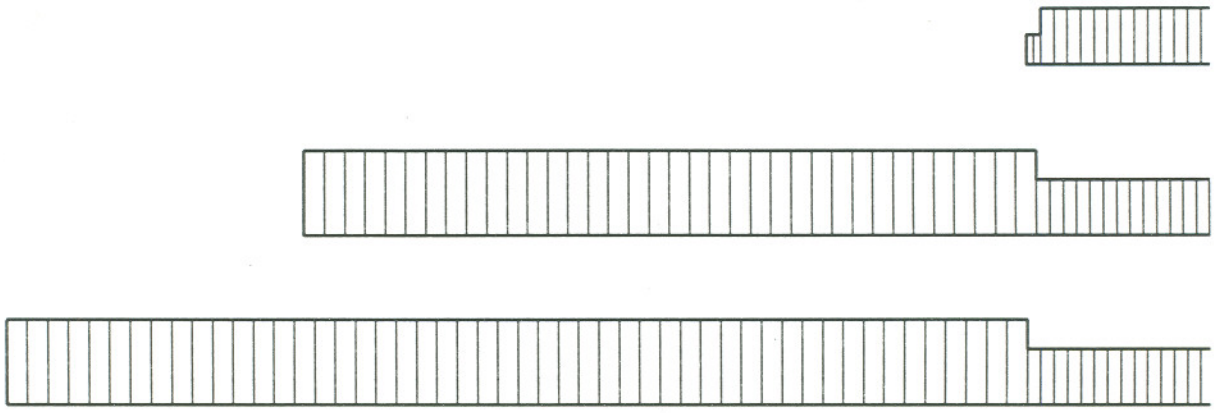


Figure 3.5: Space-Time Diagram for Equal Allocation

Figure 3.5 shows a space-time diagram of this algorithm in action. The vertical bars inside the space-time blocks represent checkpoints.

The action shown in Figure 3.5 is taken from the beginning of a typical run of our simulator. Characteristics of the workloads we investigated using our simulator are presented in more detail in Section 4.3. For this example, P_{eq} has been set to 0.5 and M, the total memory devoted to input buffers in the system, is 64 blocks. The first sort submitted to the system is to sort 2337 blocks. Since this sort is much larger than the total memory in the system, it is allocated the maximum allowed for a single sort: $M \times P_{eq}$, or half the memory in this case. A second sort of 1537 blocks is submitted to the system some time later. Now that there are two sorts in the system, each should receive

$M/2$ blocks of memory. The first sort already has this much, and since an equal amount is currently available the second sort immediately receives is full allocation and begins executing. Toward the end of the diagram, a third sort is submitted to the system. At this point all of the memory is in use, so sort #3 is queued until one of the active sorts checks in. The next sort to check in is sort #1. Sort #1 is now allocated $M/3$ blocks of memory. Sort #3 can now begin executing, but it does not receive its full allocation because only $M/6$ blocks of memory are available ($M/2 - M/3$ just freed by sort #1). Soon sort #2 checks in and is also allocated $M/3$ blocks of memory. This new allocation frees just enough memory that when sort #3 checks in it can now be allocated its full $M/3$ blocks of memory. As desired, the memory broker has quickly converged to an allocation of M/n blocks of memory for each of the n sorts active in the system.

Notice that this variant of the memory broker is so simple that it would not even need to be implemented as a separate process. Since all the queries using the memory broker are part of the DBMS and hence trusted entities, the memory broker could be as simple as a couple of shared variables and semaphores to protect them. The small amount of computation necessary on each "call" to the memory broker could be implemented in the query itself.

In spite of its simplicity, the equal allocation variant of BAS turned out to perform surprising well. Full results of its performance are reported in Chapter 5.

3.4.2 Marginal Gains

Previous work in memory allocation in a multi-query environment has shown the value of using marginal gains [NFS91, YC93, FNS91]. Generally, marginal gains is taken to mean that queries receive memory based on the marginal gain in performance that will result from the proposed memory allocation. While marginal gains works well with static algorithms, we have found that a straight application of marginal gains to adaptive algorithms actually yields worse performance than our equal allocation memory broker algorithm.

In Section 3.4.2.1 we present our implementation of marginal gains. Section 3.4.2.2

analyzes why it fails to perform better than our implementation of equal allocation.

3.4.2.1 Our Implementation of Marginal Gains

There are two aspects of marginal gains that need to be dealt with in order to implement it. The first is the calculation of the value of each additional block of memory to a query. Following the lead of Faloutsos, Sellis and Ng in their implementations of MG-x-y and its variants [FNS91], we make the value of a block of memory equal to the expected change in the number of IO operations that a query could expect from one additional page of memory. The detailed derivation of the formula used to calculate this value is presented in Appendix B. The second aspect of implementing marginal gains is how to apply this value in distributing memory among competing queries. When memory becomes available, it should be given to those queries to which it is most valuable. The tricky part is that as memory is allocated to a query, its need for more memory decreases until eventually it is no longer the "neediest" query, and the rest of the memory should be given to other queries that now need it more. Here, the adaptive nature of BAS required a more sophisticated algorithm than that used in MG-x-y.

The difficulty is that in MG-x-y, memory is distributed only among queries waiting to enter the system. This is acceptable in MG-x-y because queries receive a fixed allocation when they start. In contrast, the whole point of BAS is to allocate memory to queries multiple times during their life. The equivalent in BAS to the memory distribution algorithm of MG-x-y would be to make a query wait each time it checked in for a couple more queries to also check in before any of them could receive their new memory allocation. Given the frequency with which queries check in under BAS, this strategy would lead to queries spending an unacceptable amount of time waiting.

To achieve the goal of distributing memory to those queries to which it is most valuable, we made the memory broker maintain a reserve of memory to pass out to needy queries. The intent is that if a query checks in and is found to need memory worse than the average query currently active in the system, it can be allocated some from the memory broker's reserve. Conversely, if a query checks in and needs memory less than

the average query, some of its memory should be taken away and saved in the reserve for queries that need it worse.

We implemented this strategy by making each query submits a "bid" every time it checks in with the memory broker. This bid is the number of IO operations that the query estimates it could save if it were allocated one more block of memory. The bid is calculated using the formula derived in Appendix B. The memory broker maintains an average of the most recent bids of all queries currently active in the system. It then buys memory from queries that are bidding below average and sells memory to those that are bidding above the current average.

The exact steps taken by the marginal gain version of the memory broker when a query checks in are as follows:

1. Recalculate the average bid not including the bid of the current query. It makes no sense for the query to bid against itself. If there are no other queries in the system, set the "average" to zero. This effectively allows the current query to buy as much memory as it can use, up to the maximum allowed for a single query.

2. Conceptually, all of the memory in the memory broker's reserve and all but two blocks of the memory currently allocated to the query are temporarily placed in a pool of memory to be auctioned to the highest bidder. For each block of memory in this pool, the memory broker allocates it to the highest bidder. The bidders are the memory broker, the current query and possibly a query that has been queued for lack of memory. A query in the queue can participate in the bidding if there is enough memory in the pool to give it two blocks to start bidding with. Recall that sorts make no progress with fewer than two input buffers.

   The bids of the current query (and the query in the queue, if there is one) are calculated using the formula derived in Appendix B. The memory broker bids 0 if its reserve is full, or the average bid of all other queries in the system otherwise. In effect, the memory broker bids on behalf of queries that cannot because they are not at a checkpoint.

This algorithm for partitioning memory between the current query and the memory broker was chosen because it is intuitive and simple. In an actual implementation, a less intuitive but more efficient algorithm that achieves the same effect would be more appropriate than auctioning each block individually.

3. Check that no query has been allocated more than the maximum that it can effectively use. Queries can be allocated more memory than they need because the formula used to calculate the bids is a continuous approximation of the actual marginal gain value. If a query has been allocated more than it can use, return the excess to the memory broker's reserve.

4. After completing the steps outlined above, check that the memory allocated to each query does not exceed the maximum allowed. Recall that no query should be allocated more than $P_{mg} \times M$ blocks of memory. If more than this amount has been allocated, put the excess back in the memory broker's reserve. This event is generally a result of the query being the only one in the system. If this check were not performed, the query would likely buy all of memory, preventing other queries from entering the system. Note that the more memory given to a query, the less often it checks in. Hence the concern about not giving all of memory to a single query.

The formula used by the marginal gains variant of BAS makes use of the size of the sort (B in the formula derived in Appendix B). This is the reason for the assumption in Section 3.1 that the sort is aware of how much data it is sorting. Yet, the algorithm should still work well if it were to use an estimate rather than the exact value for B. There are two reasons to believe this. First, B would be known exactly by the time the sort begins the first merge phase. Since the sorts that we investigate are of sufficient size that they spend more time merging than generating runs, the value of B would be known exactly rather than estimated during most of the life of the sort. Second, as shown by BAS's insensitivity to parameter settings (see Section 5.3), the adaptive nature of the algorithms can smooth over the effect of small mis-allocations of memory. While

we have not performed any experiments to prove it, we believe that this property would also reduce the cost of small errors in estimating the size of sorts.

We present two graphs of the behavior of the marginal gains function. The formula that the simulation uses to calculate marginal gains attempts to estimate the change in number of IO operations necessary to complete the sort if one more block of memory were allocated to the sort for the remainder of its life. Since more memory reduces the number of IO operations necessary, the formula always yields a negative number. These graphs show the formula exactly as it is calculated. In speaking of the formula, we will actually use the more intuitive notion of a "higher" bid when in reality the bid would be more accurately described as "of larger magnitude".

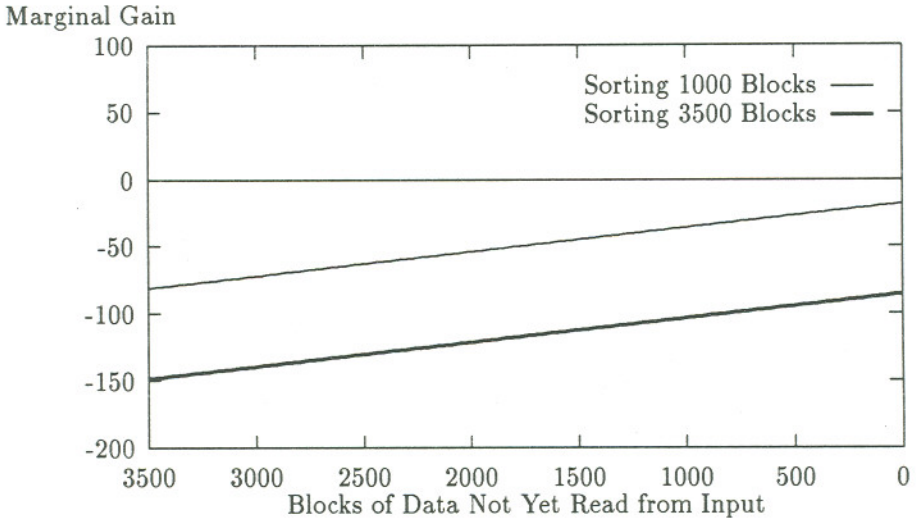


Figure 3.6: Marginal Gains During the Run Creation Phase of a Small and Large Sort

In Figures 3.6 and 3.7 we show how the marginal gain varies during the life of two different sized sorts. The smaller sort sorts 1000 blocks of data, the larger one sorts 3500. Both sorts are assumed to have been working with 32 blocks of data since they

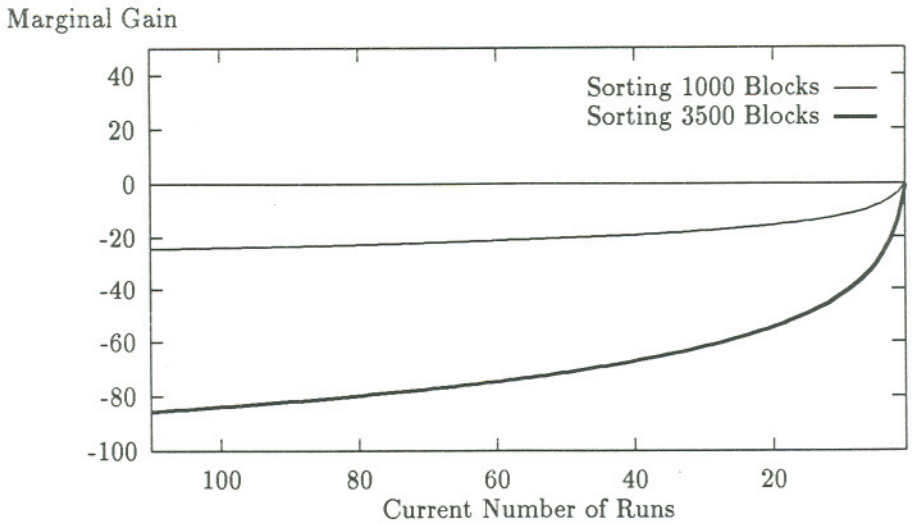


Figure 3.7: Marginal Gains During the Merge Phases of a Small and Large Sort

began executing. These numbers are chosen to match typical values from the workload used in our simulation. As can be seen in the figures, memory becomes progressively less valuable to the sorts as they progress. Also, the larger sort has a greater need of memory than the smaller one. These observations are true in general. Larger sorts avoid more IO operations for a given increase in memory allocation and any sort benefits more from memory allocated earlier.

In Figure 3.8 we show how the marginal gain varies as a function of the amount of memory that a sort has been allocated in the past. For this graph, we again show two sorts of 1000 and 3500 blocks respectively. Each is assumed to be half way through its run creation phase. The graph shows what the marginal gain for each sort is as a function of the amount of memory previously allocated to it. We assume a constant allocation thus far in the life of the sort. Large sorts are more sensitive to memory starvation. By this we mean that a large sort that has a history of small memory allocations will be in

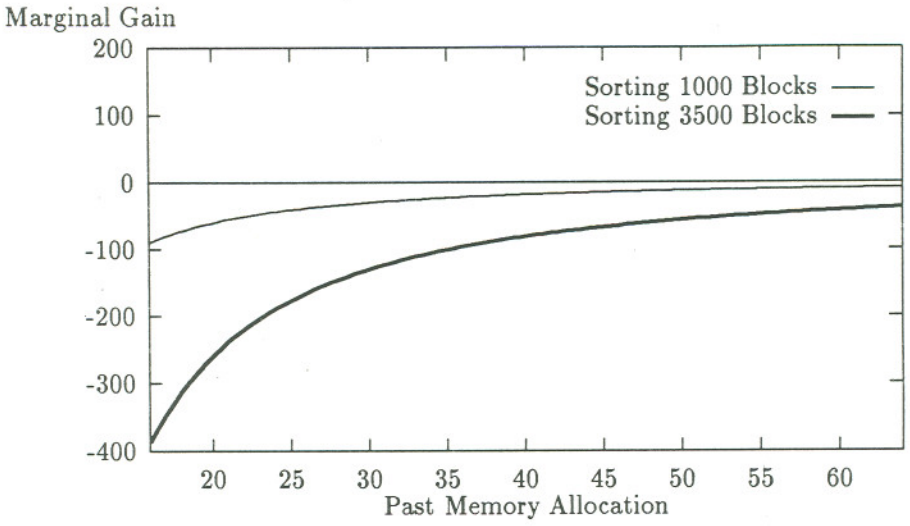


Figure 3.8: Marginal Gains for a Small and a Large Sort with Different Past Memory Allocations

greater need of memory than a small sort in the same situation.

The amount of memory bought by the memory broker is controlled by a tunable parameter P_{mg} where $0 < P_{mg} \leq 1$. This parameter is the minimum fraction of the total memory in the system which the memory broker should ensure is allocated to sorts rather than kept in its reserve. In other words, the memory broker will refrain from buying more than $(1 - P_{mg}) \times M$ blocks of memory no matter how low the price seems to be. P_{mg} is also used to cap the memory allocation of a query when it is the only one running in the system. In this case, the query will never be given more than $P_{mg} \times M$ blocks of memory. This limit is imposed so that there will be some memory free to start new queries.

Figure 3.9 shows the marginal gains algorithm in action. This example shows exactly the same sequence of sorts admitted to the system as the example at the end of the presentation of the equal allocation in Section 3.4.1. As in the previous example, the

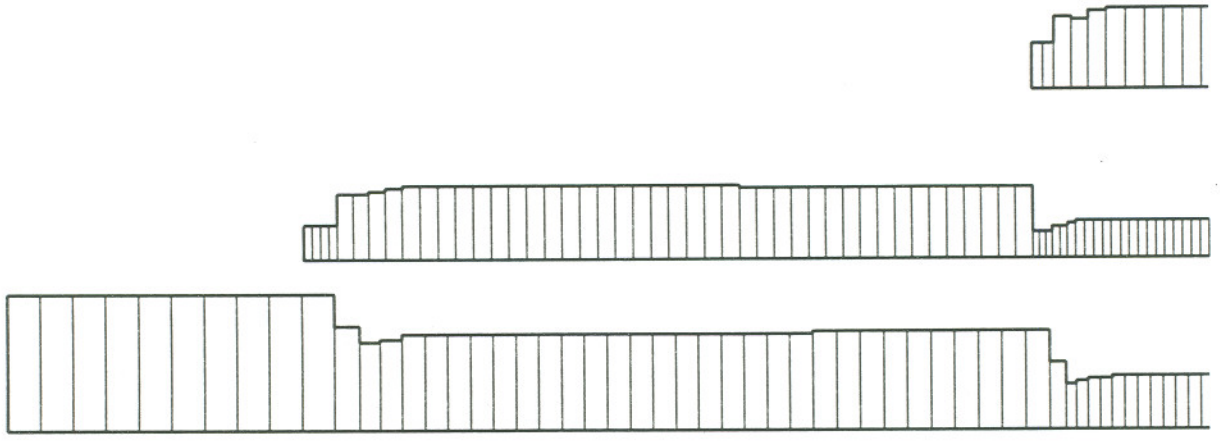


Figure 3.9: Space-Time Diagram for Marginal Gains

vertical bars indicate checkpoints and the total memory devoted to input buffers in the system is 64 blocks. The parameter P_{mg} is set to 0.8 in this example.

The first sort admitted to the system will sort 2337 blocks of data. Since it is the only query in the system, it is allocated $P_{mg} \times M = 51$ blocks of memory. The remaining 13 blocks are kept "on reserve" by the memory broker for starting new queries. When the second sort (sorting 1537 blocks of data) is submitted, it is allocated these 13 blocks of memory and can begin execution immediately.

Now, marginal gains comes into play. During the next two times that sort #1 checks in, the memory broker allocates less memory to it so that it can allocate more to sort #2. Sort #2 is checking in more often, so sometimes there is no new memory to allocate to it, but gradually its memory allocation is increased. At some point, sort #2 has been allocated enough memory that the value of additional memory for it is less than for sort #1. At this point, the memory broker ends up passing out blocks of its

reserve to both sorts until all memory is in use. This process is visible in the space-time diagram for sort #1 as a small upward correction just before the allocations stablize.

When the allocations have stablized, sort #1 has considerably more memory than sort #2 (36 blocks versus 28). This is because sort #1 is sorting more data (2377 blocks versus 1537).

Eventually, a third sort is submitted to the system. As when the second sort was submitted, memory is taken away from the sorts currently active in the system and passed out on the basis of which sort finds the memory most valuable. With three participants in the exchange, there is a little more oscillation, but the allocations quickly converge to a stable distribution where memory is equally valuable to all queries in the system.

3.4.2.2 Challenges in Applying Marginal Gains

In the course of investigating marginal gains as applied to adaptive algorithms, we have come to believe that a straight application of marginal gains is not the right technique for adaptive algorithms. However, there may be other approaches to partitioning memory based on some calculated value of the memory that *would* work. For example, it may be possible to adapt recent work by Yu and Cornell [YC93] that is based on the space-time product of queries and avoid the problems outlined below. In any case, we see two major problems with naive application of marginal gains to adaptive algorithms.

First, in a static scheme, the memory only needs to be partitioned among the queries that are queued waiting to enter the system. Since the queries that are already executing are by definition static, they can neither receive nor give up memory and hence do not need to be taken into account when memory is partitioned. When the queries are adaptive, any query in the system might participate not only with the new query entering the system, but even with other queries already in the system. This adds greatly to the cost of computing how to partition memory simply through the sheer increase in number of possible partitionings.

Second, when the algorithms are adaptive, there is no guarantee of how much a given

query's response time will be reduced by a given change in its memory allocation. This uncertainty results from the fact that the query may change its allocation again in the future. This weakness in marginal gains is more serious than it might seem at first. A naive application of marginal gains to adaptive algorithms will often give very bad answers for common situations. As an illustration, we present an example of a large sort in progress when a small sort enters the system. In Figure 3.10 we show space-time diagrams of various re-allocations of memory for this example.

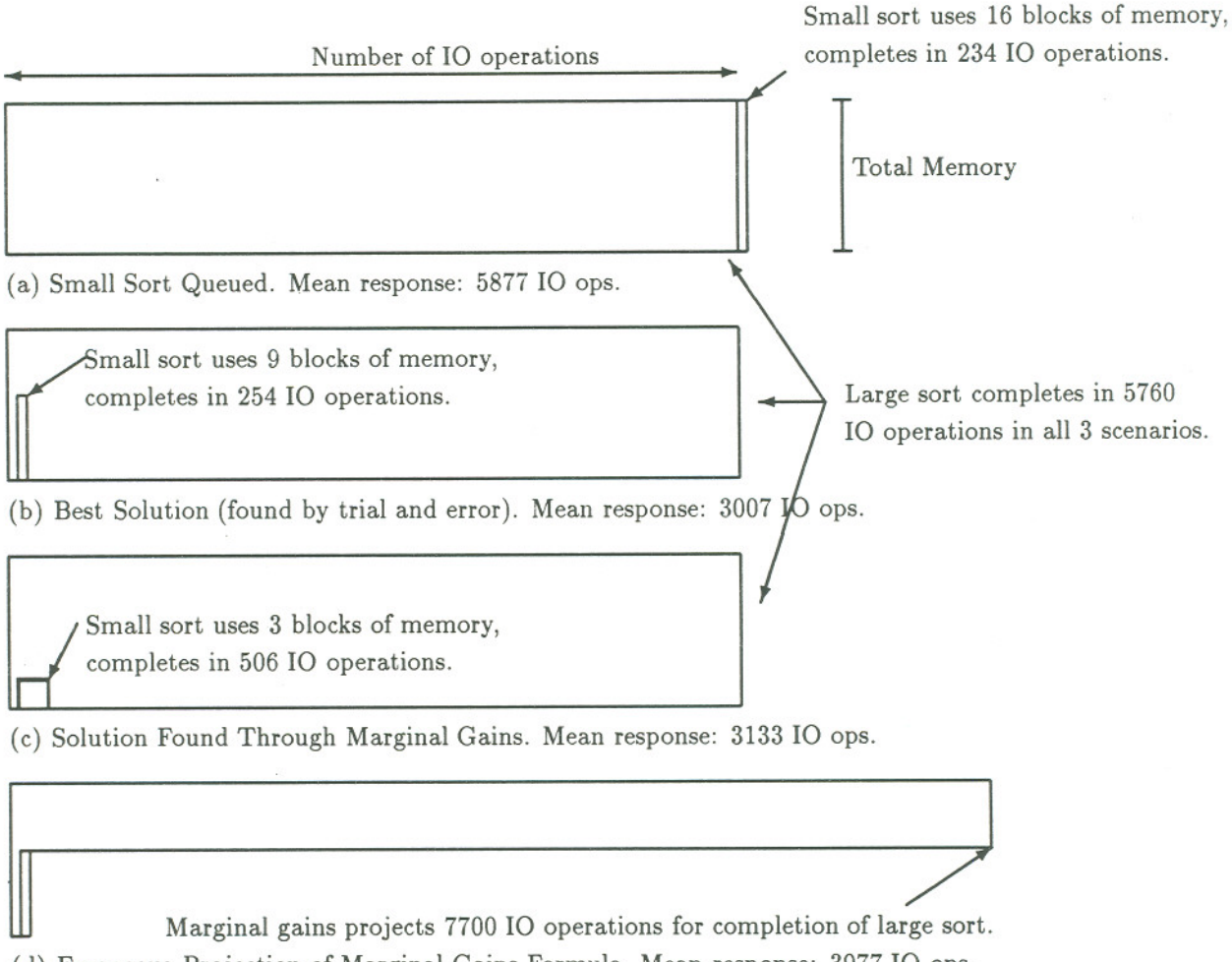


Figure 3.10: Space Time Diagrams for Large Sort in Progress, Small Sort Starting

In this example, the large sort sorts 1024 blocks of data, starting at time zero. The

small sort sorts 64 blocks of data after the large sort has had just enough time to complete 4 runs of data. The total amount of memory available for input buffers is 16 blocks. The results of the different scenarios were obtained by hardwiring BAS to give the jobs the allocations shown in the scenario rather than using one of its regular memory broker algorithms.

As the example starts, the large sort is using all of memory. When the small sort enters the system after 128 IO operations (1.28 seconds) the large sort needs to give up some its memory so that the smaller sort can begin to make progress. The alternative of simply queuing the smaller sort until the large one is done (see Figure 3.10(a)) results in an average response time of 58.77 seconds. It is possible to do much better than this.

The question then is how much memory the large sort should give up. To find out what the optimal solution might be, we simulated what would happen if the large sort gave 2, 3, 4, ... , 14 of its 16 blocks to the small sort. (Remember that external sorts make no progress with fewer than two input buffers.) The best of these solutions is shown in Figure 3.10(b). The average response time of the two sorts here is 30.07 seconds.

This solution is not too surprising, but it is not what would be found through a naive application of marginal gains. Using the marginal gains formula from Appendix B, we would arrive at the solution shown in Figure 3.10(c). This solution has an average response time of 31.33 seconds.

The source of the difficulty is that the marginal gains formula fails to take into account the fact that the memory used by the smaller sort will soon be available again. In effect, the marginal gains formula projects the situation illustrated in Figure 3.10(d), which has an average response time of 39.77 seconds.

This problem really stems from the fact that we cannot predict the future. If the small sort that has just arrived is actually the first of a steady stream of similar sized sorts, then the scenario shown in Figure 3.10(d) may be closer to the truth than our "best solution" shown in Figure 3.10(b).

One of the interesting results of our work is that any reasonable partitioning of the memory in a situation like this yields very similar average response times. One reason

is that the more memory given to the smaller sort, the sooner it finishes and makes the memory available to the large sort again. Another reason is that large sorts have a certain amount of "slack" in their memory requirements. This slack comes from the fact that the memory allocation is unlikely to divide exactly into the size of the data. As a result, there will be a merge phase that does not use all of the input buffers. Interestingly, brief memory shortages during run creation result in more runs of a smaller size, but there are still the same number of passes through the data. This happens because these extra runs will be merged at no extra cost by simply using more of the available input buffers during the merge phase, which would not otherwise have used them all.

3.5 Checkpoints Versus Interrupts

In our model, the interaction between queries and the memory broker occurs at "check points". The idea here is that when the query reaches a point where it can conveniently change its memory usage, it sends a message to the memory broker.

An alternative would be for the memory broker to send interrupts or signals of some sort to a query when it needs to change its memory allocation. Both approaches have merit. Checkpoints are simpler and easier to implement, interrupts would likely result in better performance.

3.5.1 Advantages of Checkpoints

The biggest advantage of using checkpoints is that they are easier to implement. Since queries only check in at times when it would be convenient for them to adjust the size of their working set, the adaptive algorithms are much simpler and more intuitive. Even if you did implement an adaptive algorithm based on interrupts, there are some points in a query's execution when it would be impractical to change the working set size. The result would be that interrupts would often end up behaving like checkpoints anyway.

There are several alternatives for implementing checkpoints:

Messages

Perhaps the best implementation of checkpoints would be with actual messages. The query sends a message to the memory broker when it is time to check in and blocks waiting for the response. The memory broker acts on the message, and sends the query its new maximum working set size via a return message. In this alternative, the memory broker is truly a separate process or thread.

Shared Variables

If the algorithms used by the memory broker are very simple, it may be easier to build the functionality into the query itself. The equal allocation variant of BAS is simple enough that it lends itself especially well to this scheme. The critical data structures for the algorithm would be kept in a shared variable protected by a some sort of mutual exclusion protocol (a semaphore, for example). Each query would perform its own calculation to determine its allowed working set size based on the values of the shared variables. One of the reasons this method would work is that all of the query algorithms are trusted members of the DBMS. There is no possibility of queries intentionally "cheating".

Polling

One aspect of BAS that is apparent from examining traces of our simulator is that the majority of checkpoints do not result in any change to the size of the query's memory allocation. This fact suggests that it may be better for queries to check in only when necessary. Checkpoints implemented with polling would be identical to those implemented with messages, except that queries would only check in if the memory broker had set a flag for them since their last checkpoint. This scheme would avoid the overhead of messages except when there is something that actually needs to be communicated.

3.5.2 Advantages of Interrupts

The advantage of interrupts is that the system can be made to adapt more quickly. For example, if all of the memory in the system is in use and a new query is submitted the memory broker would not need to wait for an active query to check in in order to find memory for the new query. Rather, the memory broker could choose one or more queries and interrupt them immediately. For interrupts to work well, the adaptive algorithms would need to be more sophisticated than those investigated in BAS. Pang, Carey and Livny have explored the algorithms necessary for this approach [PCL93a, PCL93b].

Chapter 4

Simulation

In this chapter we discuss details of how we simulated our ideas. Section 4.1 contains a brief description of the simulation tool that we used. Section 4.2 describes the system modeled by the simulator. The workloads used in our experiments are described in Section 4.3.

4.1 Simulation Tool

We wrote the simulator in the simulation language DeNet [Liv90]. DeNet is a simulation language based on Modula-2. It provides a rich set of functions and data types for managing events, queues, and other components of a discrete event simulation. Constructs built into the language facilitate message passing between modules of the simulation, which correspond to components of the system being modeled.

4.2 Model

We model a simple system consisting of a single CPU connected to a set of disks. Jobs submitted to the system are modeled internally as data structures containing all the fields necessary to completely describe the current state of a job. Our workload is discussed in detail in Section 4.3.

DeNet, the simulation tool we used, encourages a model in which separate entities of the system being modeled are cast as separate modules in the simulator. In keeping with this approach, our simulator consists of the following modules:

- Source
- CPU
- Disks (All disks are grouped in one module)
- Memory Broker
- Sink

The relationships of these modules are illustrated in Figure 4.1. Each of these modules and their interactions are explained below.

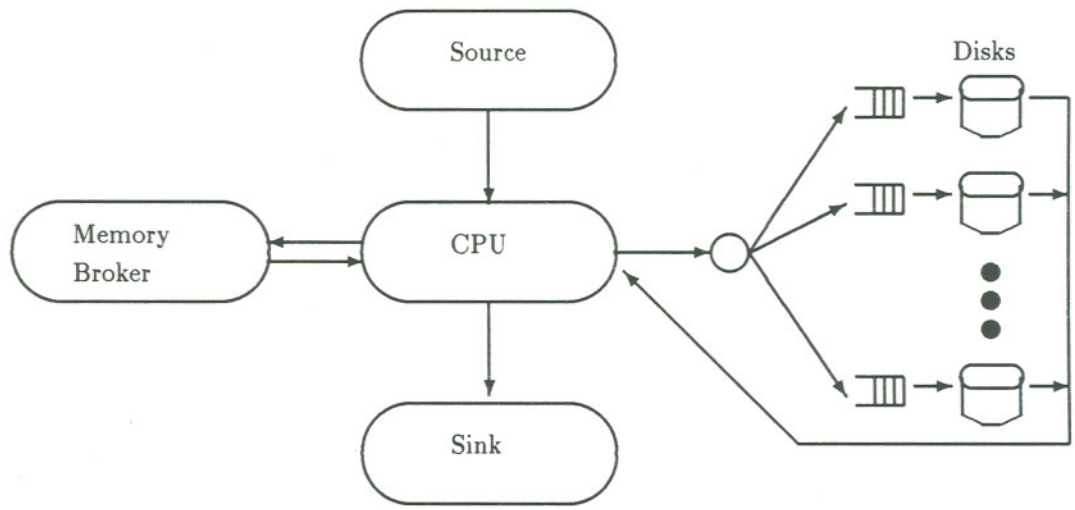


Figure 4.1: Modules of the Simulator

4.2.1 Source Module

The source generates the job stream. It accepts parameters that completely specify the characteristics of the job stream to produce. We give details of the workloads produced by the source in Section 4.3.

4.2.2 CPU Module

The CPU in our cases acts as a dispatcher to route jobs to the service they need next. Since disk accesses are the bottleneck in most DBMSs, we do not model CPU time. Furthermore, the amount of memory devoted to a sort has very little impact on the total CPU time used. In particular, taking memory away from an external sort reduces the amount of CPU time used to sort runs by almost exactly the amount of additional CPU time needed to merge the larger number of runs that result.

4.2.3 Disk Module

Disk accesses are modeled as a constant 10ms delay. We assume that there are sufficient disks that some number of jobs can run concurrently without significant disk contention. For the experiments reported in this thesis, we restricted the maximum number of concurrent jobs to four.

4.2.4 Memory Broker Module

The CPU module dispatches jobs to the memory broker when they need to check in. The memory broker generally makes a memory allocation for the job and returns it to the CPU immediately. In a few situations, the algorithms used by the memory broker require that the job be queued until memory becomes available for it. The algorithms employed by the memory broker were described in Section 3.4. The calculations performed by the memory broker are actually quite simple. Hence, those memory allocations that do not require that the job be queued for some period are modeled as taking zero time.

Because our memory broker ensures that the sum total of the sizes of the working sets of all queries in the system is strictly bounded, there will be no page faulting in BAS[1]. Hence, we do not model memory access.

[1]Depending on how memory is managed in the DBMS, there may be some small number of page faults when a query first starts, or when one query's working set grows at the expense of another.

4.2.5 Sink Module

Completed jobs are sent to the sink. The sink collects statistics and performs cleanup necessary for the internal workings of the simulator.

4.3 Workload

We investigated the behavior of BAS on two different workloads. The first is what we call the "bursty" workload, which consists of short bursts of jobs separated by periods of quiessence. The second workload is a more traditional one, consisting of a steady stream of jobs.

There are two features common to all the experiments reported here. First, in both workloads, each job sorts a random amount of data. The amount of data sorted is distributed exponentially with a mean of 2500 blocks. Second, in all experiments reported here, the simulator simulates 100 jobs from submission to completion.

We measure the size of sort jobs in blocks of data to be sorted. Memory is also measured in units of the same size. Although we refer to this unit as a "block", it is not necessarily a physical block of the disk. Rather it is the preferred amount of data that the DBMS reads or writes in a single call. Graefe (who uses the word "cluster" where we use the word "block") reports that performance for external sorting in Volcano is greatly improved by increasing the block size from 4 KB to 16 KB [Gra90b]. Smaller performance gains are achieved by continuing to increase the block size, until somewhere around 40 KB.

Keeping in mind the discussion above, our average sort size of 2500 blocks corresponds to sorts with an average size of 40 - 100 MB depending on whether you choose your block size to be 16 KB, 40 KB or something in between. The sorts execute in a workspace where 64 blocks (1 MB - 2.5 MB) are devoted to input buffers. We see no reason why our results would not scale to larger sorts in larger amounts of memory. Keep in mind also that this amount of memory is for only enough disks to allow four concurrent jobs. More disks would allow more concurrency and hence require more memory.

The bursty workload is described in more detail in Section 4.3.1. The exact parameters for the steady workload are presented in Section 4.3.2.

4.3.1 Bursty Workload

One of the strengths of BAS is its ability to perform well with workloads where the number of active jobs varies greatly over time. In order to demonstrate this, we investigated what we term a "bursty" workload.

In the bursty workload, jobs arrive in bursts of from one to four jobs. The number of jobs in a burst is uniformly distributed. Within a burst, jobs have a mean interarrival time of one second. This interarrival time is exponentially distributed. Since the average response time for jobs in this workload is measured in hundreds of seconds, the jobs in a burst can be thought of as starting nearly simultaneously. There is also a mean interarrival time between the bursts themselves. For the experiments reported in Chapter 5, we varied this interarrival time from 500 seconds (corresponding to a fairly light workload) down to 75 seconds, at which point the workload was heavy enough to nearly saturate the system.

Bursty workloads could occur naturally in a couple of ways. For example, if the DBMS implemented pipelined multi-way joins using sort-merge, the individual sorts that make up a single join would appear in a sudden burst.

4.3.2 Steady Workload

We also ran BAS against a more traditional workload. In the steady workload, jobs arrive with a mean interarrival time that varies from 120 down to 30 seconds in our experiments. The interarrival time is exponentially distributed. As in the bursty workload, each sort sorts 2500 blocks of data on average and the simulator does 100 jobs from submission through completion. While we have performed no formal analysis of variance, observations of the simulator against workloads with fewer jobs show that runs of 100 jobs are more than sufficient to guarantee statistically significant results.

Chapter 5

Simulation Results and Analysis

We ran BAS against two types of workloads: bursty and steady. The bursty workload consists of bursts of one to four jobs in an otherwise quiet system. The steady workload is a more traditional workload consisting of a steady stream of jobs.

The bursty workload shows how BAS can yield improved performance in an environment where the number of concurrent jobs varies greatly over time. Here, the performance improvement ranged from about 7% to 20% depending on the mean interarrival time between the bursts. The longer the time between bursts, the greater the benefit of using BAS.

The steady workload shows that there is still a performance improvement when using BAS on a more ordinary workload. Here, BAS also outperforms our simulation of a traditional method of memory allocation, but not by as wide a margin.

In Section 5.1 we present the results of our simulations. Section 5.2 shows how these results compare with those obtained by authors of related work. Finally, we present results of experiments that show that the two variants of BAS give good results for a wide range of settings of their parameters.

5.1 Results and Analysis

In Figure 5.1 we graph the average response times of jobs for the bursty workload with different mean interarrival times between the bursts. When there is enough space between bursts, the marginal gains variant of BAS works best. However, as the bursts come

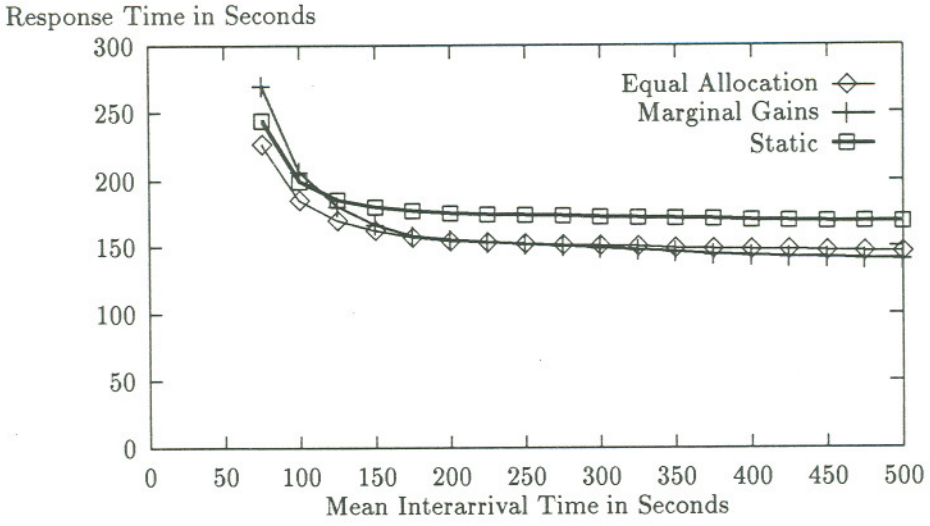


Figure 5.1: Average Response Time of All Jobs in Bursty Workload

closer together and begin to overlap, the equal allocation variant of BAS becomes the better choice. In fact, for very heavy workloads marginal gains does not even beat static allocation. The properties of marginal gains that lead to this behavior were discussed in Section 3.4.2.2.

Note that even in the region where marginal gains is better than equal allocation, the difference is very slight. Given that marginal gains is also a considerably more complex algorithm than equal allocation, and hence would be more difficult to implement, it is doubtful that this version of marginal gains is worth the trouble. On the other hand, there may be other methods of applying marginal gains to adaptive algorithms that are worth exploring.

To demonstrate that BAS is not only useful for special workloads like our bursty workload, we ran it against a more traditional workload. The results of this experiment are shown in Figure 5.2. This workload, consists of a steady stream of randomly sized

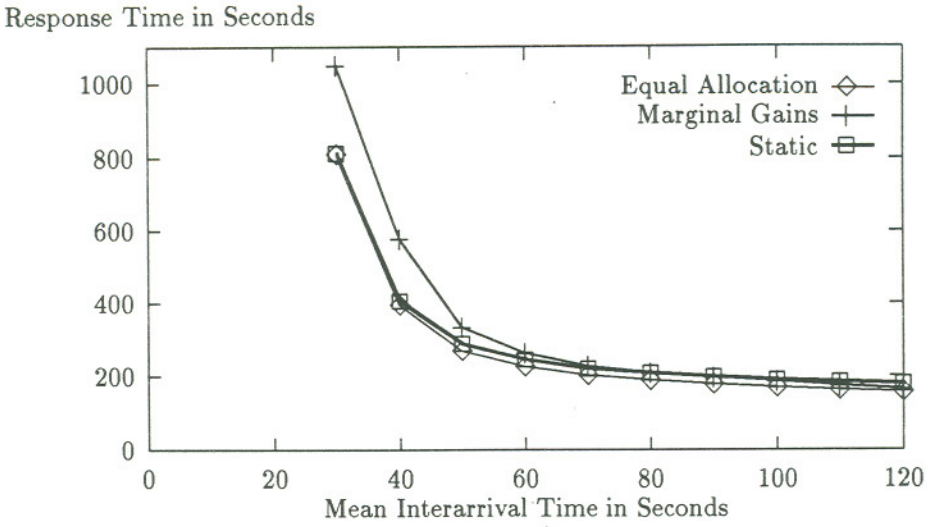


Figure 5.2: Average Response Time of All Jobs in Steady Workload

jobs. The exact parameters of the workload were described in Section 4.3.2.

For this workload, equal allocation turns out to give the best performance at any interarrival rate. We attribute this surprising result to the problems of applying marginal gains to an adaptive algorithm which we outlined in Section 3.4.2.2.

When the workload is very heavy, static allocation begins to do better than marginal gains and about as well as equal allocation. This is not surprising: the strength of memory adaptive algorithms comes from their ability to make use of memory that would otherwise be idle. When the system is close to saturation, there will be very little idle memory. In this case, the best performance is obtained by running just enough jobs to keep all the disks busy. This is precisely what a static allocation of 25% does.

The static memory allocation against which we have been comparing BAS has a parameter P_{st}. To show that we have chosen a reasonable setting for this parameter, we ran the static memory allocation algorithm against the bursty workload for P_{st} set to

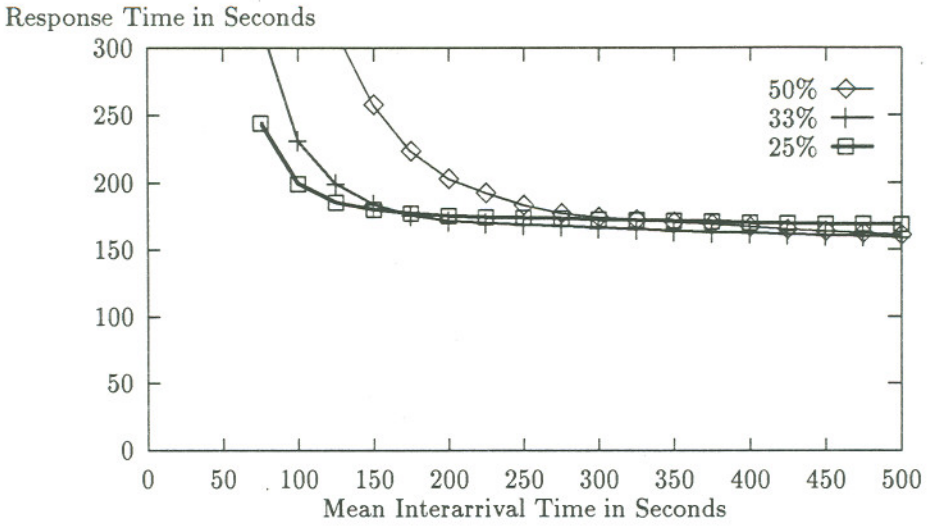


Figure 5.3: Average Response Time of All Sorts Using Static Memory Allocation

0.50, 0.33 and 0.25. These values correspond to running 2, 3 or 4 jobs concurrently.

The results of this experiment are shown in 5.3. A setting of 0.50 (50%) is obviously worse than the other two settings. At 33%, the algorithm does very slightly better than 25% for most workloads, but saturates considerably sooner than 25%. Since heavier workloads are the more interesting ones anyway, we chose to set this parameter to 25%. This setting gives the best performance for heavy workloads, and reasonable performance for lighter ones.

5.2 Comparison to Previous Work

In this section we compare our simulation results with results obtained by authors of related works.

5.2.1 DBMIN

Chou and DeWitt in their paper on DBMIN [CD85] investigate a much different workload from that investigated using BAS. In particular, the workloads investigated by Chou and DeWitt include no sorts. In contrast, the workloads for BAS consist of nothing but sorts.

Visual inspections of the graphs in Chou and DeWitt's paper on DBMIN shows that the performance improvement over the baseline results offered by DBMIN is of roughly the same magnitude as that of BAS. However, the reader must keep in mind that the very different workloads make it difficult to compare the two proposals directly.

5.2.2 MG-x-y

MG-x-y [NFS91, FNS91] is a direct extension of DBMIN and like DBMIN, the workloads used to investigate MG-x-y include no sorts.

Visual inspection of the graphs in Ng, Faloutsos and Sellis's papers on MG-x-y shows that it gives a substantial performance benefit over DBMIN. This is especially true of MG-x-y using predictive load control. However, both DBMIN and MG-x-y are really concerned with page replacement policies. In contrast, BAS is concerned with allocating workspace for sorts and other memory-intensive database operators. It may be possible to adapt some of the ideas of DBMIN for use in conjunction with BAS.

5.2.3 Return on Consumption

Visual inspection of the results of Yu and Cornell's buffer management strategy based on return on consumption [YC93] indicates that they obtained a performance improvement of as much as 30% over their baseline algorithm. In contrast, our best improvement was about 20%.

We attribute much of this to the fact that Yu and Cornell investigated hash joins. Hash joins are more sensitive to memory allocation than sorts are. Consider a sort and join too large to fit in memory, but small enough to be completed in two passes through the data (i.e. one merge phase for the sort). If the sort is executed with one fewer page

of memory, then one additional page will need to be written to a run on disk and read back in. In contrast, the hash join would have to write two additional pages to disk and read them back in, one from the inner relation and one from the outer.

5.2.4 Adaptive Hash Join

Zeller and Gray measured the performance of their adaptive hash join algorithm against that of a non-adaptive sort-merge join. Their best improvement was 59%.

While this improvement over the baseline is much greater than we obtained from BAS, it must be noted that they have switched not only from a non-adaptive to an adaptive algorithm, but also from sort-merge to hash join. Shapiro has shown that hash join is superior to merge-join for sufficiently large inputs [Sha86]. It is not clear how much of the performance improvement that Zeller and Gray found should be attributed to adaptive algorithms and how much to using hashing instead of sort-merge as their base algorithm. Also, the greater sensitivity of hash joins to differences in memory allocation discussed in the previous section applies here as well.

5.2.5 Memory-Adaptive External Sorting and Partially Preemptable Hash Joins

Pang, Carey and Livny present and compare a variety of algorithms for memory adaptive external sorting [PCL93a]. The focus of their investigation is to evaluate the relative performance of their different algorithms. Too this end, their experiments all simulate a single *adaptive* sort that is subjected to random fluctuations in its memory allocation. They do not compare any of their algorithms to a non-adaptive sort.

In contrast, our experiments investigate the merits of adaptive sorting algorithms over non-adaptive ones. While their work is very significant and relevant to ours, their experiments are too different for any direct comparison of results.

The experiments in their companion paper on partially preemptible hash joins [PCL93b] are also too different for direct comparison of the results for much the same reasons.

5.3 Sensitivity of BAS to Parameter Settings

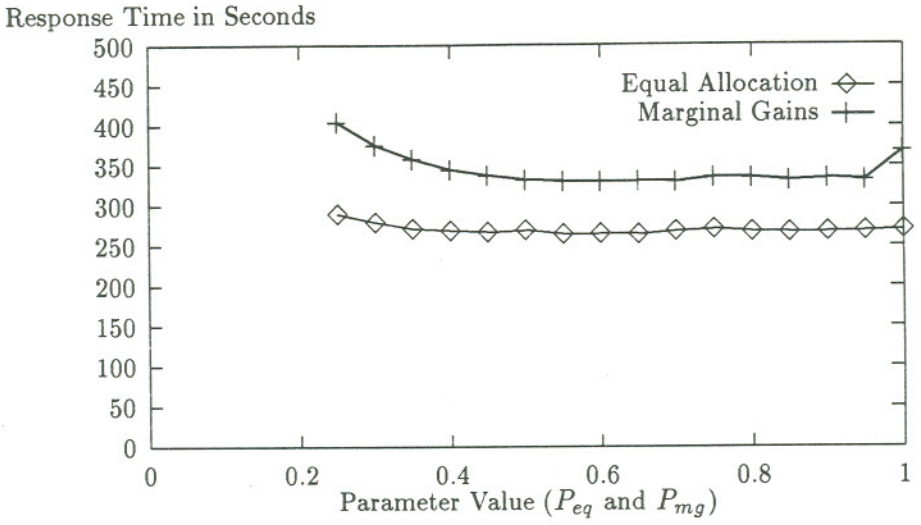


Figure 5.4: Sensitivity of Equal Allocation and Marginal Gains to Parameter Settings

One of the strengths of BAS is that although the algorithms for the memory broker are parameterized, they work nearly as well for any reasonable value of their respective parameters. To show this, we ran the simulator against the bursty workload with a mean interarrival rate between bursts of 50 seconds and varied the parameters of the algorithms (P_{eq} for equal allocation and P_{mg} for marginal gains). The results are shown in Figure 5.4. In the graph, it is clearly visible that the performance is close to the best possible (for the given algorithm) over a wide range of values for the parameter.

A similar experiment for static allocation was illustrated earlier in Figure 5.3. Note that for static allocation, the parameter *does* matter a great deal.

Chapter 6

Conclusion

In this investigation, we proposed a combination of memory adaptive algorithms and a central agent to allocate memory.

We simulated our ideas as applied to external sorting and showed that they performed consistently better than a simulation of the hueristics traditionally used in commercial DBMSs. Surprisingly, equal allocation, the simpler of our two algorithms for allocating memory, outperformed our more sophisticated marginal gains algorithm. The reason for this was discussed at length. Essentially, our application of marginal gains fails to take into account the memory fluctuations inherent in an environment of memory adaptive algorithms. Although we had anticipated this weakness, we were surprised at its severity.

On the other hand, BAS turns out to require very few modifications to the normal architecture of a DBMS. Especially in view of the effectiveness of the simpler equal allocation memory broker algorithm, it seems to offer much potential for actual implementation.

Bibliography

[CD85] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of the 11th International Conference on VLDB*, pages 127–141, 1985.

[FNS91] Christos Faloutsos, Raymond Ng, and Timos Sellis. Predictive Load Control for Flexible Buffer Allocation. Technical Report UMIACS-TR-91-34, University of Maryland at College Park, 1991.

[Ger93] Robert Gerber, 1993. Private conversation.

[Gra90a] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference*, pages 102–111, 1990.

[Gra90b] Goetz Graefe. Parallel External Sorting in Volcano. Technical Report CU-CS-459-90, Univeristy of Colorado at Boulder, 1990.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, 1973.

[Liv90] Miron Livny. *DeNet User's Guide, Version 1.5*, 1990.

[NFS91] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible Buffer Allocation Based on Marginal Gains. *Proceedings of the 1991 ACM SIGMOD International Conference*, pages 387–396, 1991.

[NKT88] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th International Conference on VLDB*, 1988.

[PCL93a] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-Adaptive External Sorting. In *Proceedings of the 19th International Conference on VLDB*, 1993.

[PCL93b] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially Preemptible Hash Joins. *SIGMOD*, 1993.

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 3(11):239–264, September 1986.

[Sto81] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), 1981.

[Ver86] A. Inkeri Verkamo. Comparison of External Sorting and Internal Sorting in Virtual Memory. *Performance Evaluation*, 6(2):135–145, July 1986.

[YC93] Philip S. Yu and Douglas W. Cornell. Buffer Management Based on Return on Consumption in a Multi-Query Environment. *VLDB Journal*, 1993.

[ZG90] Hansjörg Zeller and Jim Gray. An Adaptive Hash join Algorithm for Multiuser Envronments. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of the 16th International Conference on VLDB*, pages 186–197, 1990.

Appendix A

Derivation of Introductory Example

In this appendix we present the numbers behind the graphical example in the introduction. Recall that the example consisted of two sorts: S_1 (the smaller sort, meaning that it sorts less data) and S_2 (the larger sort). This example was constructed using the values listed below. As elsewhere in this text, memory is measured in blocks, time is measured in number of IO operations completed, and CPU time is ignored. In this example we also assume that there is no disk contention.

Total Memory

The amount of memory available for input buffers is 12 blocks. There would also be two additional blocks used to buffer output for the two sorts. We count these separate from the 12 input buffers to simplify the calculations.

Sort S_1

This sort is the smaller one. It sorts 6 blocks of data starting at time 0.

Sort S_2

This sort is the larger one. It sorts 16 blocks of data, and also starts at time 0.

A.1 Static Memory Allocation

Since S_1 uses so little memory, a reasonable static allocation is to give S_1 as much memory as it can use and leave the rest for S_2. This also happens to be exactly half

the memory, so this example matches what the equal allocation variant of the simulator would do with this workload.

In any case, S_1 gets 6 blocks of memory in this example, making it an internal sort. Thus, the sequence of events for S_1 is:

Time 0-5

Read all six blocks to be sorted from the input. Six IO operations.

Time 5

Sort the data.

Time 6-11

Write all six blocks of sorted data to the final output. Six IO operations.

S_1 takes a total of 12 IO operations.

The 6 blocks of memory not used by S_1 are allocated to S_2. S_2 sorts 16 blocks of data, so it will need to create 3 runs of sorted data on the disk. Using the run creation optimization described in Section 3.3.3.1, S_2 can save 4 IO operations by holding a run of 2 blocks in memory rather than writing them to disk. As a result, the third disk run will contain only 2 blocks of data, rather than the 4 that would happen without the optimization. The sequence of events for S_2 is as follows:

- *Time 0-11*

  Create a sorted run of 6 blocks on disk. 12 IO operations.

- *Time 12-23*

  Create another sorted run of 6 blocks on disk. 12 IO operations.

- *Time 24-27*

  Create a run of 2 blocks on disk. 4 IO operations.

- *Time 28-29*

  Create a sorted run of 2 blocks *in memory*. 2 IO operations.

- *Time 30-59*

  Merge disk runs with memory run and write all of it to final output. 30 IO operations (14 blocks read from disk, all 16 written to final output).

For a total of 60 IO operations.

The grand total of both sorts is 72 IO operations.

A.2 Dynamic Memory Allocation

Using dynamic memory allocation, it is possible for S_2 to use all of the available memory during most of its lifetime. Only during the execution of S_1 does S_2 restrict its memory allocation to 6 blocks for the duration of creating one run. To best present the sequence of events that occurs when S_1 and S_2 run using dynamic algorithms, we recreate the space-time diagram of Figure 1.1(b) in Figure A.1, this time showing the internal events.

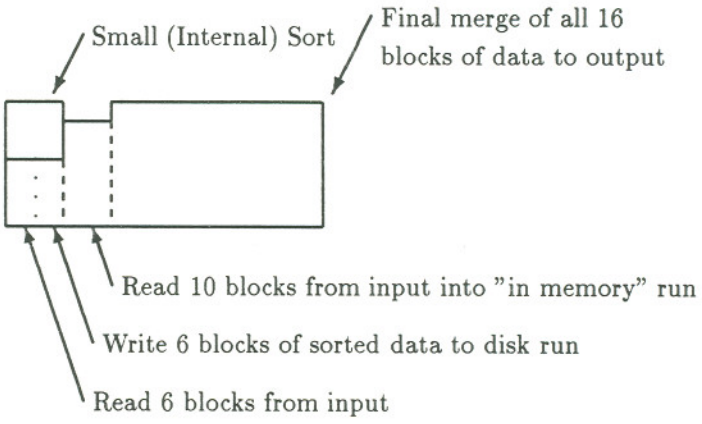


Figure A.1: Space Time Diagram of Adaptive Sorts Showing Run Creation

The total number of IO operations in this case is 56. This is a 29% improvement over the number required when using static memory allocation.

Appendix B

Marginal Gains Formula and its Derivation

The following variables will be used in the discussion and formulas of this appendix:

E Existing runs of sorted data on disk. (Does not include runs not yet created, when considering a sort still in progress.)

B Total blocks of data to sort.

b Blocks remaining to be sorted.

M Memory allocation (measured in blocks).

R_0 Total number of level-0 runs.

Marginal gains in BAS measures the change in the number of I/O operations which would be caused by each additional block of memory allocated. It would be impractical to try to calculate this value exactly because it would require simulating the rest of the sort's lifetime. Instead, we use an analytical approximation based on the number of times the data is passed. We have checked the formula for this approximation by comparing it against hand simulation of various examples. It is exact in cases where the data is passed an integral number of times. In the remaining cases it gives a value that is slightly higher than the true value.

To illustrate the derivation of the formula, consider as an example sorting 64 blocks of data. If 8 input buffers are used, then the sort must read and write all of the data two

times, once to split all of the input into level-0 runs on disk, and once more to merge all of the level-0 runs into the final output. Sorting the same 64 blocks of data using 4 input buffers would require three complete passes through the data.

In general, the number of passes through the data will be

$$\log_M B \tag{B.1}$$

But this only works right before any of the input has been consumed. We want to be able to estimate the number of I/O operations remaining at an arbitrary checkpoint. At any check point there remain $\frac{b}{B}$ passes through the data to finish creating the level-0 runs. After all the level-0 runs have been created, there will be

$$\log_M R_0 \tag{B.2}$$

additional passes through the data to merge the R_0 level-0 runs into the "single run", which is the final output. R_0 will be E, the number of existing runs plus $\frac{b}{M}$ level-0 runs yet to be created with the current memory allocation. Thus, the total number of passes remaining to be made through the data at any checkpoint is:

$$\frac{b}{B} + \log_M \left(E + \frac{b}{M} \right) \tag{B.3}$$

The total amount number of IO operations is the number of passes through the data times the number of blocks of data times 2 (read and write):

$$2B \left[\frac{b}{B} + \log_M \left(E + \frac{b}{M} \right) \right] \tag{B.4}$$

This can be simplified:

$$2B \left[\frac{b}{B} + \log_M \left(\frac{EM + b}{M} \right) \right] \tag{B.5}$$

$$2B \left[\frac{b}{B} + \log_M (EM + b) - \log_M (M) \right] \tag{B.6}$$

$$2B \left[\frac{b}{B} + \log_M (EM + b) - 1 \right] \tag{B.7}$$

$$2B \left[\log_M (EM + b) + \frac{b - B}{B} \right] \tag{B.8}$$

$$2\left[B\log_M(EM+b)+b-B\right] \tag{B.9}$$

The marginal gain is the derivative of this with respect to memory:

$$\frac{d}{dM}2\left[B\log_M(EM+b)+b-B\right] \tag{B.10}$$

This is not immediately differentiable because the independent variable (M) occurs as the base of the log function. We use the laws of logarithms to convert to natural logs:

$$\frac{d}{dM}2\left[B\frac{\ln(EM+b)}{\ln(M)}+b-B\right] \tag{B.11}$$

Now we can differentiate:

$$2B\cdot\frac{d}{dM}\frac{\ln(EM+b)}{\ln(M)} \tag{B.12}$$

$$2B\left[\frac{1}{\ln(M)}\cdot\frac{d}{dM}\ln(EM+b)-\frac{\ln(EM+b)}{(\ln(M))^2}\cdot\frac{d}{dM}\ln(M)\right] \tag{B.13}$$

$$2B\left[\frac{1}{\ln(M)}\frac{E}{EM+b}-\frac{\ln(EM+b)}{(\ln(M))^2}\frac{1}{M}\right] \tag{B.14}$$

$$\frac{2B}{\ln(M)}\left[\frac{E}{EM+b}-\frac{\ln(EM+b)}{M\ln(M)}\right] \tag{B.15}$$

The marginal gains variant of BAS uses this formula to estimate marginal gains.

Biographical Note

Kevin Beck was born in Missoula, MT August 10, 1962. He received his Bachelor of Science in Computer Science (Systems Option) from Montana State University in March 1985. Since moving to Portland in 1985 Kevin has worked as a computer programmer. His main area of interest in Computer Science is database internals, especially performance issues.